Fundamentals of Programming 2

Singly Linked and Doubly Linked Lists

Arkadiusz Chrobot

Department of Information Systems

April 25, 2025

Notes

Outline

Notes

Introduction

In the last lecture, we have discussed two dynamic data structures, the stack and the queue. They are a special case of more generic data structures called lists. In lists, a node can be added or removed at any location. They are also linear structures, meaning that any node can have at most one successor and one predecessor.
In this lecture we are going to discuss two kinds of lists, the singly linked list and the doubly linked list. To explain how they work and how they are build we will use two programs applying the list to store natural numbers (one node — one number), in a non-decreasing order.

Notes

Singly Linked List

Node Data Type

```
#include<stdio.h>
#include<stdlib.h>

struct list_node
{
    int data;
    struct list_node *next;
} *list_pointer;
```


Notes

Singly Linked List

Node Data Type

The first of the programs demonstrates how to use the singly linked list. In the foregoing slide the first part of its code is shown. There are included two header files, `stdio.h` and `stdlib.h`. The first one allows the program to use the `printf()` function, and the second one enables it to apply functions for managing the heap.
Just like the stack or queue, the list needs a data type for its node to be defined. Its definition is in lines 4–8. The `data` member is for storing a natural number, although its data type allows it to store integer number. The `next` member is a pointer, that makes it possible to link the node with another.
In the line no. 8 is declared *the list pointer* (the `list_pointer` variable). It is a pointer that always should point to the first node of the list or be empty, if the list is also empty. In this program the list pointer is a global variable, meaning that its default value is zero or `NULL`. It is a correct value, because the list is initially empty.

Notes

Singly Linked List

The next two slides contain declarations of functions that together are responsible for adding a new node to the list, in such a way, that the numbers stored in the structure form a non-decreasing sequence. The pointer to the pointer parameter used in the functions allows them to handle any of the cases of adding a new node to the singly linked list:

1. adding to the empty list,
2. adding at the front of the list,
3. adding inside the list,
4. adding at the end of the list.

Aside from generally describing the code of the functions we are going to analyse how the functions work in all of these cases, to better understand them.

Notes

Singly Linked List

The `create_and_add_node()` Function

```
1  int create_and_add_node(struct list_node **node, int number)
2  {
3      struct list_node *new_node = (struct list_node
        ↪  *)malloc(sizeof(struct list_node));
4      if(!new_node)
5          return -1;
6      new_node->data = number;
7      new_node->next = *node;
8      *node = new_node;
9      return 0;
10 }
```


Notes

Singly Linked List

The `create_and_add_node()` Function

The `create_and_add_node()` function is responsible for creating a new node and adding it to the list. It takes two arguments. The first one is passed by the `node` parameter and it is *and address of a pointer to the list node* before which a new one should be inserted. The second argument is a number that should be stored in the new node. First, the function allocates memory for the new node (line no. 3) and checks if this operation has been successful (line no. 4). If not, the function returns `-1` and exits (line no. 5). Otherwise, it assigns to the `data` member of the new node the number that is passed by the second parameter (line no. 6). Then it assigns to the `next` member of the new node the address of the node *stored in the pointer pointed by the* **`node`** *parameter* (line no. 7). Finally, the address of the new node is stored in the pointer pointed by the `node` (line no. 8) and the function exits returning `0`.

Notes

Singly Linked List

The add_node() Function

```
1 int add_node(struct list_node **node, int number)
2 {
3     while(*node != NULL && (*node)->data < number)
4         node = &(*node)->next;
5     return create_and_add_node(node, number);
6 }
```


Notes

Singly Linked List

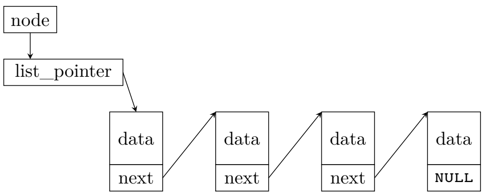
The add_node() Function

The `create_and_add_node()` function is invoked in the program by `add_node()`. Only the latter should be used for adding a new node to the list in the rest of the program. It takes as the arguments the address of the list pointer, which is passed by the `node` parameter, and the number, passed by the `number` parameter, that should be stored in the new node. First, the function performs the `while` loop (lines no. 3–4), to find a node in the list, *before which* the new one should be inserted. This loop stops when the pointer pointed by the `node` parameter is empty, or when the number stored in the node pointed by the pointer, whose address is stored in the `node` parameter, is greater or equal to the number passed by the `number` parameter. Please notice, that in each iteration of the `while` loop, *the address of the* `next` *member*, of the ensuing node in the list is assigned to the `node` parameter. The next slide illustrates how the list is traversed in the loop.

Notes

Singly Linked List

Traversing The List

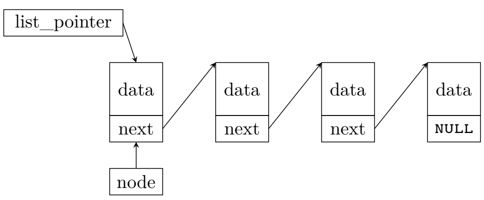


Traversing the singly linked list

Notes

Singly Linked List

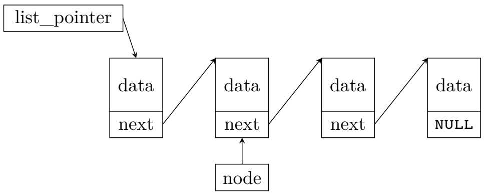
Traversing The List



Traversing the singly linked list

Notes

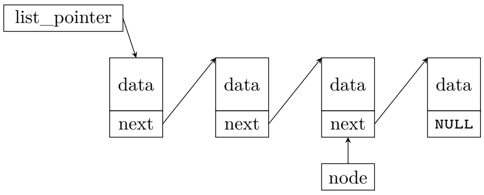
Singly Linked List
Traversing The List



Traversing the singly linked list

Notes

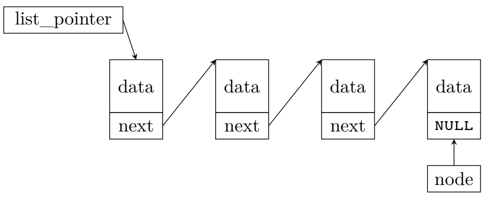
Singly Linked List
Traversing The List



Traversing the singly linked list

Notes

Singly Linked List
Traversing The List



Traversing the singly linked list

Notes

Singly Linked List
The `add_node()` Function

After the `while` loop stops, the `add_node()` functions invokes the `create_and_add_node()` function and exits returning the same value as the former function.
These short descriptions don't explain thoroughly how the functions perform the operation of adding a node to the list. Let's analyse than each of the mentioned cases, starting with the one where the list is empty.

Notes

Singly Linked List

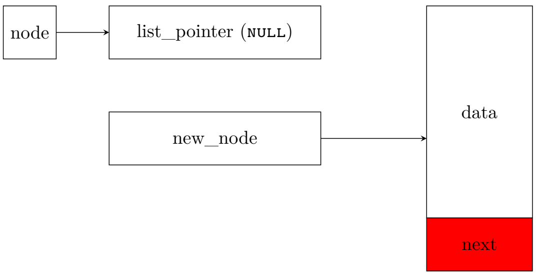
Adding The First Node

The `while` loop stops at once, because the `list_pointer`, whose address is passed to the `add_node()` function is empty — the `*node != NULL` expression is false. The `create_and_add_node()` is invoked, which allocates memory for the new node, and if the operation is successful, it stores in the node's `data` member the number passed by the `number` parameter (line no. 6, slide no. 7). Next, the function assigns to the `next` member of the new node the address stored in the pointer pointed by the `node` parameter (line no. 7). Let's remind, that in this case it is the list pointer, which is empty. It means that the `NULL` is assigned to the member. This value is valid, because the new node becomes, at the same time, the first and the last node in the list. The statement in the line no. 8 assigns the address of the new node to the list pointer, making the `list_pointer` to point to this node, and creating a singly linked list with only one node. The entire operation is illustrated in the next slide.

Notes

Singly Linked List

Adding The First Node

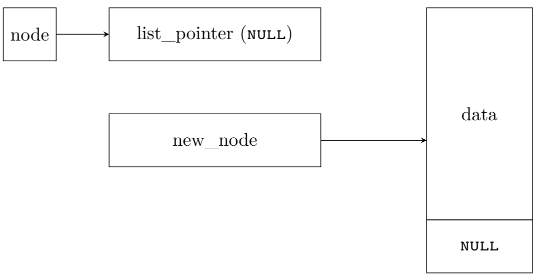


Before the line no. 7 of the `create_and_add_node()` is performed

Notes

Singly Linked List

Adding The First Node

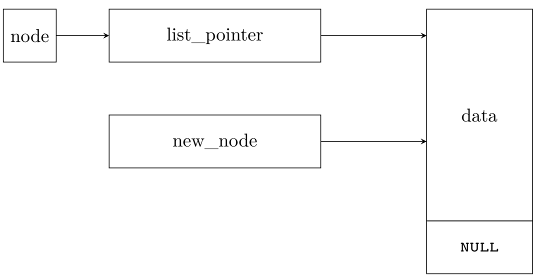


Before the line no. 8 of the `create_and_add_node()` is performed

Notes

Singly Linked List

Adding The First Node



After the line no. 8 of the `create_and_add_node()` is performed

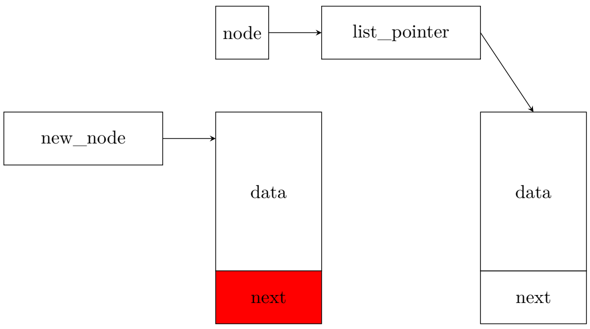
Notes

Singly Linked List
Adding At The Front

In this case the `list_pointer` stores the address of the list first node, but the number that is in it, is greater or equal to the one that should be stored in the new node. It means that the `while` loop stops at once, because the `(*node)->data < number` is false, and the `create_and_add_node()` function is invoked. This function creates a new node and, if the operation is successful, it assigns to the `data` member of the new node the number passed by the `number` parameter. Next, the address from the pointer pointed by the `node` parameter is assigned to the `next` member of the new node (line no. 7, slide no. 7). Just like in the former case, this pointer is the `list_pointer`, but this time it stores the address of the list first node. It means that now the `next` member of the new node points to this former first node. In the line no. 8 of the `create_and_add_node()` function the address of the new node is stored in the list pointer. It is necessary, because this pointer should point to the first node of the list, which now is the new node.

Notes

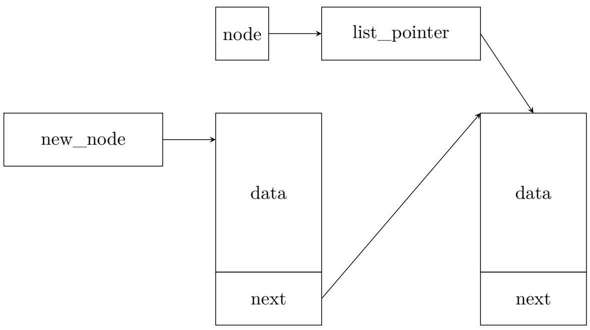
Singly Linked List
Adding At The Front



Before the line no. 7 of the `create_and_add_node()` is performed

Notes

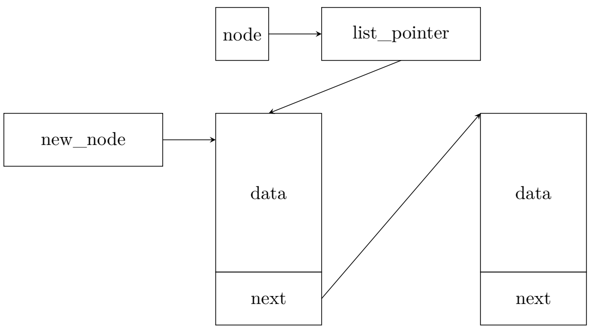
Singly Linked List
Adding At The Front



Before the line no. 8 of the `create_and_add_node()` is performed

Notes

Singly Linked List
Adding At The Front



After the line no. 8 of the `create_and_add_node()` is performed

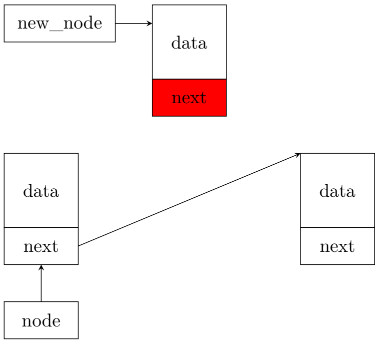
Notes

Singly Linked List
Adding Inside

In this case the new node is added inside the list, in other words, *between* two nodes that already are in the list. This time the `while` loop in the `add_node()` stops when the `node` parameter stores the address of some node's `next` member that points to another node storing the number greater or equal to the one that is passed by the `number` parameter. It means, that, just like in the case of adding the new node at list's front, the `(*node)->data < number` is false. The `create_and_add_node()` function is called. It tries to create a new node and, if the operation is successful, stores in the node the number passed by the `number` parameter (line no. 6). Next, it assigns to the new node's `next` member the address stored in the `next` member pointed by the `node` parameter. It is the address of the node *before* which the new should be inserted (line no. 7). In the line no. 8, the address of the new node is stored in the `next` member pointed by the `node` parameter.

Notes

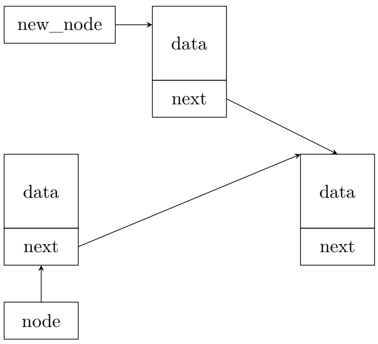
Singly Linked List
Adding Inside



Before the line no. 7 of the `create_and_add_node()` is performed

Notes

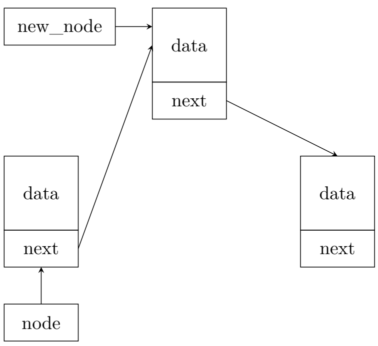
Singly Linked List
Adding Inside



Before the line no. 8 of the `create_and_add_node()` is performed

Notes

Singly Linked List
Adding Inside



After the line no. 8 of the `create_and_add_node()` is performed

Notes

Singly Linked List
Adding At The End

If the new node stores a number greater than any number already stored in the list, then it has to be added at the end of the list. In this case the `while` loop in the `add_node()` function stops when the `*node != NULL` expression is false, meaning that the `node` parameter points to the last node's `next` member that stores the `NULL` value. The address of this member is passed to the `create_and_add_node()` function, together with the number that should be stored in the new node. The latter function tries to create a new node, just like in the previous cases, and if the operation is successful, it stores the number in this node (line no. 6, slide no. 7). Next, it assigns to the `next` member of new node the `NULL` value stored in the pointer pointed by the `node` parameter. It is a correct value for this member, because the node will be added at the end of the list.

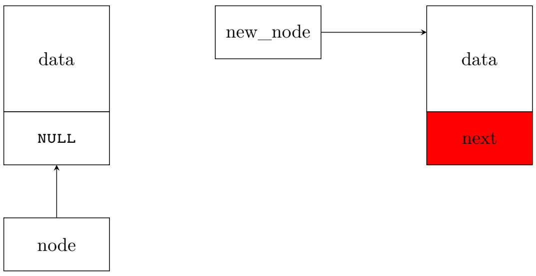
Notes

Singly Linked List
Adding At The End

In the line no. 8 the `create_and_add_node()` function assigns to the `next` member pointed by the `node` parameter the address of the new node. The former last node of the list begins to point, with its `next` member, the current last node in the list.

Notes

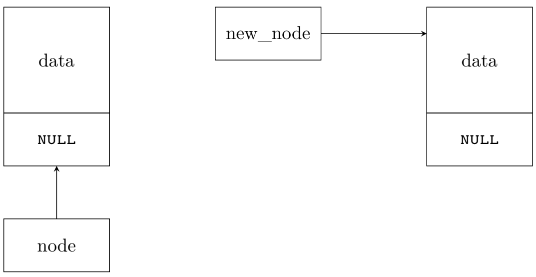
Singly Linked List
Adding At The End



Before the line no. 7 of the `create_and_add_node()` is performed

Notes

Singly Linked List
Adding At The End

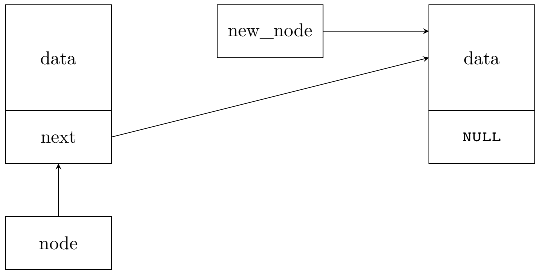


Before the line no. 8 of the `create_and_add_node()` is performed

Notes

Singly Linked List

Adding At The End



After the line no. 8 of the `create_and_add_node()` is performed

Notes

Singly Linked List

The `delete_node()` Function

```
1  void delete_node(struct list_node **node, int number)
2  {
3      while(*node && (*node)->data != number)
4          node = &(*node)->next;
5      if(*node) {
6              struct list_node *temporary = (*node)->next;
7              free(*node);
8              *node = temporary;
9          }
10 }
```


Notes

Singly Linked List

The `delete_node()` Function

The `delete_node()` function is responsible for removing a single element from the list, that stores in the `data` member the same number, as it is passed by the `number` parameter. If the function doesn't find such a node in the list then it exits without removing anything from the list. If there is more than one node in the list that stores this number, then the function removes the first of them.

The definition of this function is more concise that the two described earlier. The `delete_node()` function, just like the `add_node()` takes as the first argument the address of the list pointer. As the second one is passed the number that the node for removing should store. The function returns nothing.

Notes

Singly Linked List

The `delete_node()` Function

First, it performs the `while` loop that is quite similar to the loop in the `add_node()` function (lines no. 3–4). The only difference is in the operator applied in the second expression of the loop's condition. The loop stops when the pointer pointed by the `node` parameter is empty or points to a node that stores the number passed by the `number` parameter. The first case means, that there is no node in the list, that should be removed. In the second case such an operation should be performed. To distinguish these cases, the function checks if the pointer pointed by the `node` is not empty (line no. 5). If the condition is met, then it assigns to the `temporary` variable the address stored in the `next` member of the node pointed by the pointer whose address is, in turn, stored in the `node` parameter (line no. 6). Then it removes this node (line no. 7) and assigns to the pointer pointed by the `node` parameter the address stored in the `temporary` pointer (line no. 8).

Notes

Singly Linked List

Funkcja delete_node()

Notes

The short description, from the previous slide, doesn't discuss in the details of the work of the `delete_node()` function. Let's analyse its behaviour for the three most interesting cases:

1. deleting the first node of the list,
2. deleting an inner node in the list,
3. deleting the last node in the list.

Singly Linked List

Deleting The First Node

Notes

In the first case the `while` loop in the `delete_node()` function stops at once, because the `(*node)->data != number` is false. It means that the number it searches for is in the first node, and that node should be deleted. It also means that the condition in the line no. 5 is fulfilled. The `delete_node()` function assigns to the `temporary` variable the address stored int the `next` member of node pointed by the pointer whose address is, in turn, stored in the `node` parameter (line no. 6). In this case, this pointer is the list pointer (the `list_pointer`), and this node is the list first node. Therefore, in the `temporary` variable is stored the address of the *second node* in the list (provided it exists). The `delete_node()` function releases the memory allocated for the first node (line no. 7) and assigns to the pointer pointed by the `node` parameter the address stored in the `temporary` pointer (line no. 8).

Singly Linked List

Deleting The First Node

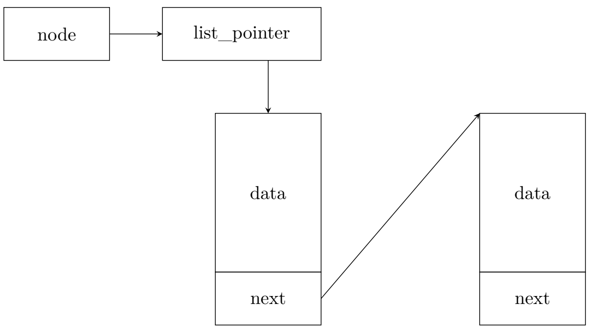
Notes

Please notice, that the function works correctly also when the first node is at the same time the last in the list. In this case, in the line no. 6, the `NULL` value is assigned to the `temporary` variable, which then will be assigned to the list pointer (line no. 8). It is an expected outcome, because after the only node is removed the list becomes empty and so should the `list_pointer` variable be.
The next slide illustrates the removal of the first node from a list that has at least two nodes, by the `delete_node()` function.

Singly Linked List

Deleting The First Node

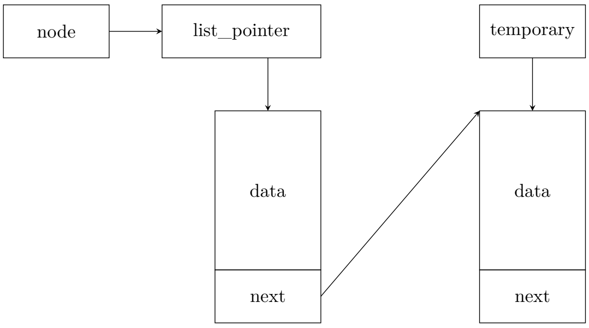
Notes



Before the line no. 6 of the `delete_node()` is performed

Singly Linked List

Deleting The First Node

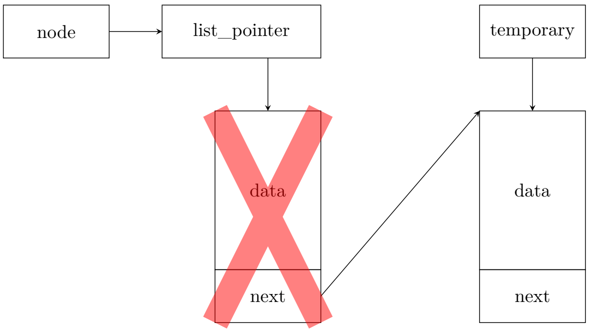


Before the line no. 7 of the `delete_node()` is performed

Notes

Singly Linked List

Deleting The First Node

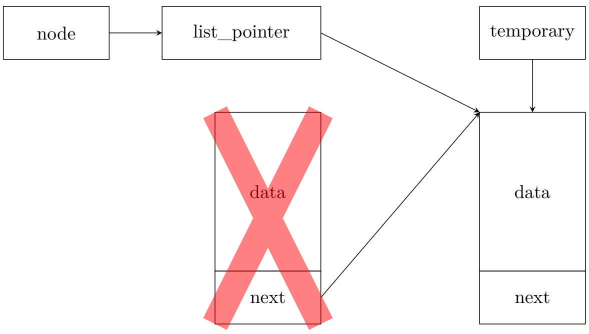


Before the line no. 8 of the `delete_node()` is performed

Notes

Singly Linked List

Deleting The First Node



After the line no. 8 of the `delete_node()` is performed

Notes

Singly Linked List

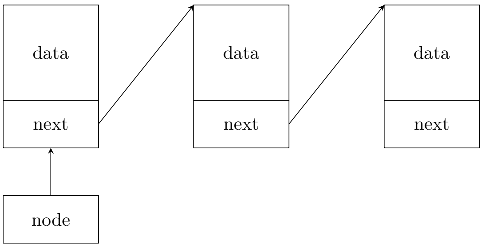
Deleting an Inner Node

In the case when the node to be removed is inside the list, the `while` loop in the `delete_node()` stops when the `node` parameter stores the address of the `next` member that points to the node storing the number passed by the `number` parameter. Yet again the `(*node)->data != number` is false, but the condition in the line no. 5 is met. The function assigns to the `temporary` pointer the address of the node in the list that is next to the one that should be removed (line no. 6). Then, it disposes the memory allocated for the latter node (line no. 7) and assigns to the `next` member pointed by the `node` parameter the address stored in the `temporary` variable. Thus, the node, that preceded the one which has been removed, starts pointing to the one that succeeded the removed node. The list stays coherent.

Notes

Singly Linked List

Deleting an Inner Node

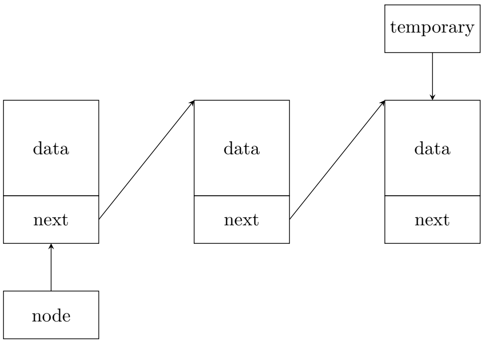


Before the line no. 6 of the `delete_node()` is performed

Notes

Singly Linked List

Deleting an Inner Node

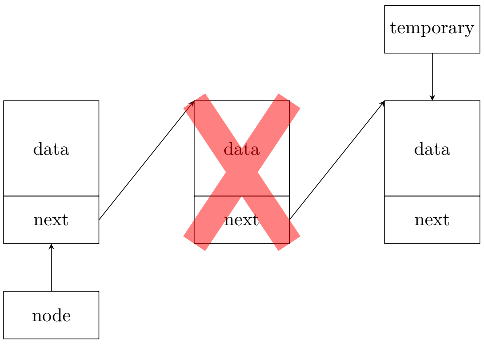


Before the line no. 7 of the `delete_node()` is performed

Notes

Singly Linked List

Deleting an Inner Node

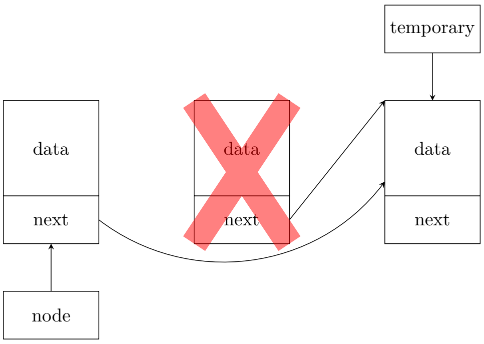


Before the line no. 8 of the `delete_node()` is performed

Notes

Singly Linked List

Deleting an Inner Node



After the line no. 8 of the `delete_node()` is performed

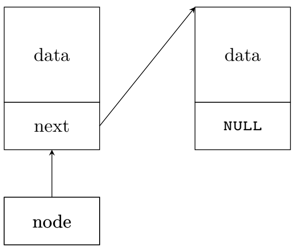
Notes

Singly Linked List
Deleting The Last Node

In the third case the `while` loop in the `delete_node()` function also stops when the `node` parameter stores the address of the `next` member that points to the node where the number passed by the `number` parameter is stored. Once again the `(*node)->data != number`) is not met, but the number that the loop was searching for is in the last node of the list. After the `delete_node()` function verifies the condition in the line no. 5, it assigns to the `temporary` variable the address stored in the `next` member of the node that is pointed by the pointer whose address, in turn, is stored in the `node` parameter (line no. 6). This pointer is the `next` member of the *last but one node* of the list. The address stored in the `temporary` variable is actually the `NULL` value. The `delete_node()` function frees the memory allocated for the last node (line no. 7) and assigns to the `next` member of the node that so far was the last but one, the `NULL` value (line no. 8) — now that node is the last in the list.

Notes

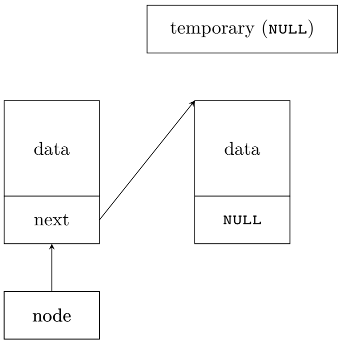
Singly Linked List
Deleting The Last Node



Before the line no. 6 of the `delete_node()` is performed

Notes

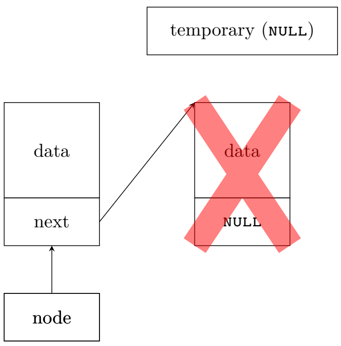
Singly Linked List
Deleting The Last Node



Before the line no. 7 of the `delete_node()` is performed

Notes

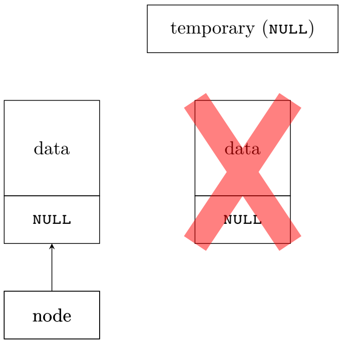
Singly Linked List
Deleting The Last Node



Before the line no. 8 of the `delete_node()` is performed

Notes

Singly Linked List
Deleting The Last Node



After the line no. 8 of the `delete_node()` is performed

Notes

Singly Linked List
The `print_list()` Function

```
1 void print_list(struct list_node *node)
2 {
3     while(node) {
4         printf("%d ", node->data);
5         node = node->next;
6     }
7     puts("");
8 }
```


Notes

Singly Linked List
The `print_list()` Function

The `print_list()` function displays numbers stored in the list. It takes only one argument, which is the address of the first node in the list. If the list is empty, the function won't print anything. The parameter of the list is a first level pointer, because there is no need to modify the list pointer or the `next` member of the nodes inside this function. The `print_list()` doesn't return any value. Inside that function a `while` loop is performed (lines no. 3–6), that traverses the list. It does so as long as the value of the `node` parameter is different than `NULL`. Inside the loop the number, from the `data` member of the node currently pointed by the `node` parameter, is displayed (line no. 4), and then the `node` is assigned the address stored in the `next` member of the node that it currently points to. In other words, the `node` parameter is "advanced" to the next node in the list (line no. 5).

Notes

Singly Linked List
The `remove_list()` Function

```
void remove_list(struct list_node **node)
{
    while(*node) {
        struct list_node *temporary = (*node)->next;
        free(*node);
        *node = temporary;
    }
}
```


Notes

Singly Linked List

The `remove_list()` Function

The `remove_list()` function is responsible for deleting the list, which means that is has to release memory allocated for all nodes. The `node` parameter of that function is a pointer to a pointer, because `remove_list()` needs to modify the entire structure of the list. The parameter is used for passing the address of the *address of the list pointer*. The function doesn't return any value.
To delete the list the `remove_list()` function performs a `while` loop (lines no. 3–7), similar to the one in the `print_list()` function. However, this one checks in its condition if the *pointer pointed by the `node` parameter* is not empty. In the loop body the address on the next node in the list (provided, it exists) is stored in the `temporary` variable (line no. 4) and the node pointed by the pointer, whose address is stored in the `node` parameter, is deleted (line no. 5). Then, to the pointer that pointed to the deleted node is assigned the address stored in the `temporary` variable. In that way the function removes all nodes of the list.

Notes

Singly Linked List

The `main()` Function, part 1

```c
int main(void)
{
    for(int i=1; i<5; i++)
        if(add_node(&list_pointer,i)==-1)
            fprintf(stderr,"Error adding a node with %d number to the list!\n",i);
    for(int i=6; i<10; i++)
        if(add_node(&list_pointer,i)==-1)
            fprintf(stderr,"Error adding a node with the %d number to the list!\n",i);
    print_list(list_pointer);
    if(add_node(&list_pointer,0)==-1)
        fprintf(stderr,"Error adding a node with the %d number to the list!\n",0);
    print_list(list_pointer);
```

Notes

Singly Linked List

Funkcja `main()`, part 2

```c
    if(add_node(&list_pointer,5)==-1)
        fprintf(stderr,"Error adding a node with the %d number to the list!\n",5);
    print_list(list_pointer);
    if(add_node(&list_pointer,7)==-1)
        fprintf(stderr,"Error adding a node with the %d number to the list!\n",7);
    print_list(list_pointer);
    if(add_node(&list_pointer,10)==-1)
        fprintf(stderr,"Error adding a node with the %d number to the list!\n",10);
    print_list(list_pointer);
    puts("");
```

Notes

Singly Linked List

Funkcja `main()`, part 3

```c
    delete_node(&list_pointer,0);
    print_list(list_pointer);
    delete_node(&list_pointer,1);
    print_list(list_pointer);
    delete_node(&list_pointer,1);
    print_list(list_pointer);
    delete_node(&list_pointer,4);
    print_list(list_pointer);
    delete_node(&list_pointer,7);
    print_list(list_pointer);
    delete_node(&list_pointer,10);
    print_list(list_pointer);
    remove_list(&list_pointer);
    return 0;
}
```

Notes

Singly Linked List
The `main()` Function

In the `main()` function all earlier defined function (excluding the `create_and_add_node()`, which is called by the `add_node()` function) are invoked, to test if they work correctly. In the first `for` loop, nodes with natural numbers ranging form 1 to 4 (lines 3–5, slide no. 37) are added to the list. Please notice the way of invoking the `add_node()` function. It is checked in each iteration of the `for` loop if the function has returned the `-1` value, which would mean that some exception has occurred. In that case the program would display an appropriate message. Also please notice the first argument of this function — as it was described before, it is the address of the list pointer. The second `for` loop (lines no. 6–8, slide no. 37) adds to the list nodes that store numbers ranging from 6 to 9. Then, the `print_list()` function is invoked (line no. 9, slide no. 37), that should display all aforementioned numbers.

Notes

Singly Linked List
The `main()` Function

Next, in the `main()` function are added to the list nodes that store the numbers 0 (lines no. 10–11, slide no. 37), 5 (lines no. 1–2, slide no. 38), 7 (lines no. 4–5, slide no. 38) and 10 (lines no. 7–8, slide nr 38). After each such an operation the `print_list()` function is called. Numbers in the new nodes are specifically chosen to test if the `add_node()` function correctly adds nodes at the front of the list (the 0 number), inside the list (the 5 number), inside the list, but if there is another node storing the same number (the 7 number) and at the end of the list (the 10 number).

Notes

Singly Linked List
The `main()` Function

After finishing adding the nodes, the `main()` function starts removing them with the help of the `delete_node()` function. First, it deletes the node that stores 0 (line no. 1, slide no. 39), to check, if the `delete_node()` function correctly removes the first node in the list. Next, the node storing the 1 is deleted (line no. 3, slide no. 39). Again it is the first node in the list. Then the `main()` function tries to delete the node storing 1 once more. This time there is no such node (line no. 5, slide no. 39), but it allows us to verify if the `delete_node()` correctly handles such a case. Then, the node that stores 4 is deleted (line no. 7, slide no. 39), that is inside the list. The node containing 7 is removed in the line no. 9, slide no. 39. It is also an inner node, but stores the same number as another node. Finally, the node storing 10, that is the last node in the list, is deleted (line no. 11, slide no. 39). After each removal of a node the `print_list()` function is invoked, to show the changes. Eventually, the `remove_list()` is called to delete the remaining nodes.

Notes

Doubly Linked List

The construction of the doubly linked list is very similar to the singly linked list. The only difference is that each element of the doubly linked list has an additional member that stores the address of the preceding node, with the exception of the first node in the list, that doesn't have a predecessor.
The descriptions of functions defined in the second program are focused on the difference between them and their equivalents in the first program, because both programs are very similar.

Notes

Doubly Linked List
Node Data Type

```c
#include<stdio.h>
#include<stdlib.h>

struct list_node
{
    int data;
    struct list_node *previous, *next;
} *list_pointer;
```


Notes

Doubly Linked List
Node Data Type

The beginning of the second program is almost the same as the first one. The difference is in the data type of the list node. It has an additional member, called `previous`. It is a pointer where the address of the node's predecessor is stored. In the case of the list first node, this member stores the `NULL` value.
It is possible to create a double linked list with only one pointer member in each node. Such a list is called an XOR-list. It allows the programmer to save space in the memory that would be occupied by the additional member, however it requires complex operations for traversing and adding or deleting nodes. Therefore, it is rarely used and it is not discussed in this lecture in details.

Notes

Doubly Linked List
The `create_and_add_node()` Function

```c
1  int create_and_add_node(struct list_node **node, struct
    ↪  list_node* preceding, int number)
2  {
3      struct list_node *new_node = (struct list_node
        ↪  *)malloc(sizeof(struct list_node));
4      if(!new_node)
5          return -1;
6      new_node->data = number;
7      new_node->next = *node;
8      new_node->previous = preceding;
9      if(*node)
10         (*node)->previous = new_node;
11     *node = new_node;
12     return 0;
13 }
```


Notes

Doubly Linked List
The `create_and_add_node()` Function

The `create_and_add_node()`, when compared with its equivalent for the singly linked list, has an additional pointer parameter (called `preceding`), that is used for passing the address of a node that contains the `next` member pointed by `node` parameter, or the `NULL` value, depending on the location, where the new node should be added to the list. The function has to take into account the `previous` member. That's why in the line no. 8 it assigns to the member the address stored in the `preceding` pointer. Additionally, it verifies if there is a node that should be the successor of the new one in the list (line no. 9), and if it is so, it stores the address of the new node in the `previous` member of that node (line no. 10).

Notes

Doubly Linked List

The add_node() Function

```
1 int add_node(struct list_node **node, int number)
2 {
3     struct list_node *preceding = NULL;
4     while(*node != NULL && (*node)->data < number) {
5         preceding = *node;
6         node = &(*node)->next;
7     }
8     return create_and_add_node(node, preceding, number);
9 }
```


Notes

Doubly Linked List

The add_node() Function

The `add_node()` function, comparing to its equivalent for the singly linked list, has a local pointer called `preceding`, that initially has the `NULL` value (line no. 3). However, in each iteration of the `while` loop, the address of the node that contains the `next` member, whose address is stored in the `node` parameter in the line no. 6, is assigned to that pointer (line no. 5). The `preceding` pointer is a helper pointer and it is passed as a second argument to the `create_and_add_node()` function (line no. 8).
Just like in the case of the previous program we are going to analyse the work of these functions for the four most interesting cases of adding a new node to the list. This time however we are mainly going to focus on the differences between these functions and their equivalents for the singly linked list.

Notes

Doubly Linked List

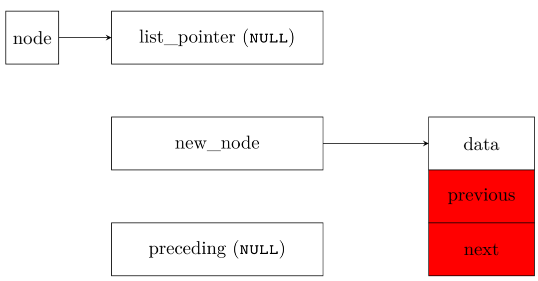
Adding The First Node

In the case of adding the first node to the list, the `while` loop in the `add_node()` function stops at once, and the `node` parameter points to the empty list pointer (the `list_pointer` variable). The `preceding` pointer is also empty. The `create_and_add_node()` function is invoked, that creates a new node, stores a number passed by the `number` parameter in it, and assigns to its `next` (line no.7) and `previous` (line no. 8) members the `NULL` value. Because there is no node that would precede, or succeed the new one in the list, the condition in the line no. 9 is not met and the function performs the statement in the line no. 11, assigning in the list pointer the address of the new node, which becomes the first and only node in the list. After that the function return 0 and exits.

Notes

Doubly Linked List

Adding The First Node

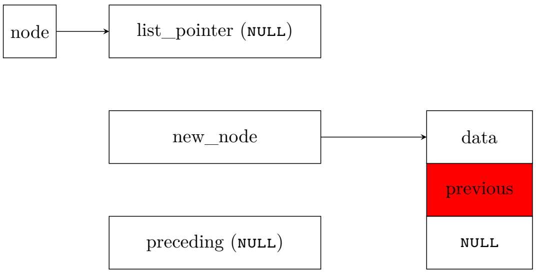


Before the line no. 7 of the `create_and_add_node()` is performed

Notes

Doubly Linked List

Adding The First Node

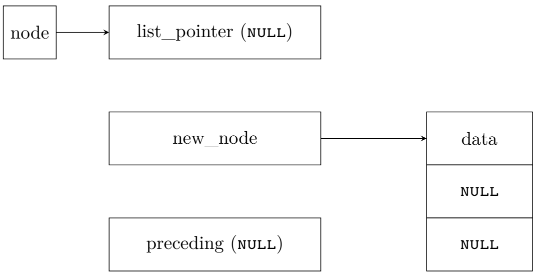


Before the line no. 8 of the `create_and_add_node()` is performed

Notes

Doubly Linked List

Adding The First Node

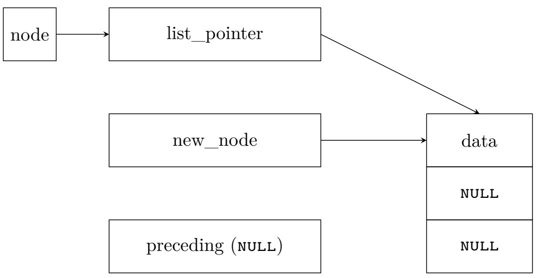


Before the line no. 11 of the `create_and_add_node()` is performed

Notes

Doubly Linked List

Adding The First Node



After the line no. 11 of the `create_and_add_node()` is performed

Notes

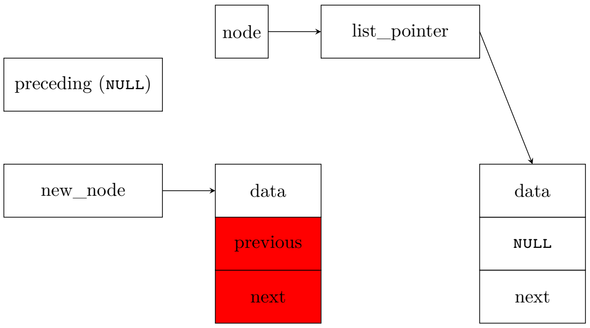
Doubly Linked List

Adding At The Front

In the case when the new node should be added at the front of the list, after the `while` loop in the `add_node()` stops, the `preceding` pointer has the `NULL` value, but the `node` pointer points to the list pointer (the `list_pointer` variable), that stores the address of the list first node. The `create_and_add_node()` function, after it creates new node and stores in it the number passed by the `number` parameter, assigns to the node's `next` filed the address of the currently first node in the list (line no. 7). Next, it assigns to the `previous` member of the new node the `NULL` from the `preceding` parameter (line no. 8). Then, it checks if there is a node that should be the successor of the new one in the list. It is the currently first node in the list, so it stores in its `previous` member the address of the new node (line no. 10). After that, the function assignes to the list pointer the address of the new node, so it becomes the first node in the list (line no. 11).

Notes

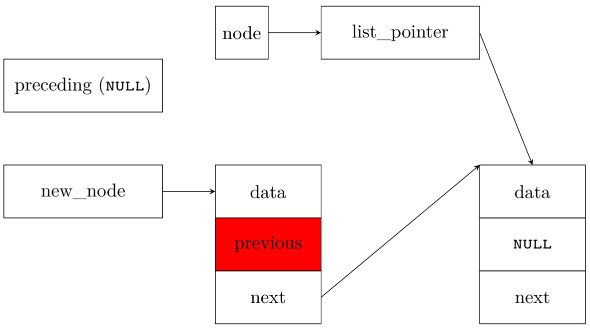
Doubly Linked List
Adding At The Front



Before the line no. 7 of the `create_and_add_node()` is performed Before the line no. 8 of the `create_and_add_node()` is performed

Notes

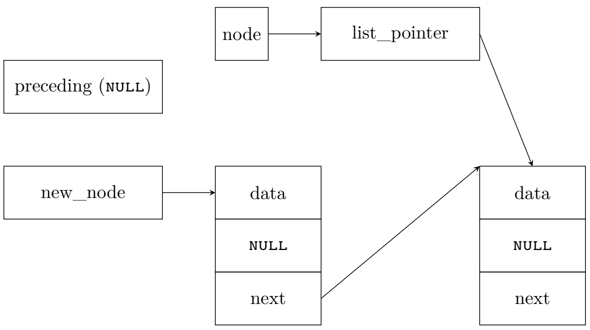
Doubly Linked List
Adding At The Front



Before the line no. 8 of the `create_and_add_node()` is performed

Notes

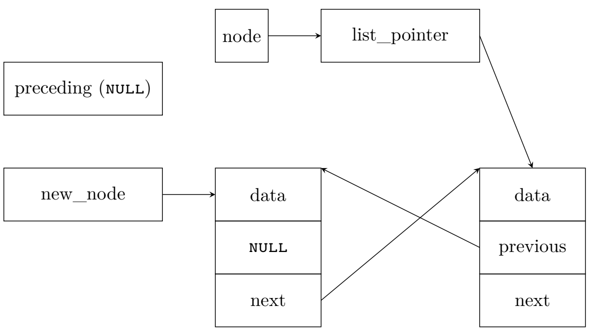
Doubly Linked List
Adding At The Front



Before the line no. 8 of the `create_and_add_node()` is performed Before the line no. 10 of the `create_and_add_node()` is performed

Notes

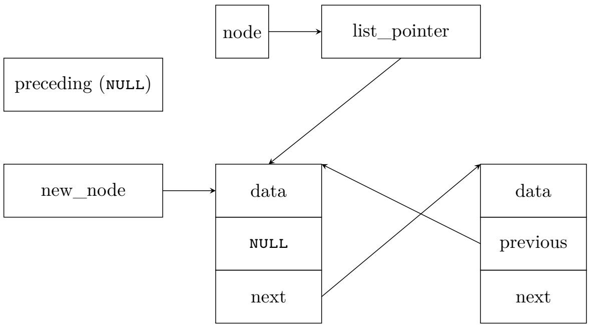
Doubly Linked List
Adding At The Front



Before the line no. 8 of the `create_and_add_node()` is performed Before the line no. 11 of the `create_and_add_node()` is performed

Notes

Doubly Linked List
Adding At The Front



Before the line no. 8 of the `create_and_add_node()` is performed After the line no. 11 of the `create_and_add_node()` is performed

Notes

Doubly Linked List
Adding Inside

If the new node should be added inside the list, then after the `while` loop in the `add_node()` function stops, the `preceding` pointer stores the address of the node *after* which the new one should be inserted, and the `node` parameter stores the address of the `next` member of this node. The `create_and_add_node()` function, after it creates the new node and assigns to it the number, stores in the `next` member of the new node the address of the node that is pointed by the `next` member, whose address is stored in the `node` parameter (line no. 7). Please notice, that the latter `next` member belongs to the node pointed by the `preceding` pointer.

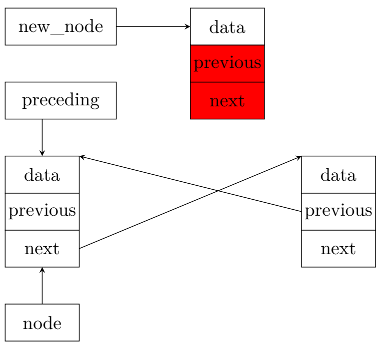
Notes

Doubly Linked List
Adding Inside

Then, the function assigns to the `previous` member of the new node the value from the `preceding` pointer, that is the address of a node that should precede the new one in the list (line no. 8). Next, it verifies that there is a node that should succeed the new one in the list (line no. 9) and stores in its `previous` member the address of the new node (line no. 10). The same address is also stored in the `next` member of the node pointed by the `preceding` pointer. Therefore, the new node is correctly added to the list and the function exits returning 0.

Notes

Doubly Linked List
Adding Inside

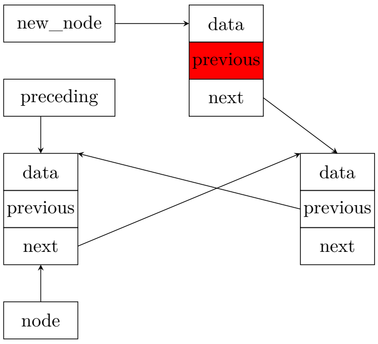


Before the line no. 7 of the `create_and_add_node()` is performed

Notes

Doubly Linked List

Adding Inside

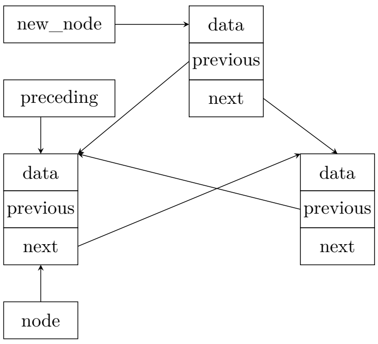


Before the line no. 8 of the `create_and_add_node()` is performed

Notes

Doubly Linked List

Adding Inside

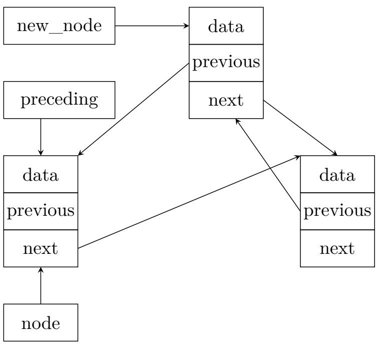


Before the line no. 10 of the `create_and_add_node()` is performed

Notes

Doubly Linked List

Adding Inside

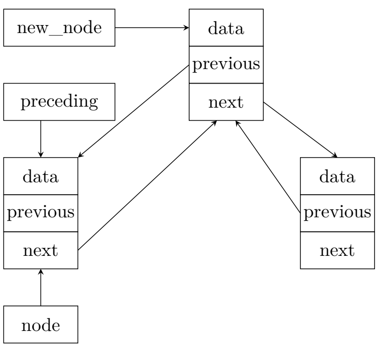


Before the line no. 11 of the `create_and_add_node()` is performed

Notes

Doubly Linked List

Adding Inside



After the line no. 11 of the `create_and_add_node()` is performed

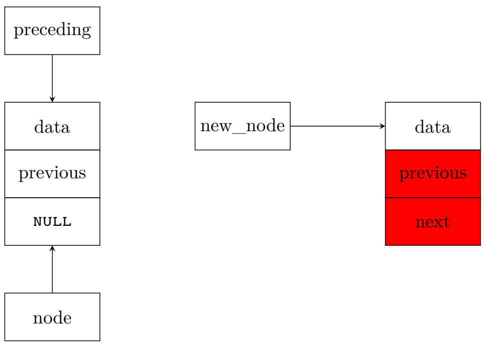
Notes

Doubly Linked List
Adding At The End

In case where the new node is added at the end of the list, after the `while` loop in the `add_node()` function stops, the `preceding` pointer stores the address of the last node in the list, and the `node` parameter stores the address of the node's `next` member. The `create_and_add_node()` function, after it creates the new node and assigns to its `data` member the number passed by the `number` parameter, stores in the `next` field of the new node the `NULL` value, because it is the value of the node's `next` member, whose address is stored in the `node` parameter (line no. 7). Next, in the `previous` member of the new node, the function stores the address of the node pointed by the `preceding` pointer (line no. 8). The condition in the line no. 9 is not met, because the new node doesn't have successor in the list. It is the last node in the list. The function performs the statement in the line no. 11, storing the address of the new node in the `next` member of the node pointed by the `preceding` pointer.

Notes

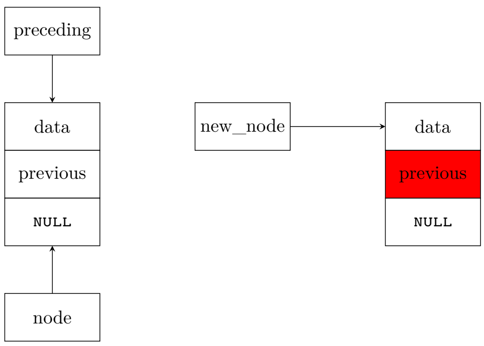
Doubly Linked List
Adding At The End



Before the line no. 7 of the `create_and_add_node()` is performed

Notes

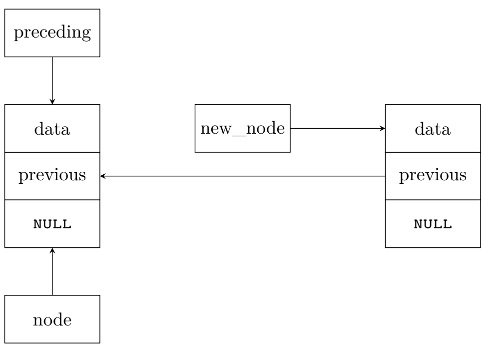
Doubly Linked List
Adding At The End



Before the line no. 8 of the `create_and_add_node()` is performed

Notes

Doubly Linked List
Adding At The End

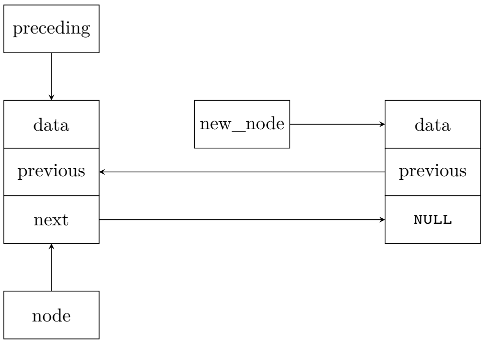


Before the line no. 11 of the `create_and_add_node()` is performed

Notes

Doubly Linked List

Adding At The End



After the line no. 11 of the `create_and_add_node()` is performed

Notes

Doubly Linked List

The `delete_node()` Function

```c
1  void delete_node(struct list_node **node, int number)
2  {
3      while(*node && (*node)->data != number)
4          node = &(*node)->next;
5      if(*node) {
6          struct list_node *temporary = (*node)->next;
7          if((*node)->next)
8              (*node)->next->previous = (*node)->previous;
9          free(*node);
10         *node = temporary;
11     }
12 }
```


Notes

Doubly Linked List

The `delete_node()` Function

The version of the `delete_node()` function for the doubly linked list has to take into account that the nodes, that potentially are neighbours of the node that should be deleted, have the `previous` member. Thus it has an additional conditional statement (lines no 7–8), compared to its equivalent for the singly linked list, that checks if there is a successor of the deleted node. If it is so, the function stores in the `previous` member of this successor the address of the node that precedes the deleted one in the list. Thanks to the additional operations, the latter node is correctly excluded from the list and can be safely removed.
Let's analyse how the function works for the same cases that has been considered in the case of its equivalent for the singly linked list.

Notes

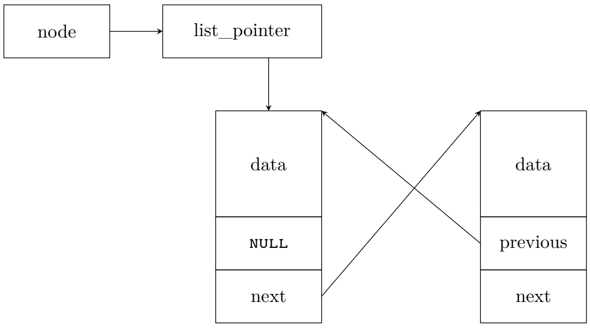
Doubly Linked List

Deleting The First Node

In the case where the first node in the list should be removed, after the `while` loop stops, the `node` parameter points to the list pointer (the `list_pointer` variable), which stores the address of the first node in the list. The function stores in the `temporary` variable the address of the second node in the list (or the `NULL` value, if it doesn't exist), which it takes from the `next` member of the first node (line no. 6). Then it checks, if that node actually exists (line no. 7), and if it is so, it assigns to its `previous` member the address that is stored in the member with the same name, but belonging to the first node. In this case it is the `NULL` value. In the line no. 9 the function releases the memory allocated for the first node, and in the line no. 10 it assigns to the list pointer, which is pointer by the `node` parameter, the address of the node that was the second one and now is the first one in the list. Please notice, that the `delete_node()` function also handles correctly the case where the first node is at the same time the only node in the list.

Notes

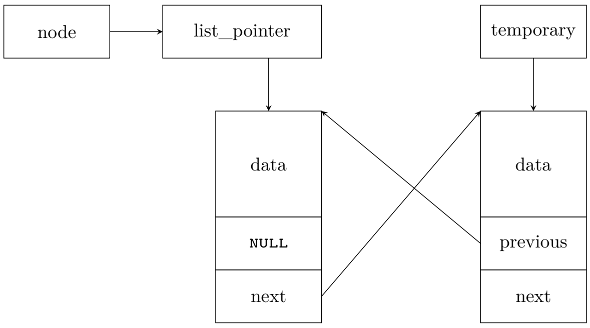
Doubly Linked List
Deleting The First Node



Before the line no. 6 of the `delete_node()` function is performed

Notes

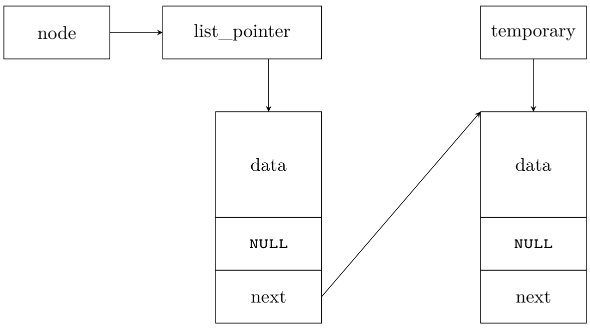
Doubly Linked List
Deleting The First Node



Before the line no. 8 of the `delete_node()` function is performed

Notes

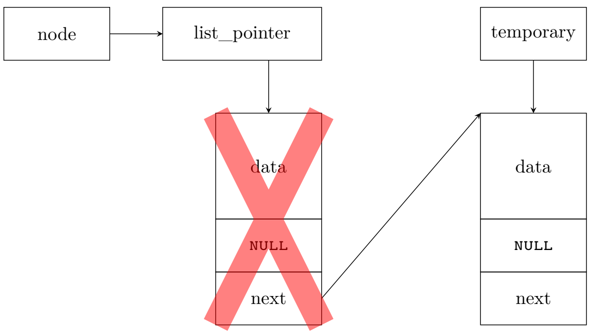
Doubly Linked List
Deleting The First Node



Before the line no. 9 of the `delete_node()` function is performed

Notes

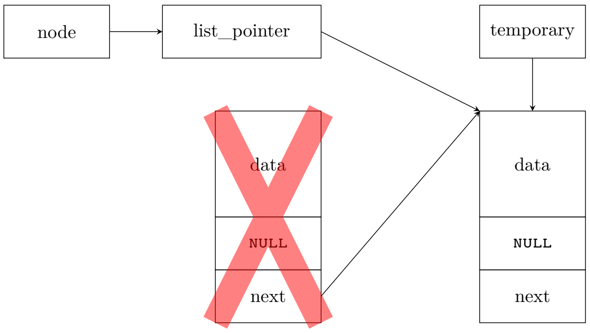
Doubly Linked List
Deleting The First Node



Before the line no. 10 of the `delete_node()` function is performed

Notes

Doubly Linked List
Deleting The First Node



After the line no. 10 of the `delete_node()` function is performed

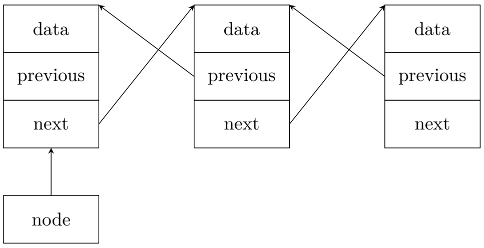
Notes

Doubly Linked List
Deleting an Inner Node

Removing a node that is located between two other nodes in the list is the most complicated case. In such a situation, after the **while** loop stops, the **node** parameter points to the **next** member that stores the address of the node to be deleted. The **delete_node()** function assigns to the **temporary** pointer the address of the node that succeeds in the list the one that should be deleted (line no. 6). It takes the address from the **next** field of the latter. Then, after checking if successor exists (line no. 7) the function assigns to its **previous** member the address that is stored in the member of the same name, but belonging to the node that should be deleted (line no. 8). After that the function releases the memory allocated for the latter node (line no. 9) and in the **next** member of its former predecessor stores the address of its former successor (line no. 10). It takes this address from the **temporary** variable. This ends this entire operation.

Notes

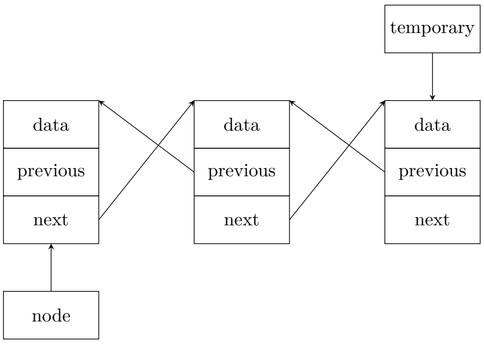
Doubly Linked List
Deleting an Inner Node



Before the line no. 6 of the `delete_node()` function is performed

Notes

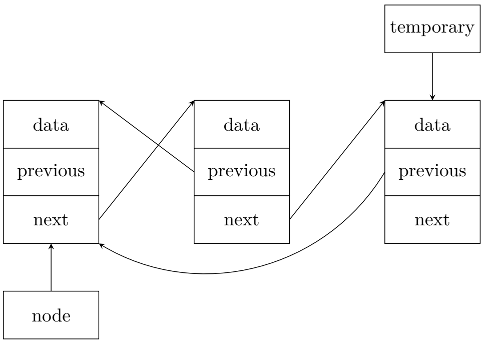
Doubly Linked List
Deleting an Inner Node



Before the line no. 8 of the `delete_node()` function is performed

Notes

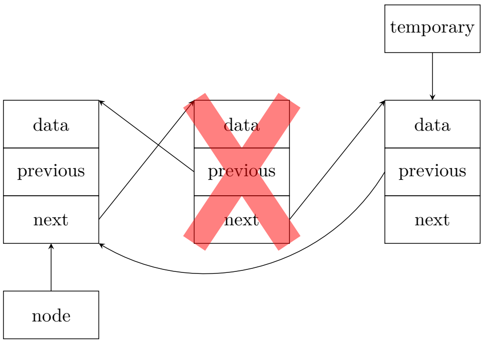
Doubly Linked List
Deleting an Inner Node



Before the line no. 9 of the `delete_node()` function is performed

Notes

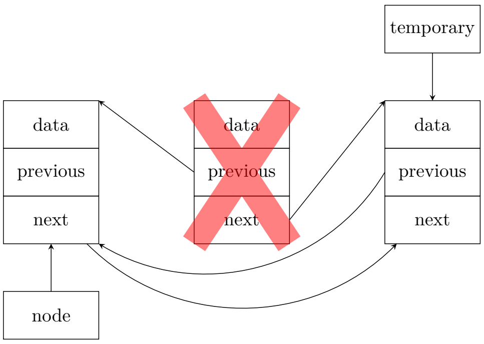
Doubly Linked List
Deleting an Inner Node



Before the line no. 10 of the `delete_node()` function is performed

Notes

Doubly Linked List
Deleting an Inner Node



After the line no. 10 of the `delete_node()` function is performed

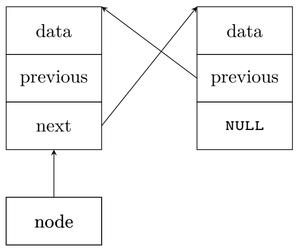
Notes

Doubly Linked List
Deleting The Last Node

The operation of removing the last node is performed in the same way as in the case of the singly linked list, so its description is skipped here, but the next slide illustrated how it is carried out.

Notes

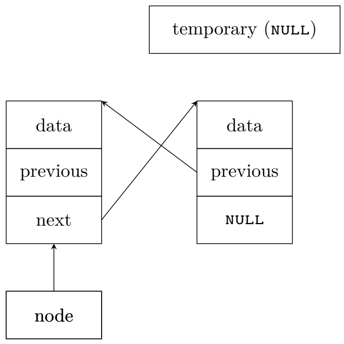
Doubly Linked List
Deleting The Last Node



Before the line no. 6 of the `delete_node()` function is performed

Notes

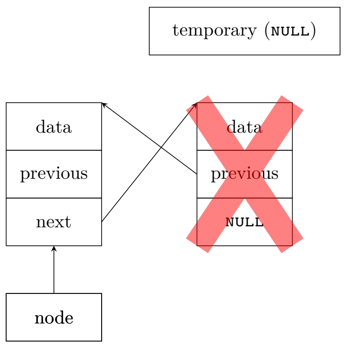
Doubly Linked List
Deleting The Last Node



Before the line no. 9 of the `delete_node()` function is performed

Notes

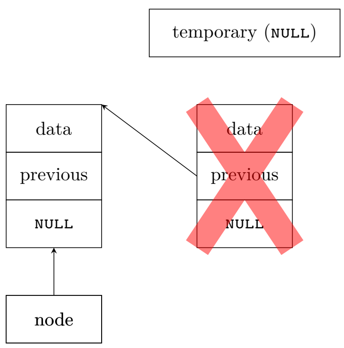
Doubly Linked List
Deleting The Last Node



Before the line no. 10 of the `delete_node()` function is performed

Notes

Doubly Linked List
Deleting The Last Node



After the line no. 10 of the `delete_node()` function is performed

Notes

Doubly Linked List

The `print_list()` Function

```
void print_list(struct list_node *node)
{
    while(node) {
        printf("%d ", node->data);
        node = node->next;
    }
    puts("");
}
```


Notes

Doubly Linked List

The `print_list()` Function

The `print_list()` function is implemented in the same way as in the program that uses the singly linked list, therefore it is not described here.

Notes

Doubly Linked List

The `print_backwards()` Function

```
1  void print_backwards(struct list_node *node)
2  {
3      while(node && node->next)
4          node = node->next;
5      while (node) {
6          printf("%d ", node->data);
7          node = node->previous;
8      }
9      puts("");
10 }
```


Notes

Doubly Linked List

Funkcja `print_backwards()`

The doubly linked list has a structure that allows the program to traverse it in two directions: from the first to the last node and from the last one to the first one. The `print_backwards()` function uses this feature. The first `while` loop (lines no. 3–4) in its body is performed until the `node` parameter points to the last node in the list (it is that node, whose `next` member has the value of `NULL`). Then, the second `while` loop (nodes no. 5–8) traverses the list until the `node` parameter becomes `NULL`. In this loop the number stored in the node currently pointed by the `node` parameter is displayed (node no. 6) and the address in the `node` parameter is replaced by the address that is stored in the `previous` member of the node that is currently pointed by this parameter (line no. 7). In other words, the pointer is "moved" to the predecessor of this node. Thanks to that, the `print_backwards()` function displays numbers stored in the list in the reversed order.

Notes

Doubly Linked List

The `remove_list()` Function

```c
void remove_list(struct list_node **node)
{
    while(*node) {
        struct list_node *temporary = (*node)->next;
        free(*node);
        *node = temporary;
    }
}
```


Notes

Doubly Linked List

The `remove_list()` Function

The `remove_list()` function is implemented in the same way as in the program that uses the singly linked list, therefore it is not described here.

Notes

Doubly Linked List

The `main()`, part 1

```c
int main(void)
{
    for(int i=1; i<5; i++)
        if(add_node(&list_pointer,i)==-1)
            fprintf(stderr,"Error adding a node with the %d number to the list!\n",i);
    for(int i=6; i<10; i++)
        if(add_node(&list_pointer,i)==-1)
            fprintf(stderr,"Error adding a node with the %d number to the list!\n",i);
    print_list(list_pointer);
    print_backwards(list_pointer);
```


Notes

Doubly Linked List

The `main()`, part 2

```c
    if(add_node(&list_pointer,0)==-1)
        fprintf(stderr,"Error adding a node with the %d number to the list!\n",0);
    print_list(list_pointer);
    print_backwards(list_pointer);
    if(add_node(&list_pointer,5)==-1)
        fprintf(stderr,"Error adding a node with the %d number to the list!\n",5);
    print_list(list_pointer);
    print_backwards(list_pointer);
    if(add_node(&list_pointer,7)==-1)
        fprintf(stderr,"Error adding a node with the %d number to the list!\n",7);
    print_list(list_pointer);
    print_backwards(list_pointer);
```


Notes

Doubly Linked List

The `main()`, part 3

```
    if(add_node(&list_pointer,10)==-1)
        fprintf(stderr,"Error adding a node with the %d
         ↪  number to the list!\n",10);
    print_list(list_pointer);
    print_backwards(list_pointer);
    puts("");
    delete_node(&list_pointer,0);
    print_list(list_pointer);
    print_backwards(list_pointer);
    delete_node(&list_pointer,1);
    print_list(list_pointer);
    print_backwards(list_pointer);
    delete_node(&list_pointer,1);
    print_list(list_pointer);
    print_backwards(list_pointer);
```


Notes

Doubly Linked List

The `main()`, part 4

```
    delete_node(&list_pointer,4);
    print_list(list_pointer);
    print_backwards(list_pointer);
    delete_node(&list_pointer,7);
    print_list(list_pointer);
    print_backwards(list_pointer);
    delete_node(&list_pointer,10);
    print_list(list_pointer);
    print_backwards(list_pointer);
    remove_list(&list_pointer);
    return 0;
}
```


Notes

Doubly Linked List

Funkcja `main()`

The only difference between the `main()` function and its equivalent in the program that uses the singly linked list is that in the former one the `print_backwards()` is invoked after each call to the `print_list()` function. The former function prints on the screen numbers stored in the list, in the reversed order, thus verifying if the list is coherent.

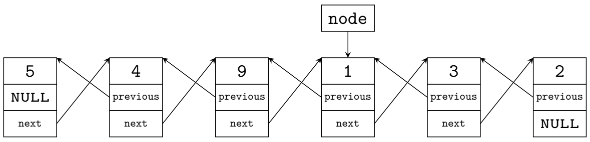
Notes

Summary

In the presented functions, like in `add_node()` or `delete_node()`, complex expressions created with the use of the pointers are applied. The next slide shows equally complicated expressions of this kind. These 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.

Notes

Summary

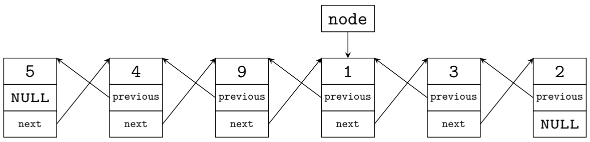


Expression no. 1

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

Notes

Summary



Expression no. 1

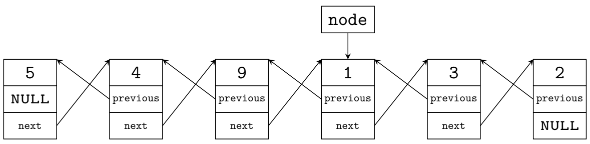
`node->next->next->data`

Answer no. 1

2

Notes

Summary

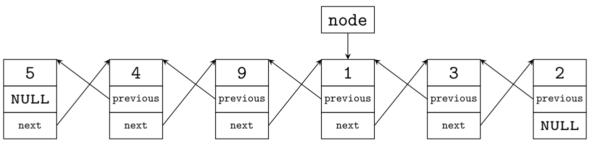


Expression no. 2

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

Notes

Summary



Expression no. 2

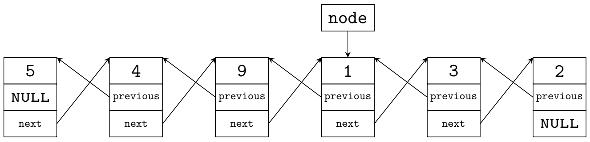
`node->previous->previous->previous->data`

Answer no. 2

5

Notes

Summary

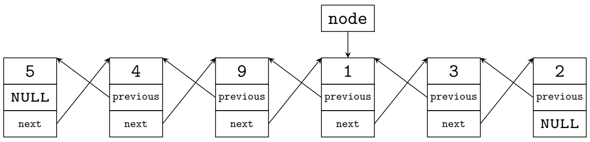


Expression no. 3

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


Notes

Summary



Expression no. 3

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

Answer no. 3

4

Notes

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

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 ☺.

Notes

Questions

?

Notes

Thank You For Your Attention!

Notes

Notes

Notes

Notes