Linked lists

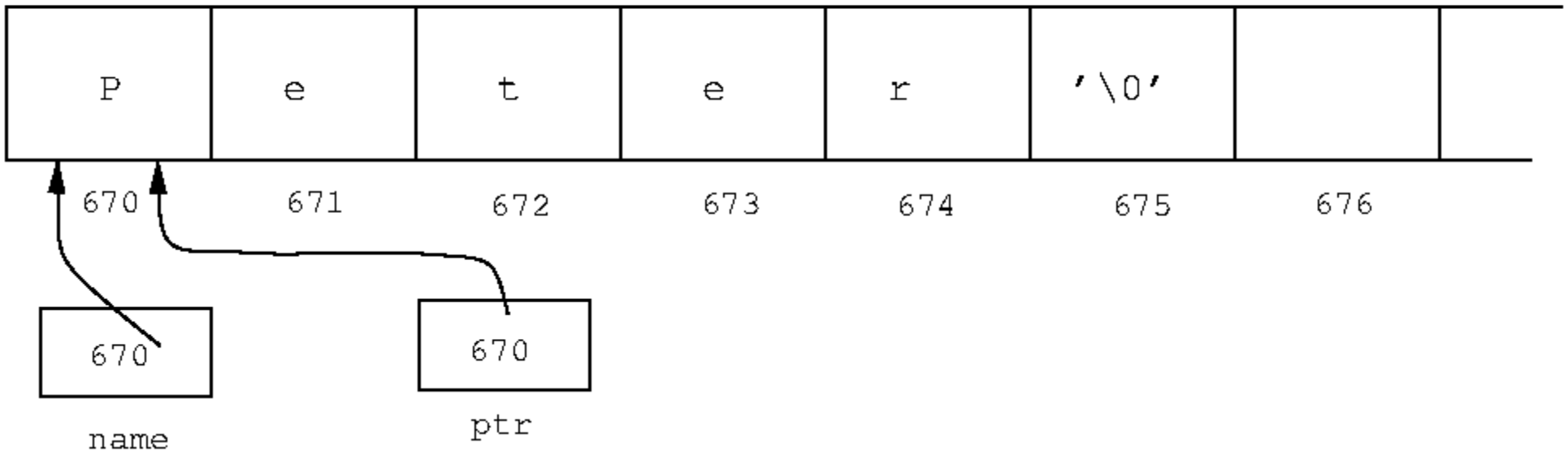
Lecture 6

Strings (recap)

- A string (character array) refers to a contiguous memory space reserved for multiple characters
- Below is a memory reserved for 10 characters in the character table "name"
- The size of a character is typically 1 byte, or byte, the basic unit of memory (8 bits)
- String ends with NULL: '\ 0'
- Individual characters can be accessed with indexes: name[4]
- The ptr pointer points to one of the characters in the table,
- It is really a matter of style and of coding conventions, since name[i] is defined to be the same as *(prt+i) where i an integer index

Strings (recap)

```
char name[10] = "Peter";
char *ptr = name
```



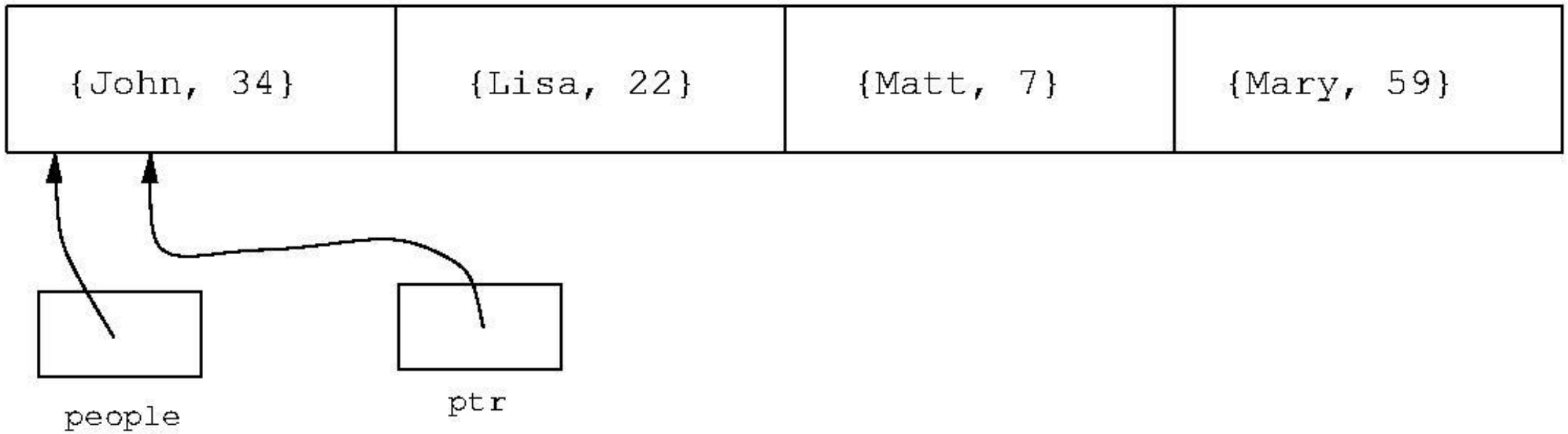
Array of numbers and structs

- An array of numbers resembles a character table
- The size of a data is always the same, that is, the amount of memory needed
- The the size depends on the data type. For instance 10 x int takes typically 40 bytes and 10 x double takes 80 bytes etc.
- No NULL character, the size information need to be stored in a variable
- A table can have many dimensions, i.e. 1, 2, 3 or more, and corresponding indices
- A table of records table is implemented in the same way, the item size is the record size

```
typedef struct person {
        char name[30];
        int age;} PERSON;
```

Array of structs

```
PERSON people[4];
PERSON *ptr = people
```

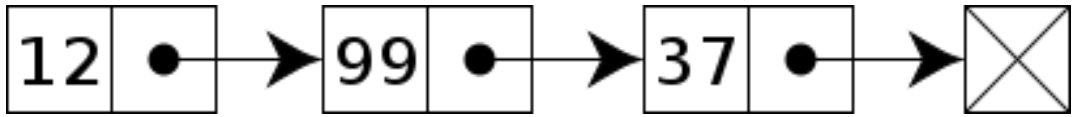


Linked lists

A **linked list** is a linear collection of data elements whose order is not given by their physical placement in memory.

Instead, each element **points** to the next.

It is a data structure consisting of a collection of nodes which together represent a sequence.



Linked lists were developed already in 1955–1956.

Linked lists

A linked list is a chain of dynamically allocated and pointer-linked *records*, called **nodes**.

All needed data structures / types have already been considered in Lectures 4 and 5.

Note that there can are often problems with a linked list, as any minor error can prevent the program from working.

Memory errors often lead to

```
Segmentation fault (core dumped),
```

which is a specific kind of error caused by accessing memory that “does not belong to you.

Be careful and do not panic. This may be difficult in the beginning.

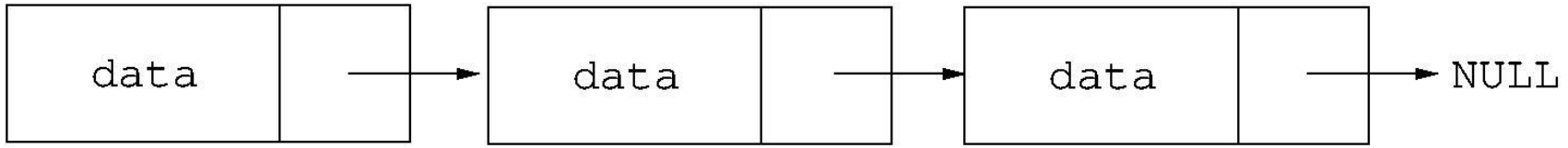
Node

A linked list consists of nodes. Node contains some data (one or more fields) and a pointer to the next node. In its simplest form, a node can be depicted in this way:

In C, we can represent a node using structures.

Linking

We can now form a **linked list** data structure just by “linking” nodes to form a list.

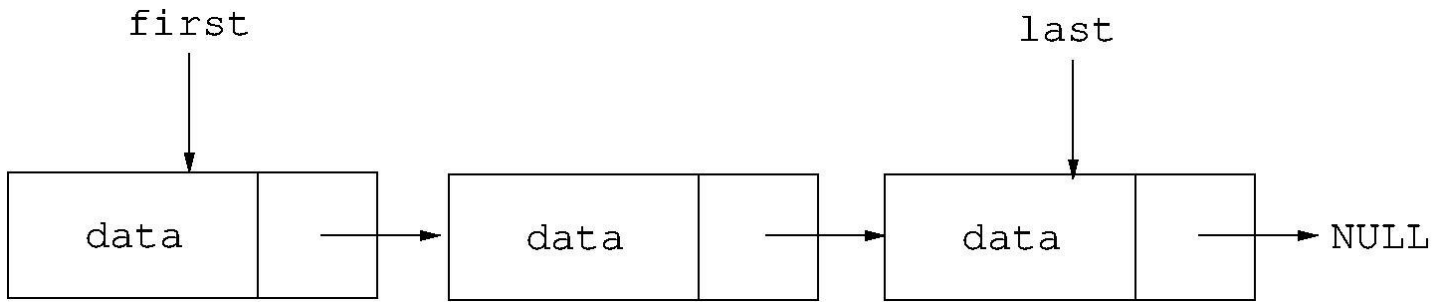


Notice that the nodes can be anywhere in the memory! Each node knows in what address the following element is.

If the element is the last one, it “points” to the NULL pointer.

First and last element

Because otherwise the list would just “float” in the memory space, we need **pointers** to the first element and the last element of the list:



When we create a new empty list, we set

```
first = last = NULL
```

Node

```
struct Node
{
    int data;
    struct Node *next;
};
typedef struct Node node;
```

Creating an empty list

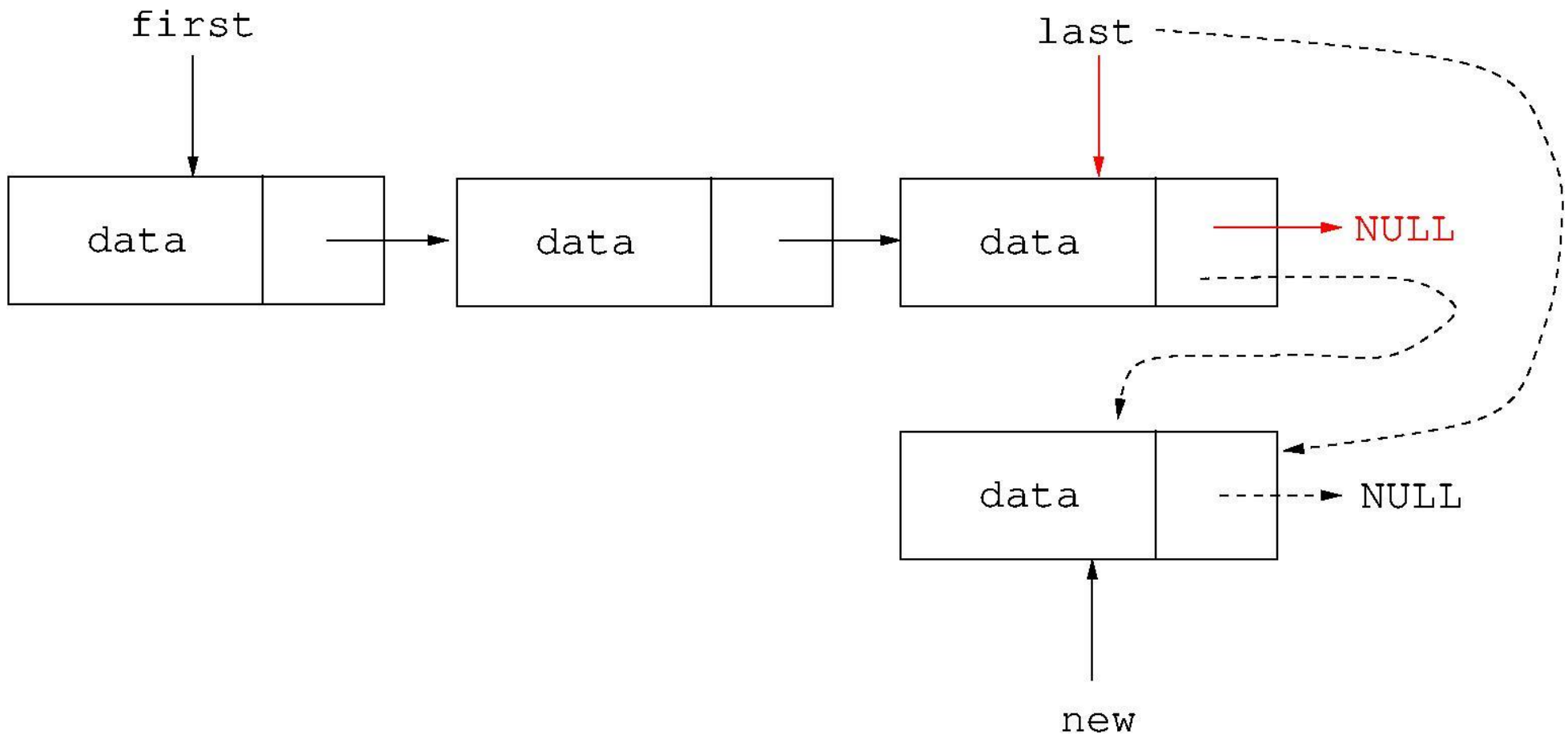
```
node *first = NULL;
node *last = NULL;
```

Inserting a new element

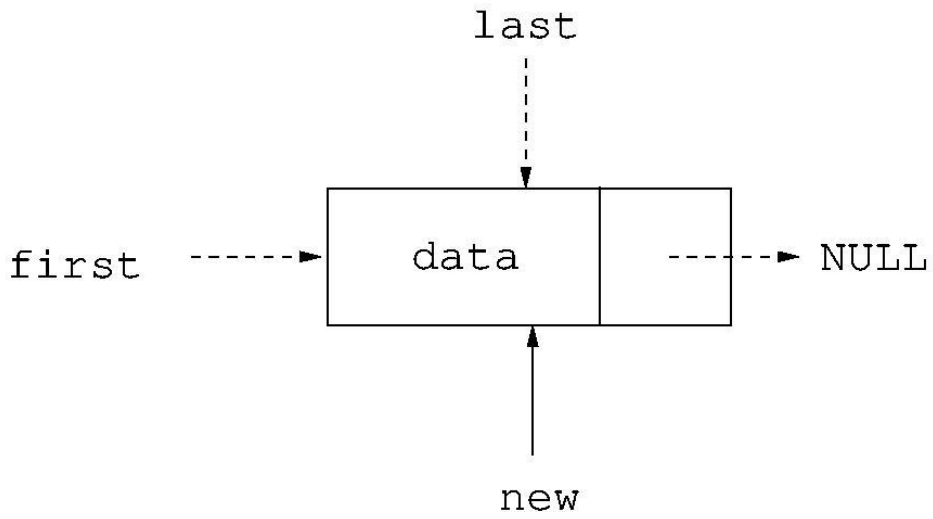
We insert a new element to the end of the list. We could also add it easily to any position just by changing the pointers. But in this way, the order of the elements is not changing.

```
node *new = (node*) malloc(sizeof(node));
new->data = x;          // set the data
new->next = NULL;
if (first == NULL) { // list was  empty
    first = new;
    last = new;
 } else
    last->next= new; // add new to the end
    last = new;
```

List was not empty

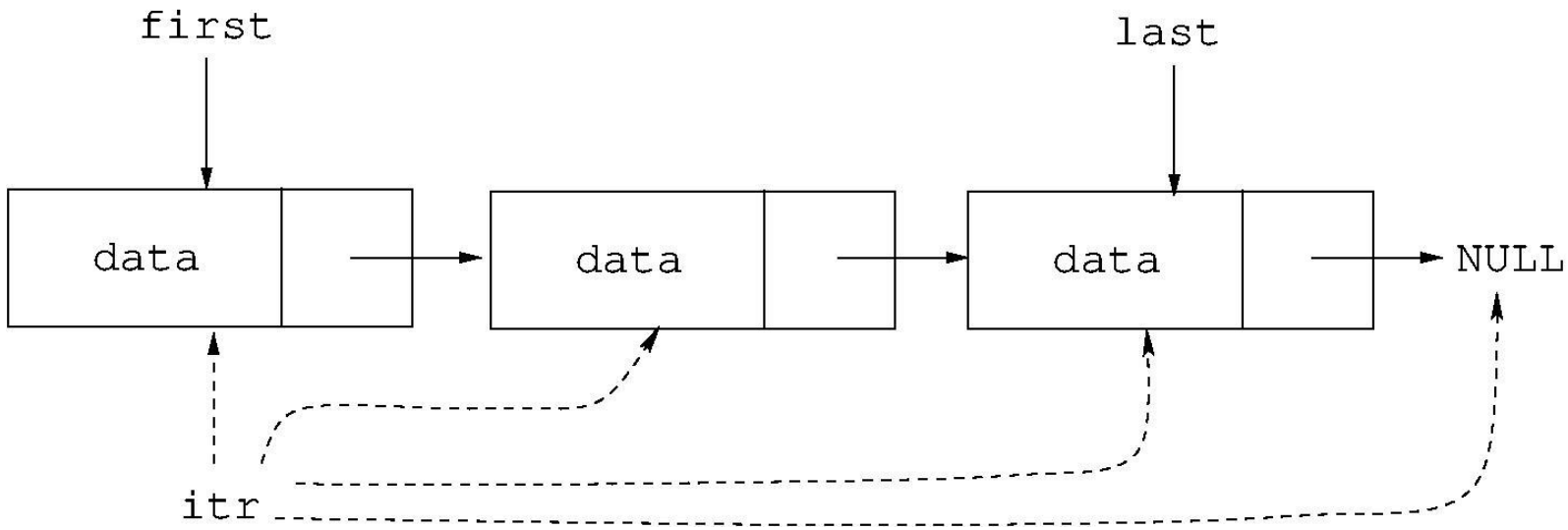


List was empty



Traversing the list

```
for (node *itr = first; itr != NULL; itr = itr->next)
        printf("%d ", itr->data);
```



File: Week6 > `Example1.c`

Freeing the memory

```
itr = first;
while (itr != NULL) {
    first = itr->next;
    free(itr);
    itr = first;
    }
last = NULL; // just in case
```

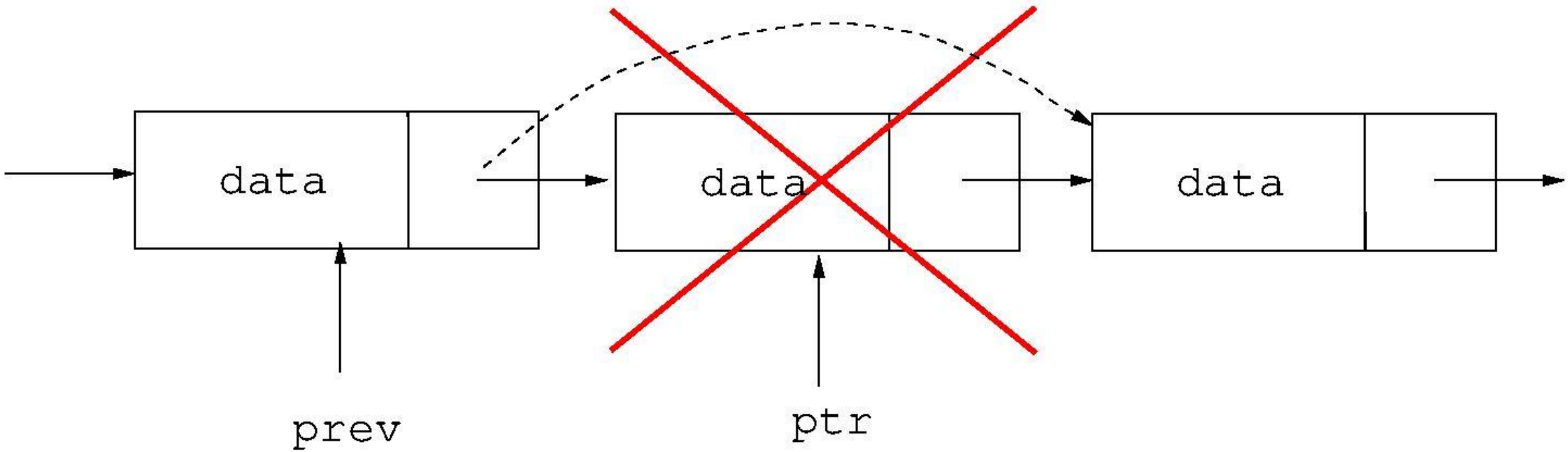
File: `Example2.c`

Performing the operations in functions

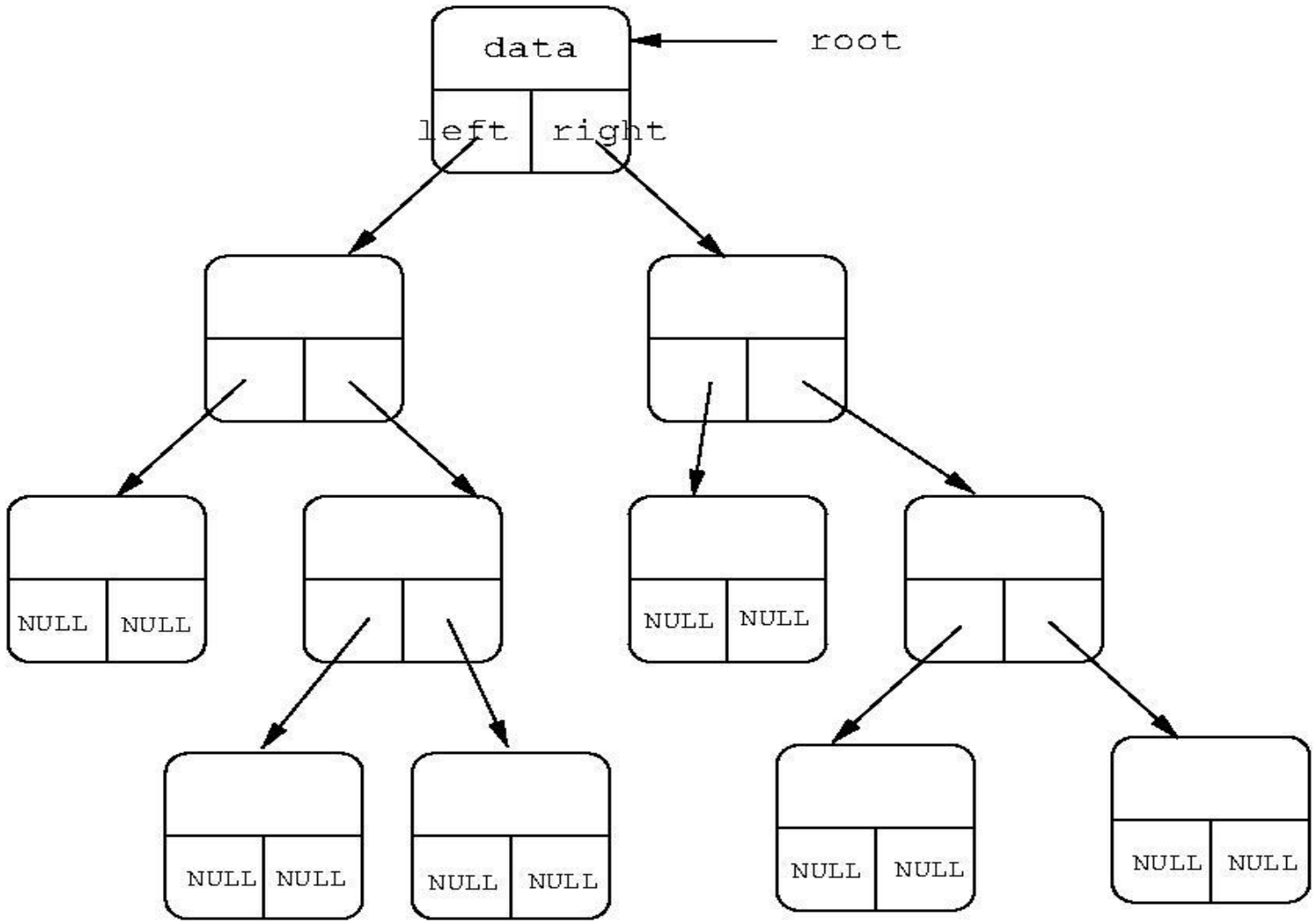
File: `Example3.c`

Removing an element (idea)

```
prev->next = ptr->next
```



You can build anything with “nodes”

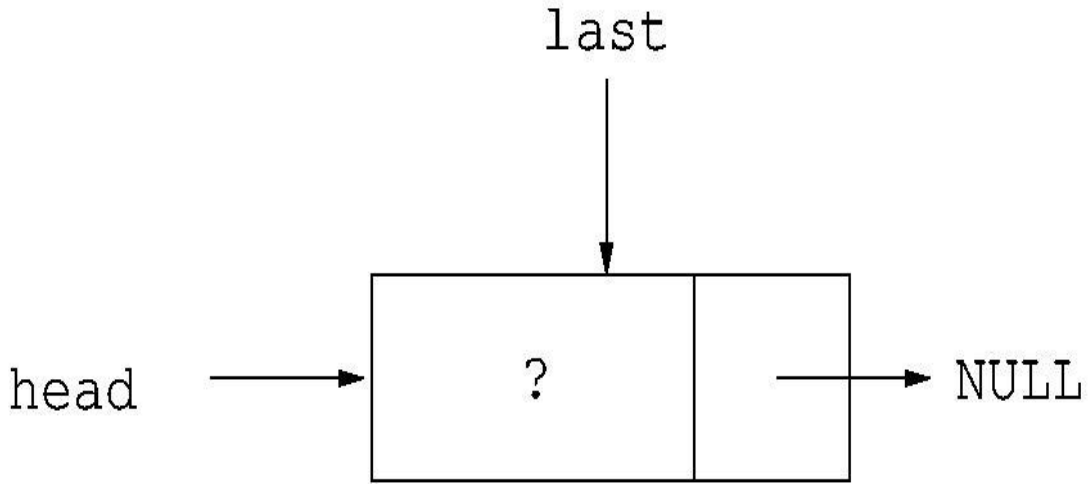


Header node

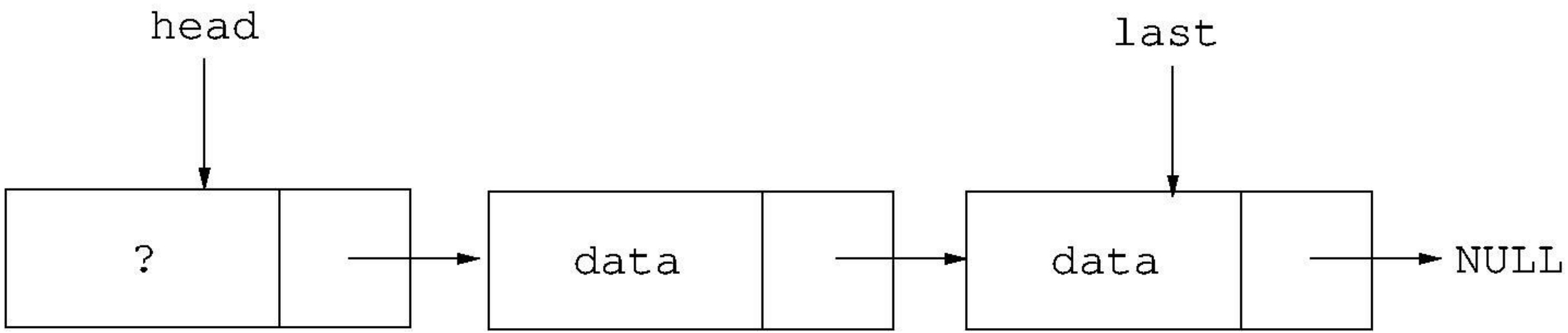
- **Header node** will simplify your life
- A header node is a special node that is found at the beginning of the list. A list that contains this type of node, is called the **header-linked list.**
- Header node `head` does not contain data (may be undefined)
- It is there just to make sure that the list is never empty
- The “actual” first element is `head->next`

Creating an empty list

```
node *head = malloc(sizeof(node));
head->next = NULL;
node *last = head;
```



Header list



File: `Example4.c`