
# Linked Lists

## Atomic Operations

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

# Linked list: Definition

**Definition of a C structure**

```
typedef struct list_s {
  int key;
  ...
  struct list_s *next;
} list_t;
```

Key and data fields

Auto-referencing pointer

```
typedef struct list_s list_t;
struct list_s {
  int key;
  ...
  list_t *next;
};
```

# Allocation of a new node

**Memory allocation**

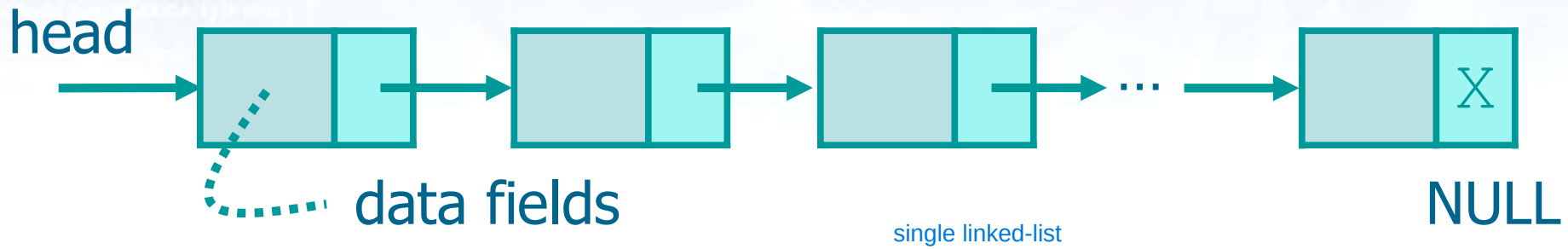
```
list_t *new_element ( ) {
  list_t *e_ptr;
  e_ptr = (list_t *) malloc (sizeof (list_t));
  if (e_ptr==NULL) {
    fprintf (stderr, "Memory allocation error.\n");
    exit (FAILURE);
  }
  return (e_ptr);
}
```

**Function call**

```
list_t *head, *new;
...
head = NULL;
...
new = new_element();
```

Initially the list is empty, thus, **head** must be initially set to NULL

# Visit




single linked-list
I cannot go back, only forward

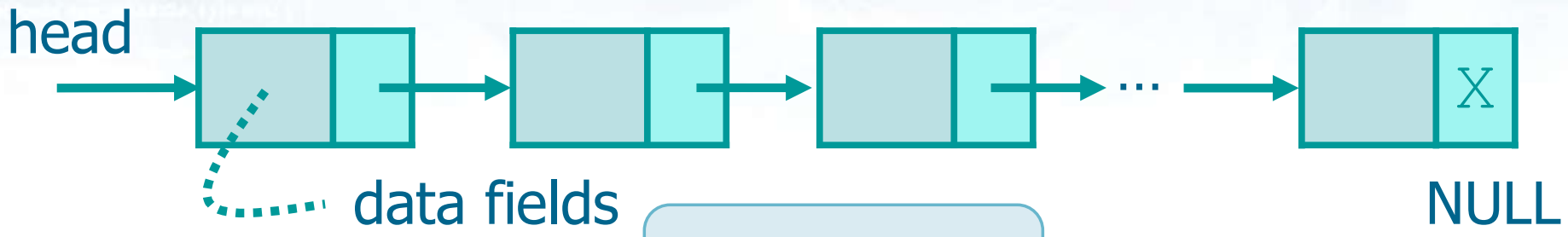
```
list_t *p;
...
p = head;
while (p != NULL) {
   ...
   p = p->next;
}
```

We check up-front, thus the algorithm works even for empty lists

Visit the element p, i.e., p->key and all data fields

Move to the next element

# Search 1



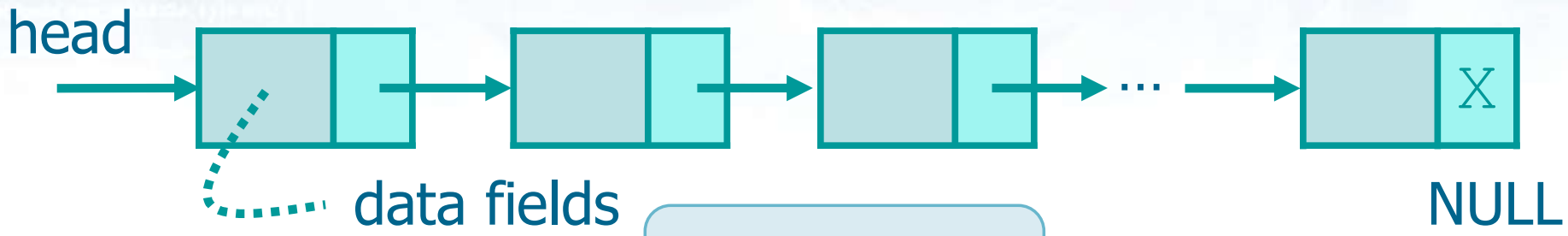

Looking for a specific **value**

```
list_t *p;
...
p = head;
while (p!=NULL) {
  if (value==p->key) {
    ...
  } else {
    p = p->next;
  }
}
```

Element found

Need to break out of the while loop once done

# Search 2




Looking for a specific **value**

```
list_t *p;
...
p = head;
while ((p!=NULL) && (p->key!=value)) {
  p = p->next;
}
if (p!=NULL) {
  ...
}
```

note that the order of the two conditions is important

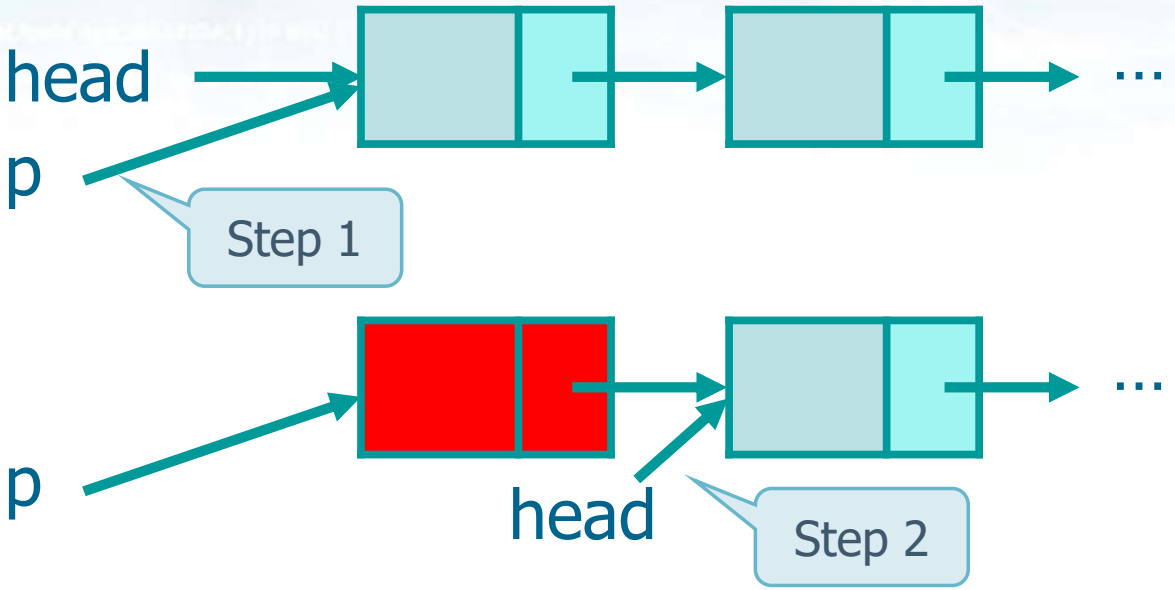
Element found

The order **matter**

`((p->key!=value) && (p!=NULL))`

**is buggy**

# Head extraction



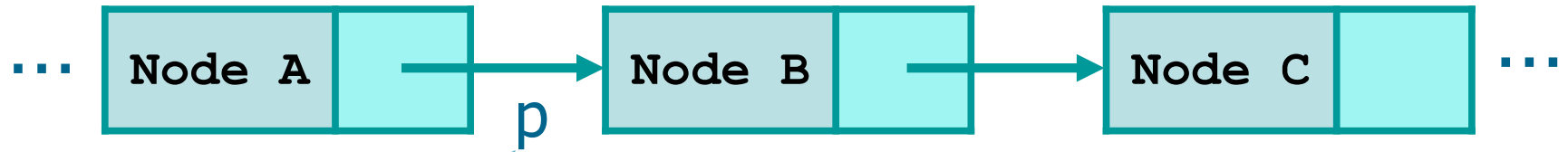

```
      if (head!=NULL) {
Step 1  p = head;
Step 2  head = head->next;
          ...
        }
```

Deal with the element p, i.e., p->key and all data fields. Finally, **free** it

# In-order extraction

- The extraction of a given element is possible only if we have access to the element placed before it
  - To extract an element we need its pointer
    - This pointer is stored in the element placed in the list before the element we want to extract
  - For now, we suppose we need to extract the successor of an element
    - We will analyze how to reach it in the next section

if we need to extract the node B we need to access to the previous pointer




With pointer p, we can extract node C not node B

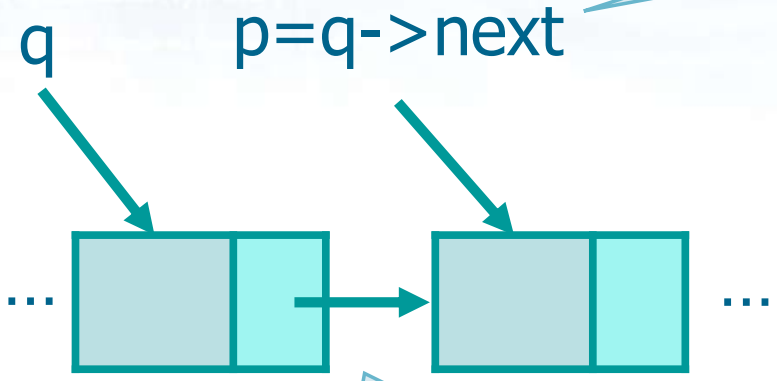
If we visit the list looking for the node to erase, we may be to late to perform the operation

# In-order extraction 1

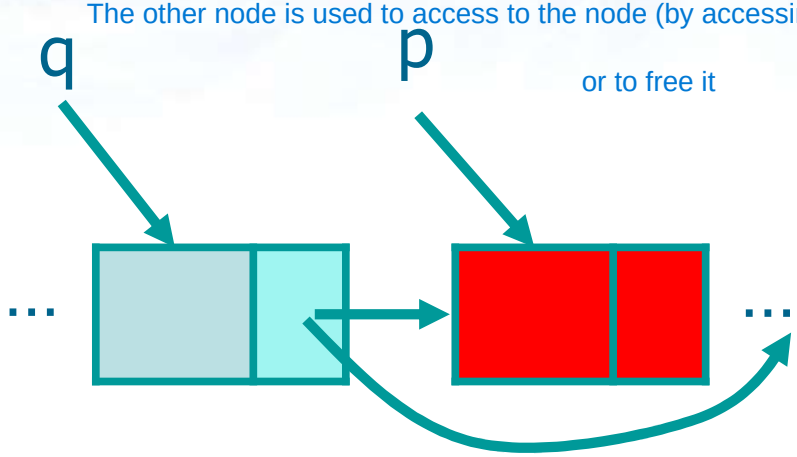
Step 1

I need two pointers because I cannot modify it when I am in the node.
Hence, one node is further checking the key comparison.
The other node is used to access to the node (by accessing to next ptr

or to free it




Ergo ... We need to traverse the list using two pointers or the pointer of the pointed node



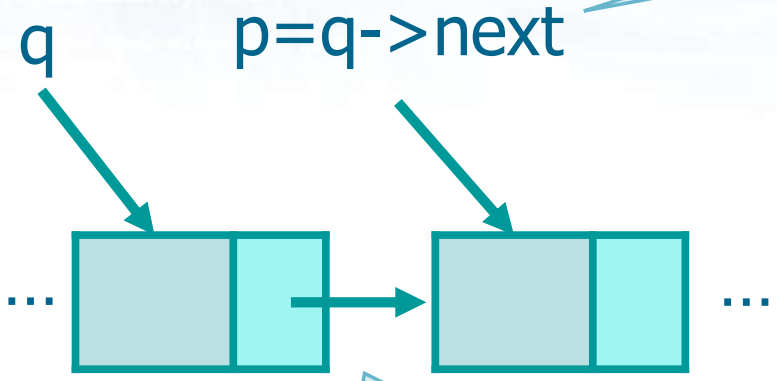

Step 2

```
Step 1  p = q->next;
Step 2  q->next = p->next;
        ...
```

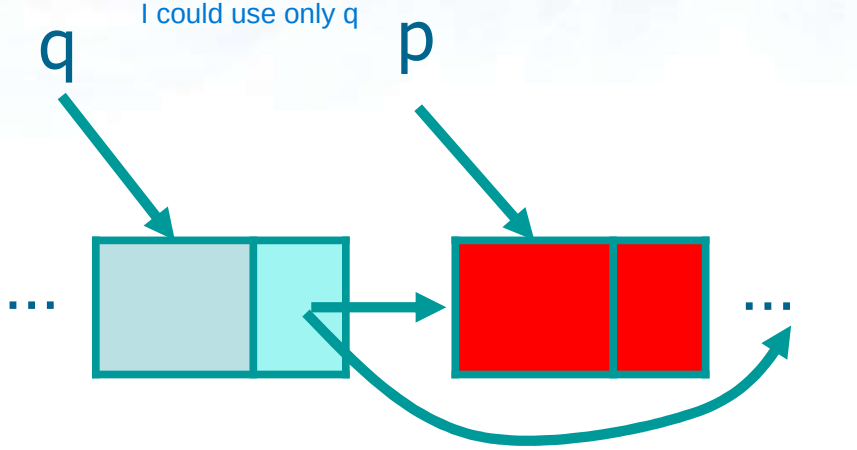
Deal with the element p, i.e., p->key and all data fields. Finally, free it

# In-order extraction 2

Step 1




In this case I use two pointers only to operate with p, but in theory I could use only q




q->next=q->next->next

Step 2

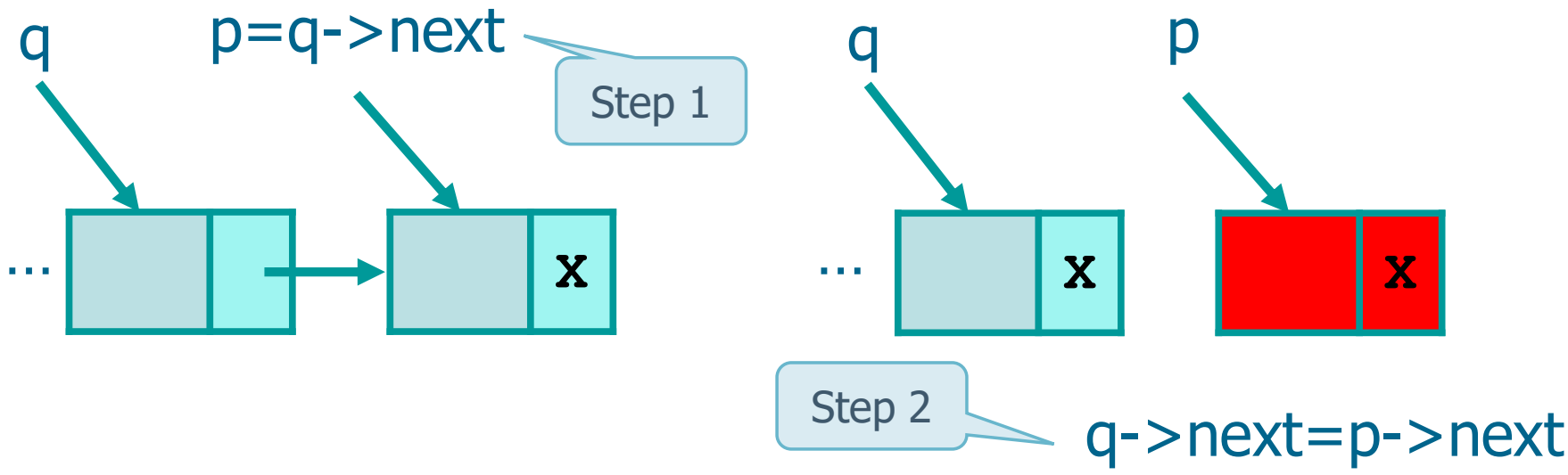
Ergo ... We need to traverse the list using two pointers or the pointer of the pointed node

```
Step 1  p = q->next;
Step 2  q->next = q->next->next;
        ...
```

p is required only to deal with the extracted element

# Tail extraction

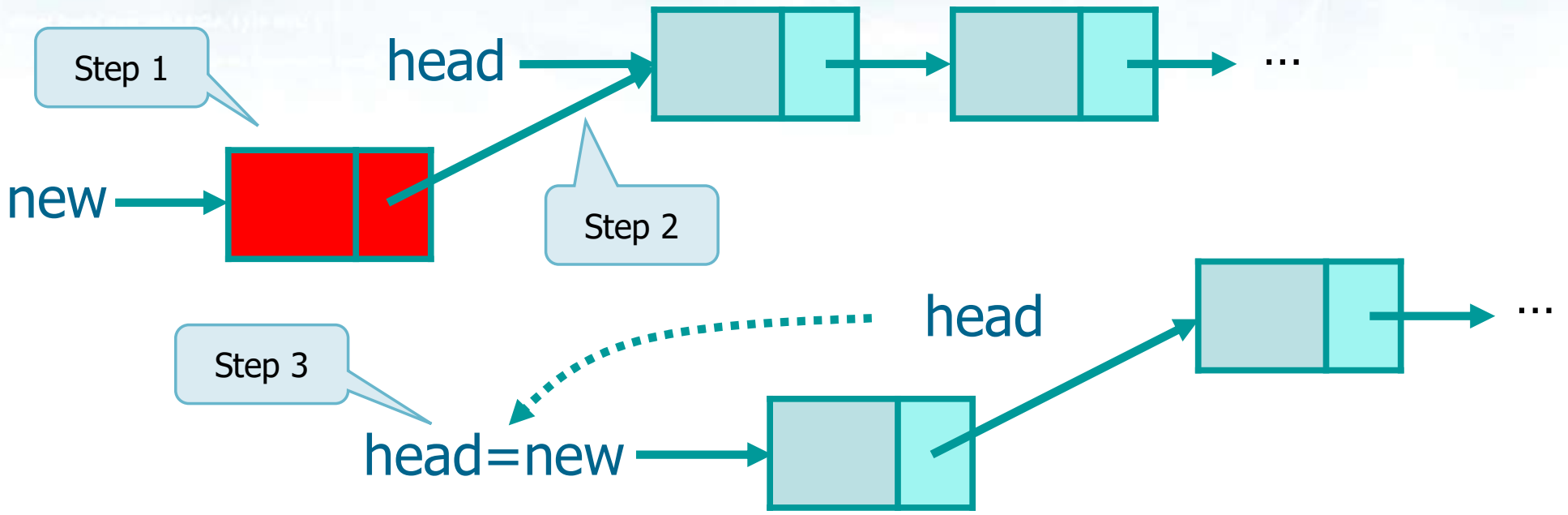
- If it is necessary extract an element from the list tail, i.e., extract the last element
  - Just use the previous code after making **p** referring to the element before the last one




```
Step 1  p = q->next;
Step 2  q->next = p->next;
        ...
```

The only difference is that p->next is NULL

# Head insertion



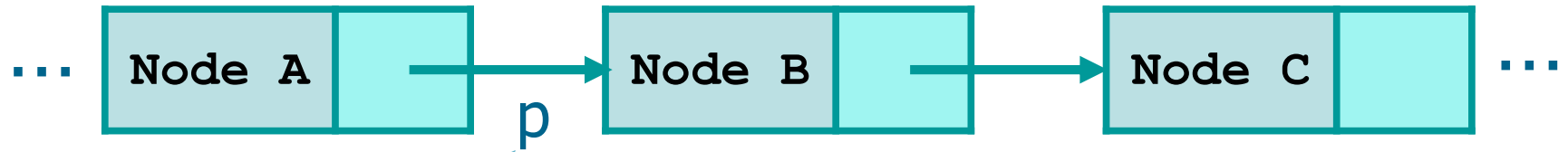

```
Step 1  new = new_element();
        ...
Step 2  new->next = head;
Step 3  head = new;
```

Set key and data fields
(possibly using strcpy, strcat, etc.)

Does it work if the lists is empty?
Yes, it does ...
If the list is empty head=NULL,
then p->next will be NULL

# In-order insertion

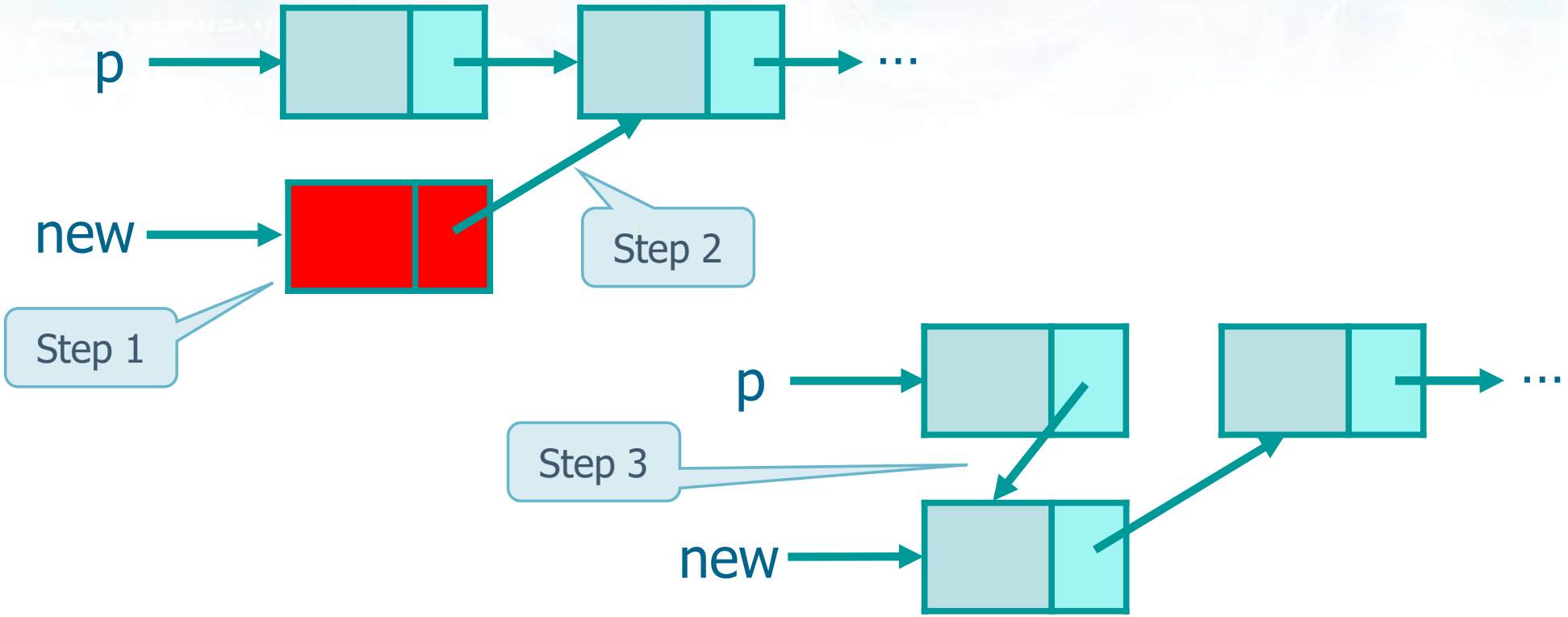
- Similarly to the extraction case, to insert a new element before a given element p, it is necessary to access the pointer field of the element coming before p
- Thus, we focus on the insertion of a new element after (not before) an existing element p
  - We will analyze how to reach it in the next section




With pointer p, we can insert a node after B not before it

If we visit the list looking for the position to insert, we may be to late to perform the operation

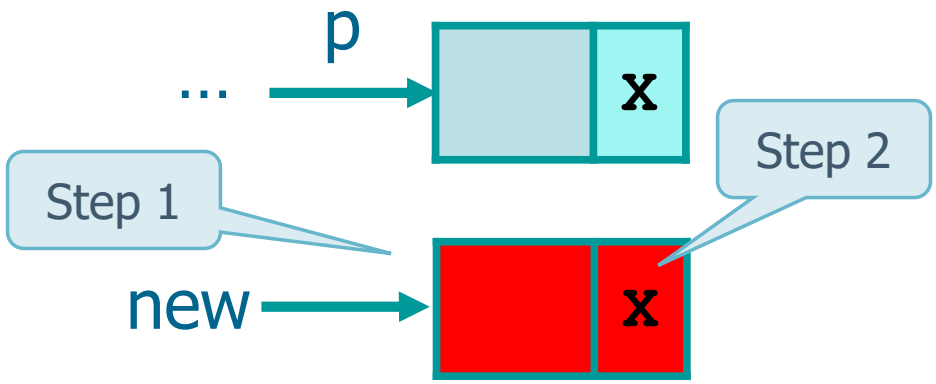
# In-order insertion

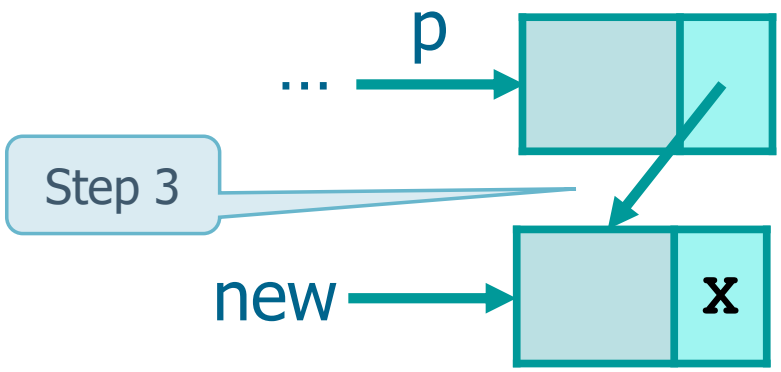



```
Step 1  new = new_element();
        ...
Step 2  new->next = p->next;
Step 3  p->next = new;
```

# Tail insertion

- If it is necessary insert an element into the list tail, i.e., as a last element
  - Just use the previous code after making **p** referring to the last element of the list

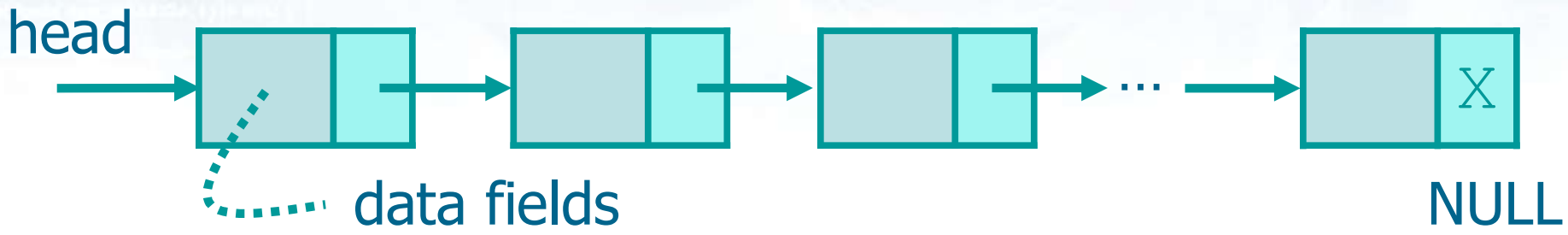



```
Step 1  new = new_element();
        ...
Step 2  new->next = p->next;
Step 3  p->next = new;
```

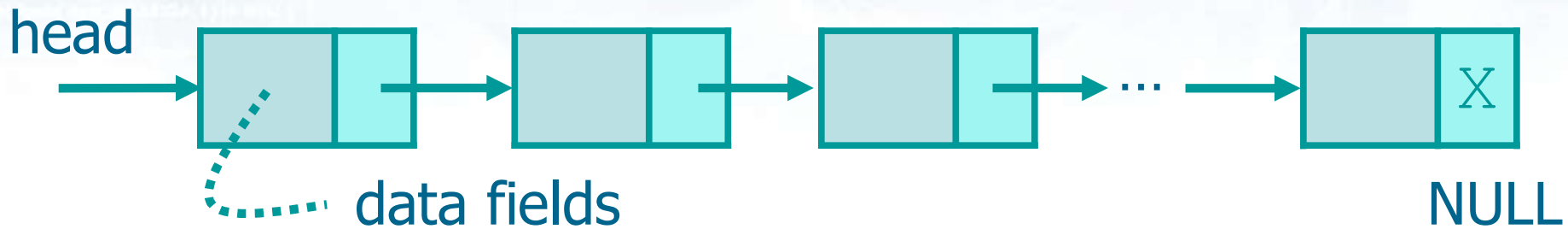
The only difference is that p->next is NULL

# Dispose a list




- Lists must be freed when they are no longer necessary
  - To free a list, we must visit it and free its elements one by one
  - Pay attention not to free an element before saving the pointer to the next element

# Dispose a list



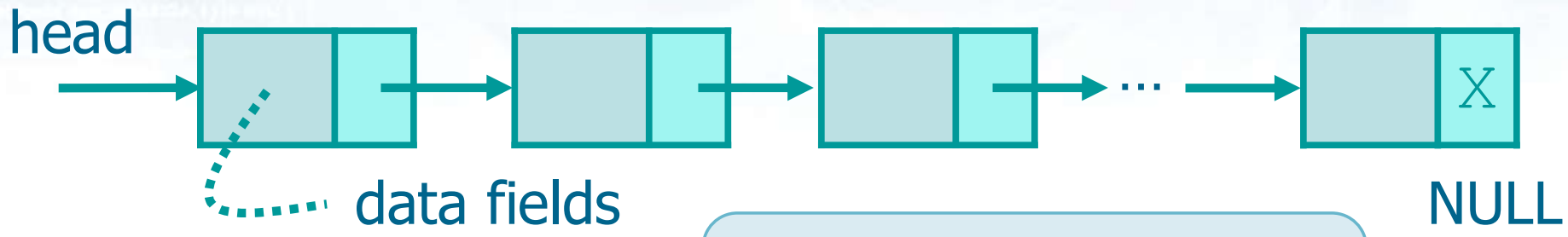

```
p = head;
while (p != NULL) {
   ...
   free (p);
   p = p->next;
}
head = NULL;
```

First, free inner fields
Then free the element itself

**Buggy code**

With **p = p->next**, we make an access to the **next** field of a freed element

# Dispose a list




We can use **head** directly to visit the list and p can be dedicated to extract elements

```
while (head != NULL) {
   p = head;
   head = head->next;
   ...
   free (p);
}
```

First, free inner fields
Then, free the element itself

# Lists with special elements

- Several operations on lists can be simplified using the so called **sentinels**
  - A sentinel (also called signal value, or dummy value, or flag value) is often used to indicate the **end** or the **beginning** of the list
  - There are at least three type of extensions using sentinels, as sentinels can be used on
    - The head of the list
    - The tail
    - On both the head and tail

never used in the course,
but helps with efficiency since we have one condition less to check in the loop

# Example

List search

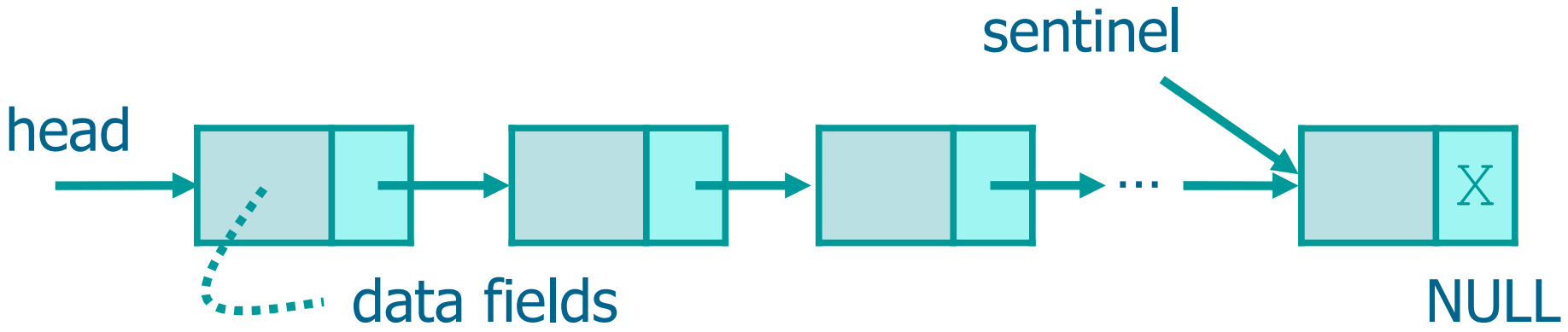
```
p = head;
while ((p!=NULL) && (p->key!=value)) {
  p = p->next;
}
if (p!=NULL) {
  ...
}
```

The **value** has been found

- Searching for an element implies checking 2 conditions
  - The pointer **p** does not have to be NULL **and** it does not have to refer to the node storing value
  - Checking the logical AND of two conditions is more expensive than checking only one condition

# Example

- We insert a sentinel element at the end of the list
  - We always have at least one element in the list, i.e., the sentinel
    - We waste a small chunk of memory
  - We can use the extra element to store the value we are looking for
  - Thus, we can simplify the search condition



# Example

```
sentinel->key = value;
p = head;
while (value!=p->key) {
  p = p->next;
}
if (p!=sentinel) {
  ...
} else {
  ...
}
```

At the very beginning, we insert the value in the sentinel

Thus, **value** is always in the list and we can simplify the condition

value found

value **not** found

- Notice that we need to maintain the sentinel node in all cases from the list initialization on
  - The code is more efficient but the logic is more complex