

```
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
```

```
#define MAXPAROLA 30
#define MAXRIGA 80
```

```
int main(int argc, char *argv[])
{
    int freq[MAXPAROLA]; /* vettore di contatori
delle frequenze delle lunghezze delle parole */
    char riga[MAXRIGA];
    int i, inizio, lunghezza;
    FILE *f;
```

```
for(i=0; i<MAXPAROLA; i++)
    freq[i]=0;
```

```
if(argc != 2)
```

```
{
    fprintf(stderr, "ERRORE, serve un parametro con il nome del file\n");
    exit(1);
}
```

```
f = fopen(argv[1], "r");
if(f==NULL)
```

```
{
    fprintf(stderr, "ERRORE, impossibile aprire il file %s\n", argv[1]);
    exit(1);
}
```

```
while( fgets( riga, MAXRIGA, f ) != NULL )
```



Linked Lists

Common Linked List

Stefano Quer

Dipartimento di Automatica e Informatica

Politecnico di Torino

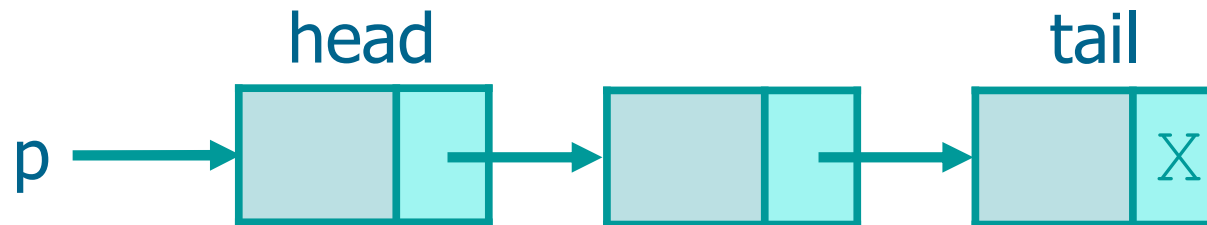
Introduction

- ❖ Basic linked list operations can be used to maintain simple linked lists with different logic
 - LIFO (Last-In First-Out)
 - FIFO (First-In First-Out)
 - Sorted list
- ❖ They can also be used to maintain more linked lists with different formats
 - Doubly-linked lists
 - List of lists

LIFO linked list

- ❖ The simplest strategy to manipulate a linked list of elements is to insert and extract elements from the same extreme

- The list has a head and a tail



- We never operate on the tail
- We always operate on the head

- The head is often called **top** in this case
- Insertion is often called **push**
- Extraction is often called **pop**

We have a cost equal to $O(n)$ to reach the tail

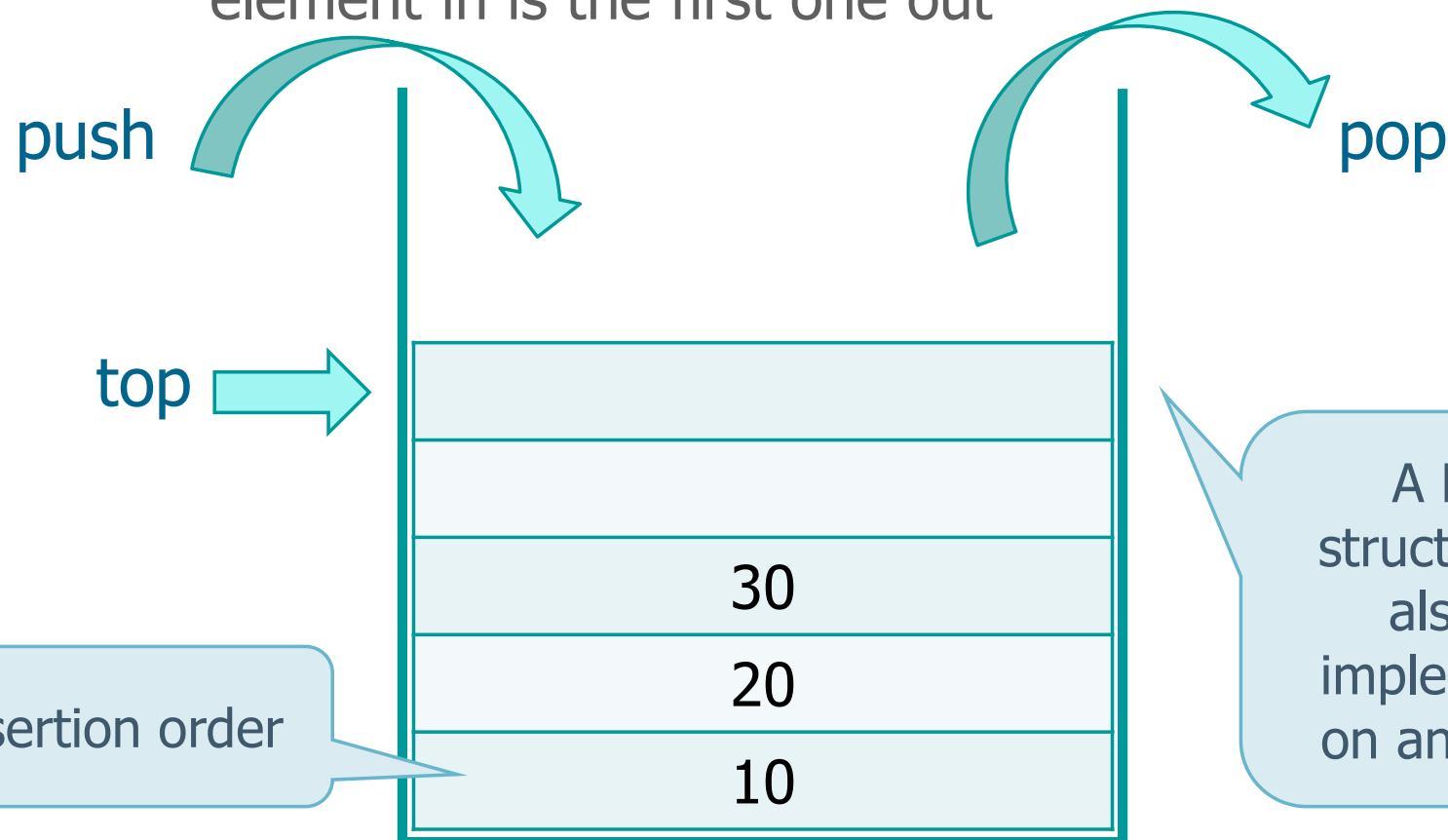
Push and Pop have a cost $O(1)$

LIFO linked list

- ❖ In computer science, such a data structure is often called **stack**
 - A stack is an abstract data type that serves as a collection of elements, with two principal operations, i.e., push and pop
 - Each push adds an element to the collection (whenever this is not full)
 - Each pop removes the last element that was added (whenever the collection is not empty)
 - This strategy creates a **pile** of objects, where insertions and extractions are done on the same side

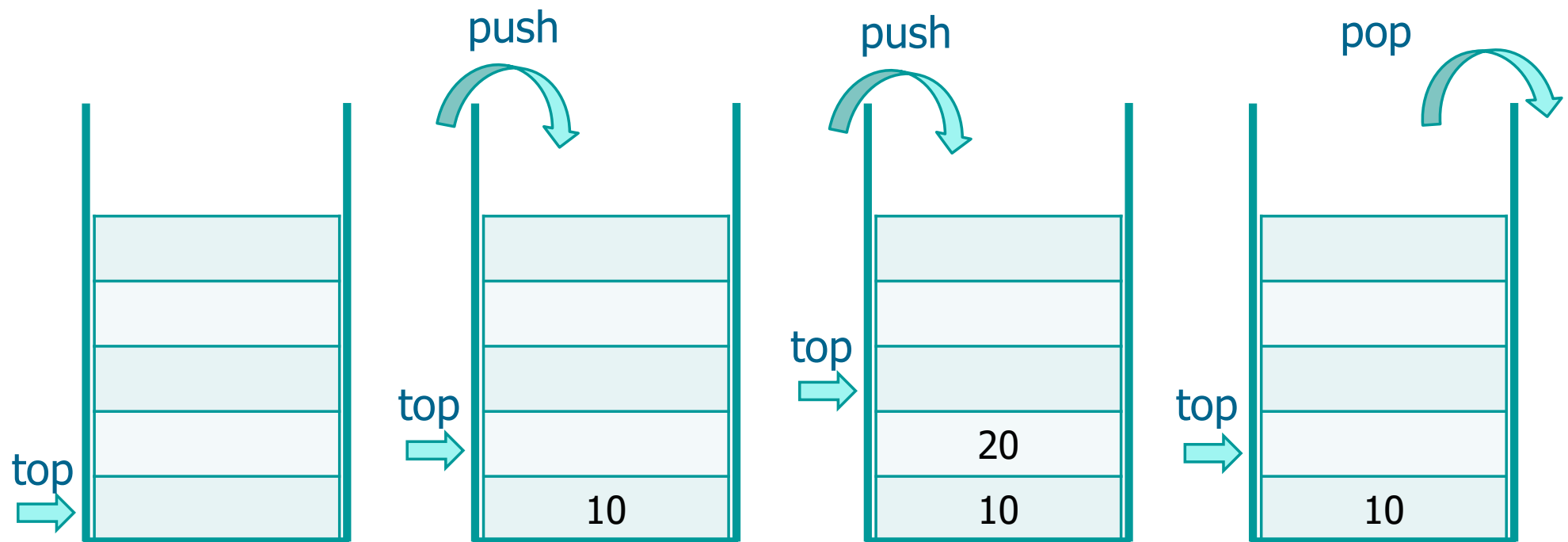
LIFO linked list

- As we insert and extract on the same end of the lists a **stack** is a **LIFO** structure
 - LIFO stands for Last In First Out, i.e., the last element in is the first one out



LIFO linked list

- A stack is usually represented as a pile of objects
 - A stack usually grows upward (towards smaller memory addresses)



Implementation

```
list_t *top;  
int val, status;
```

```
top = NULL;
```

```
do {  
    ...  
    top = push (top, val);
```

```
    ...  
    top = pop (top, &val, &status);
```

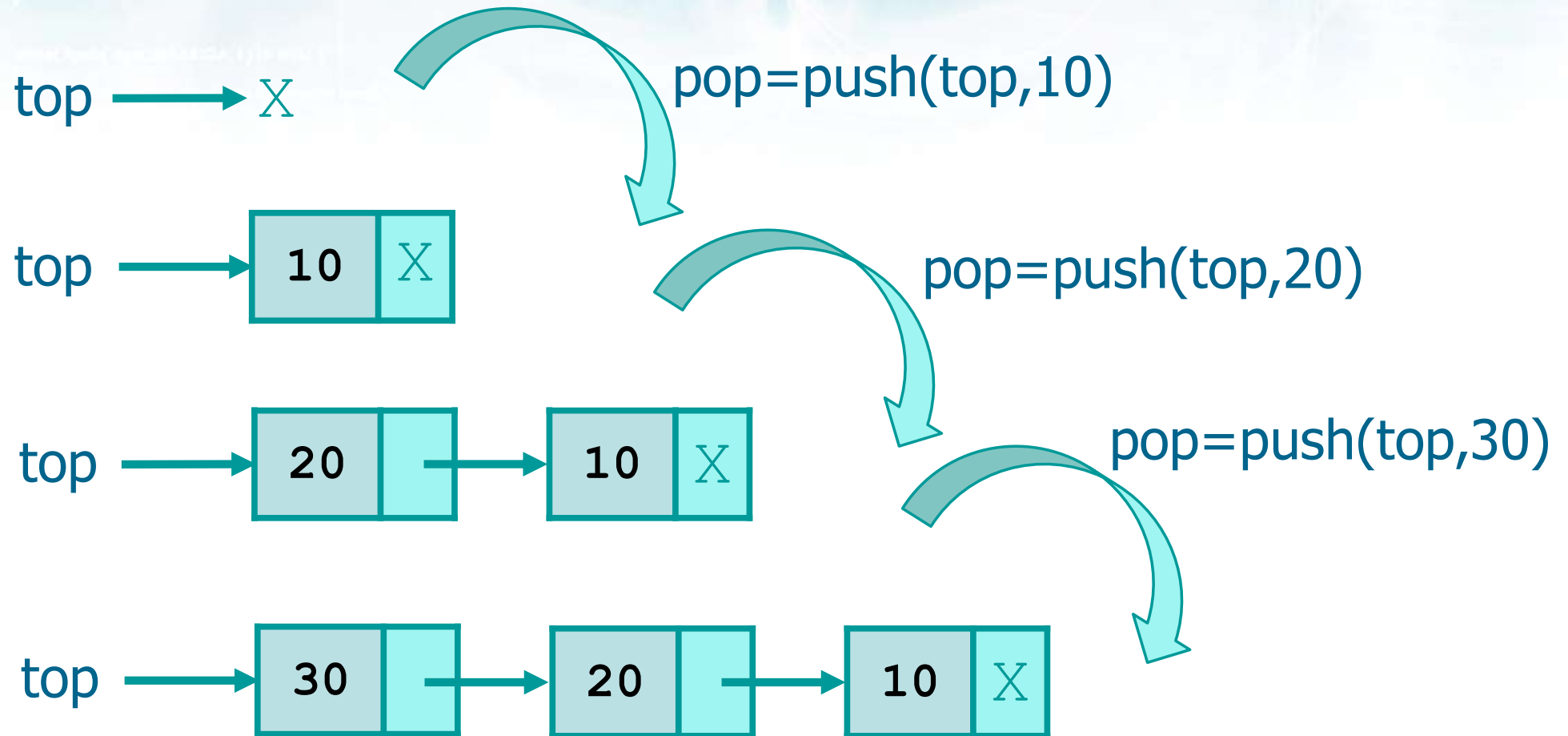
```
} while (...);
```

Type definition,
initialization, and
function calls

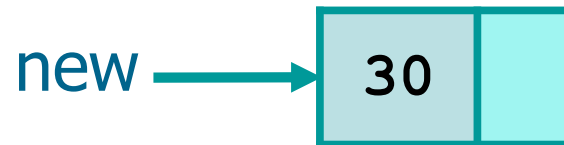
We suppose to exit if we cannot
insert an element, otherwise we
can return the operation status

If the stack is
empty we cannot
pop an element

Stack insertion: Push

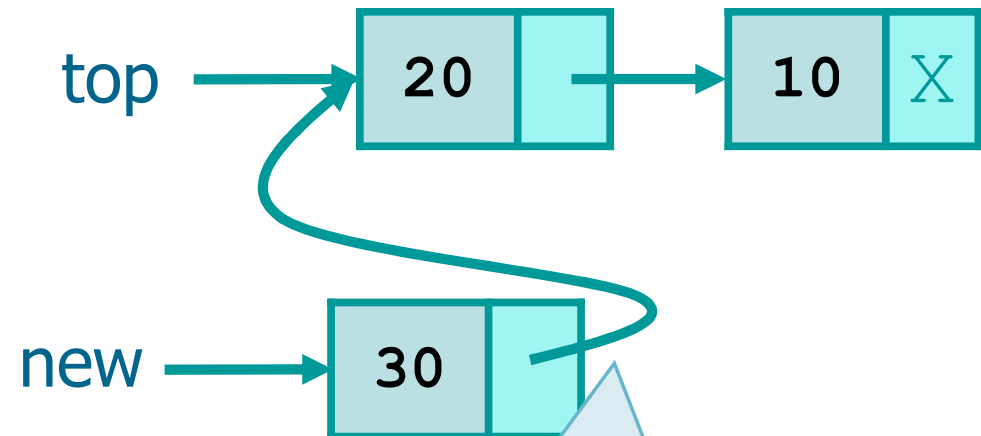
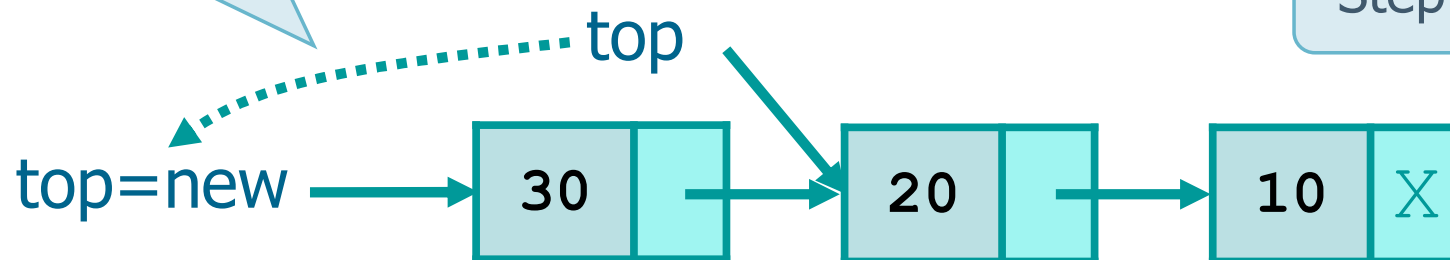


Stack insertion: Push



Step 1: new=new_element()

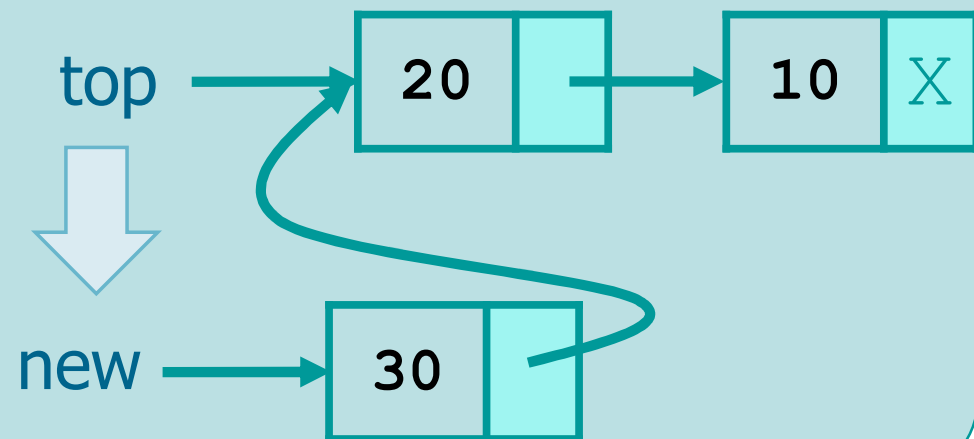
Step 3: top=new



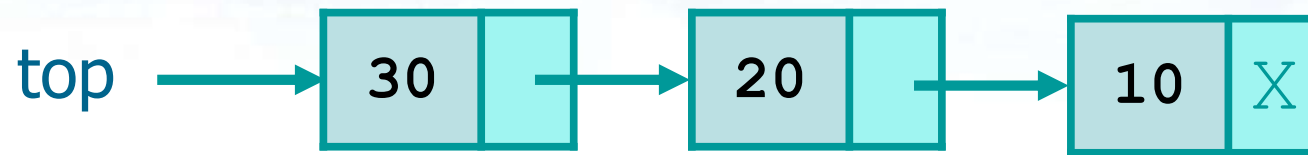
Step 2: new->next=top

Implementation

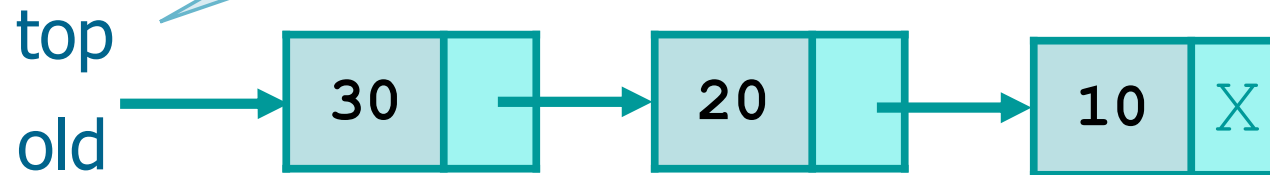
```
list_t *push (list_t *top, int val) {  
    list_t *new;  
  
    new = new_element ();  
  
    new->key = val;  
    new->next = top;  
    top = new;  
  
    return (top);  
}
```



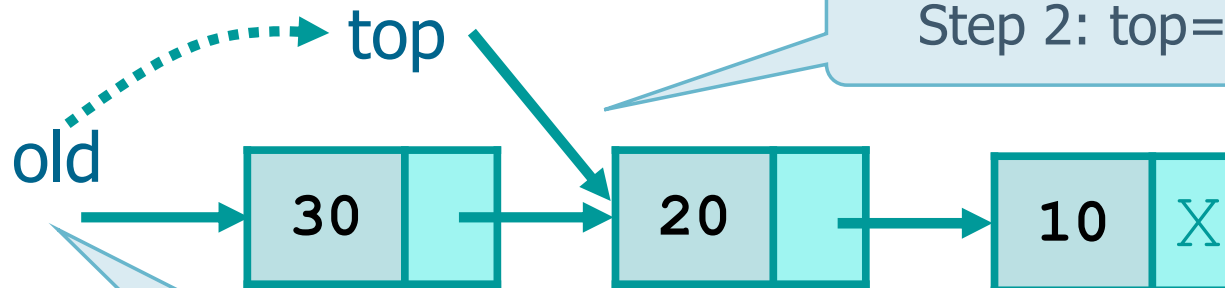
Stack extraction: Pop



Step 1: old=top



Step 2: top=top->next

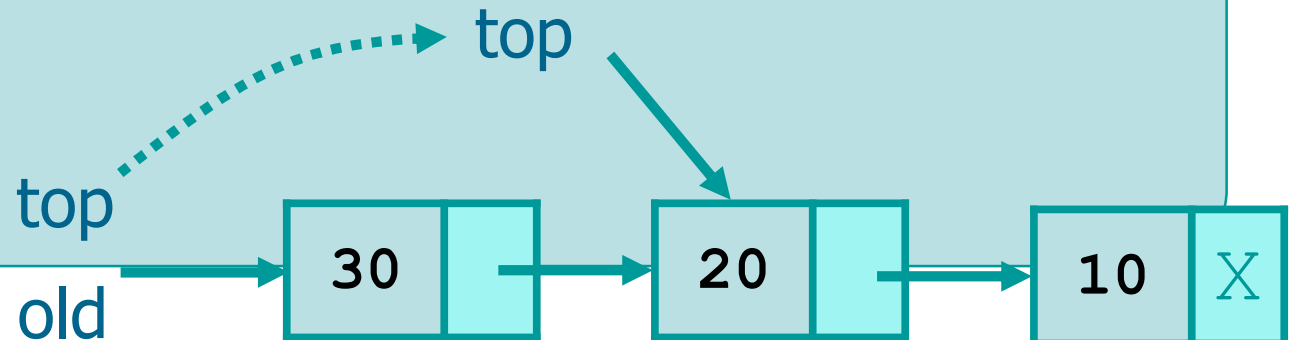


Step 3: free (old)

Implementation

```
list_t *pop (list_t *top, int *val, int *status) {  
    list_t *old;  
  
    if (top != NULL) {  
        *status = SUCCESS;  
        *val = top->key;  
        old = top;  
        top = top->next;  
        free (old);  
    } else {  
        *status = FAILURE;  
    }  
  
    return (top);  
}
```

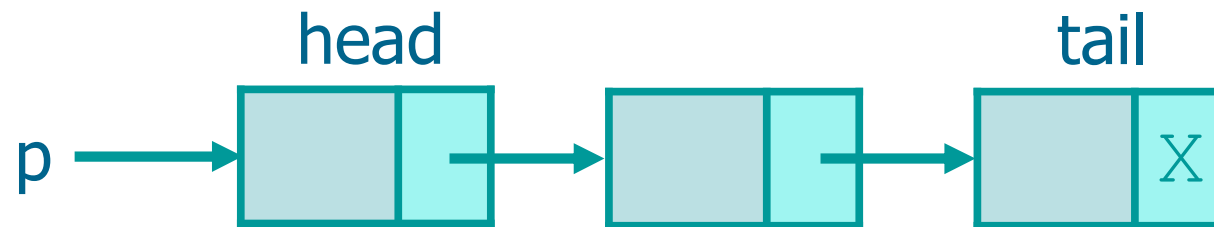
SUCCESS and FAILURE
are pre-defined constant



FIFO linked list

❖ A more complex possibility is to manipulate a simple linked list of elements by tail insertion and head extraction

- When we arrive we **wait** on the **tail**
- When we reach the **head** we are **served**



- This is essentially a **queue**

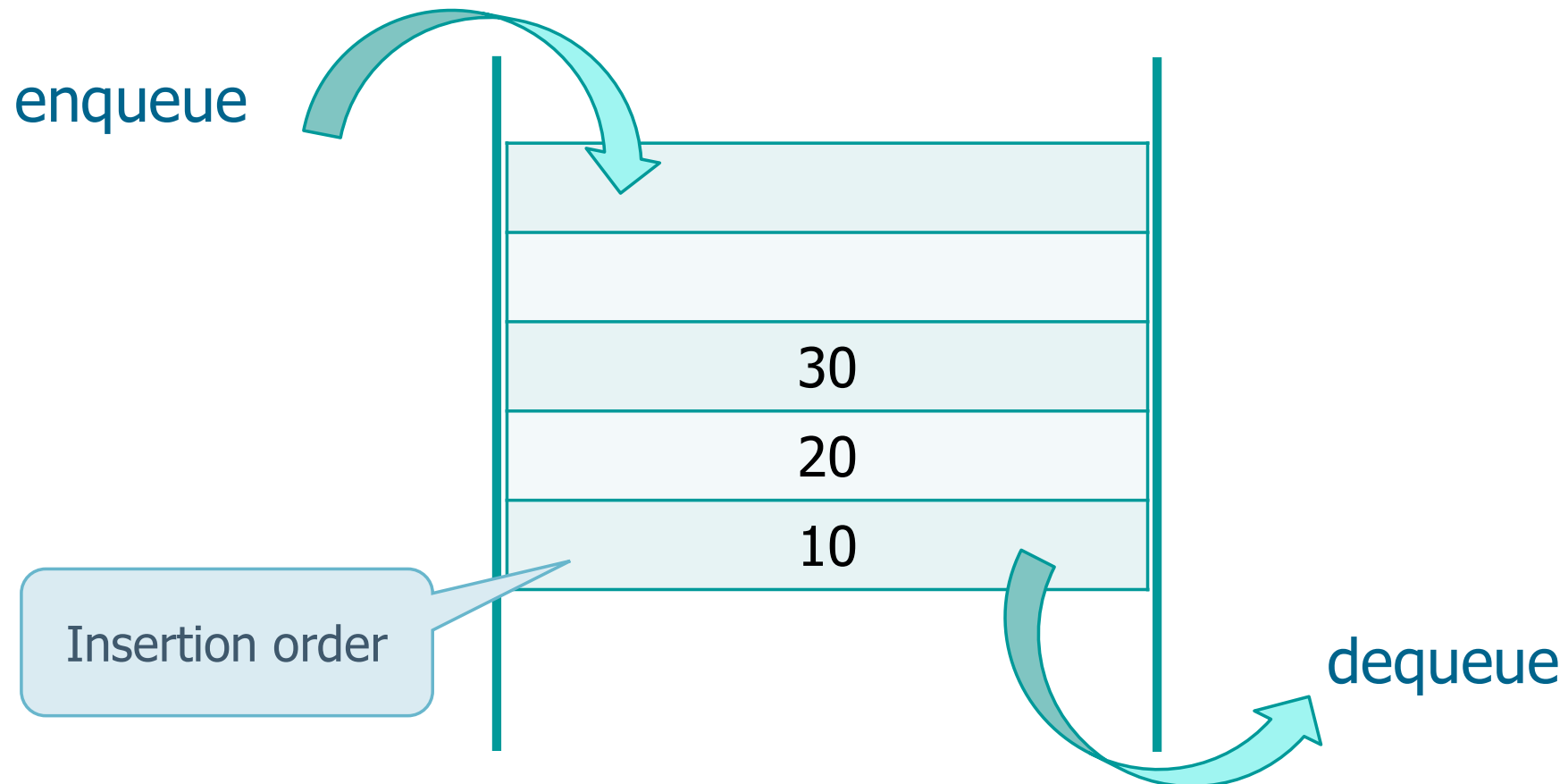
A callout box with a light blue background and a white border. It contains a diagram of a new node, a rectangle divided into a light blue data field labeled 'new' and a light blue pointer field. An arrow points from the right side of the callout box to the 'new' node. Below the diagram, the text reads: 'How do we insert on the tail? Cost $O(n)$? Hug?'

FIFO linked list

- ❖ In computer science, a **queue** is collection in which the entities are kept in order and the only) operations on the collection are the
 - Addition of entities to the rear terminal position, known as **enqueue**
 - Removal of entities from the front terminal position, known as **dequeue**
- ❖ This makes the queue a First-In-First-Out (FIFO) data structure
- ❖ In a FIFO data structure, the first element added to the queue will be the first one to be removed

FIFO linked list

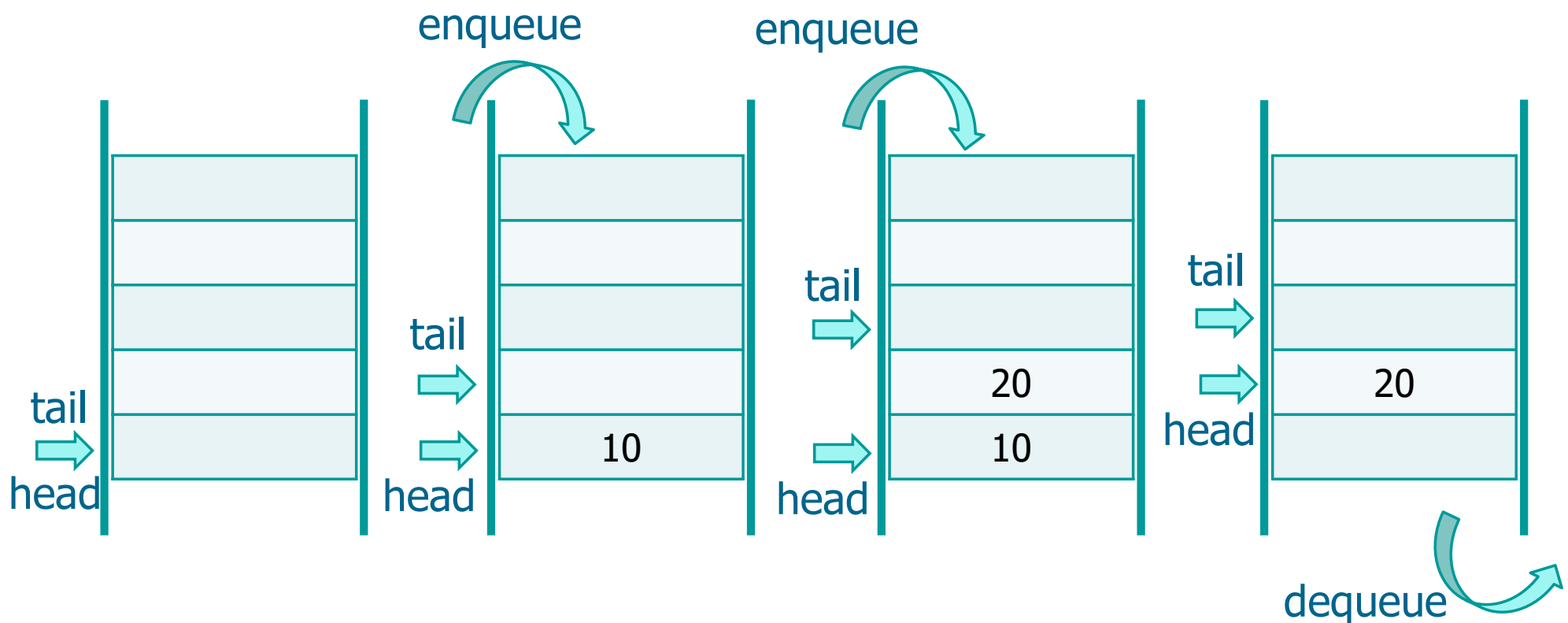
- As we insert and extract on the two opposite extremes a **queue** is a **FIFO** structure
 - FIFO stands for First In First Out, i.e., the first element in is the first one out



LIFO linked list

➤ When we

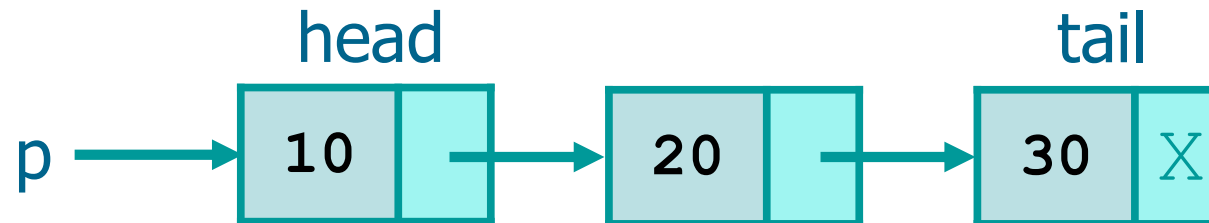
- Enter the structure, we wait on the tail
- Exit the structure, we leave from the head



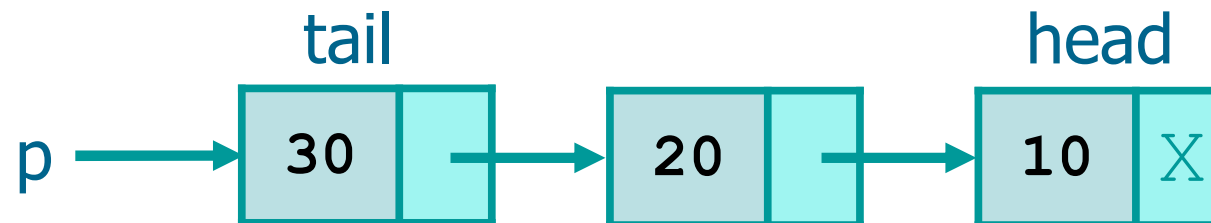
FIFO linked list

❖ A FIFO logic is difficult to implement with the standard linked list structure

- To insert on the tail we have to visit the entire list (the cost is $O(n)$ and is not admissible)



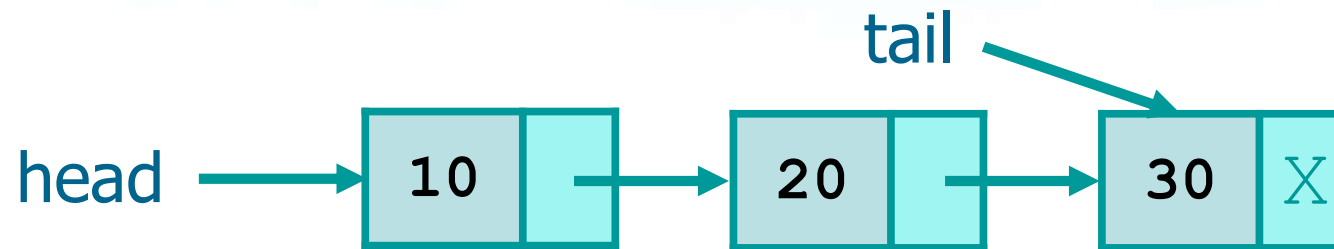
- We could also "invert" the logic



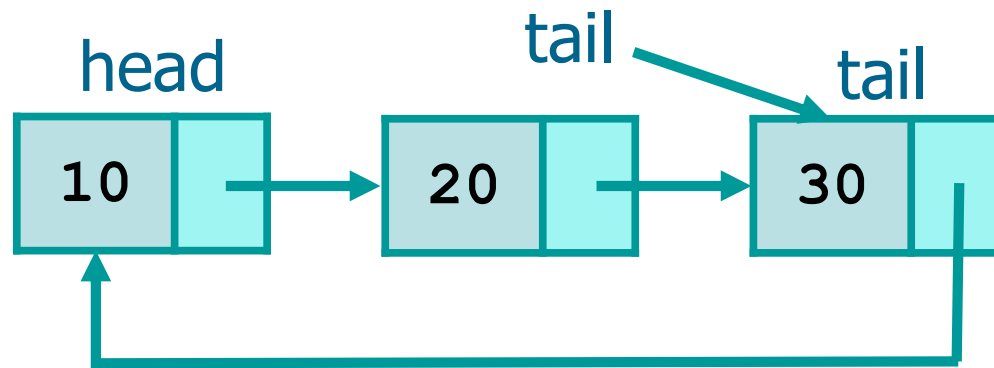
- But now, to extract from the head we would have to visit the entire list

FIFO linked list

- Thus, we either use two pointers



- OR we use a circular list and save the **head** pointer by using the pointer to the last element equal to NULL to reference the head



Both implementations are possible, as, $\text{head} == \text{tail} \rightarrow \text{next}$

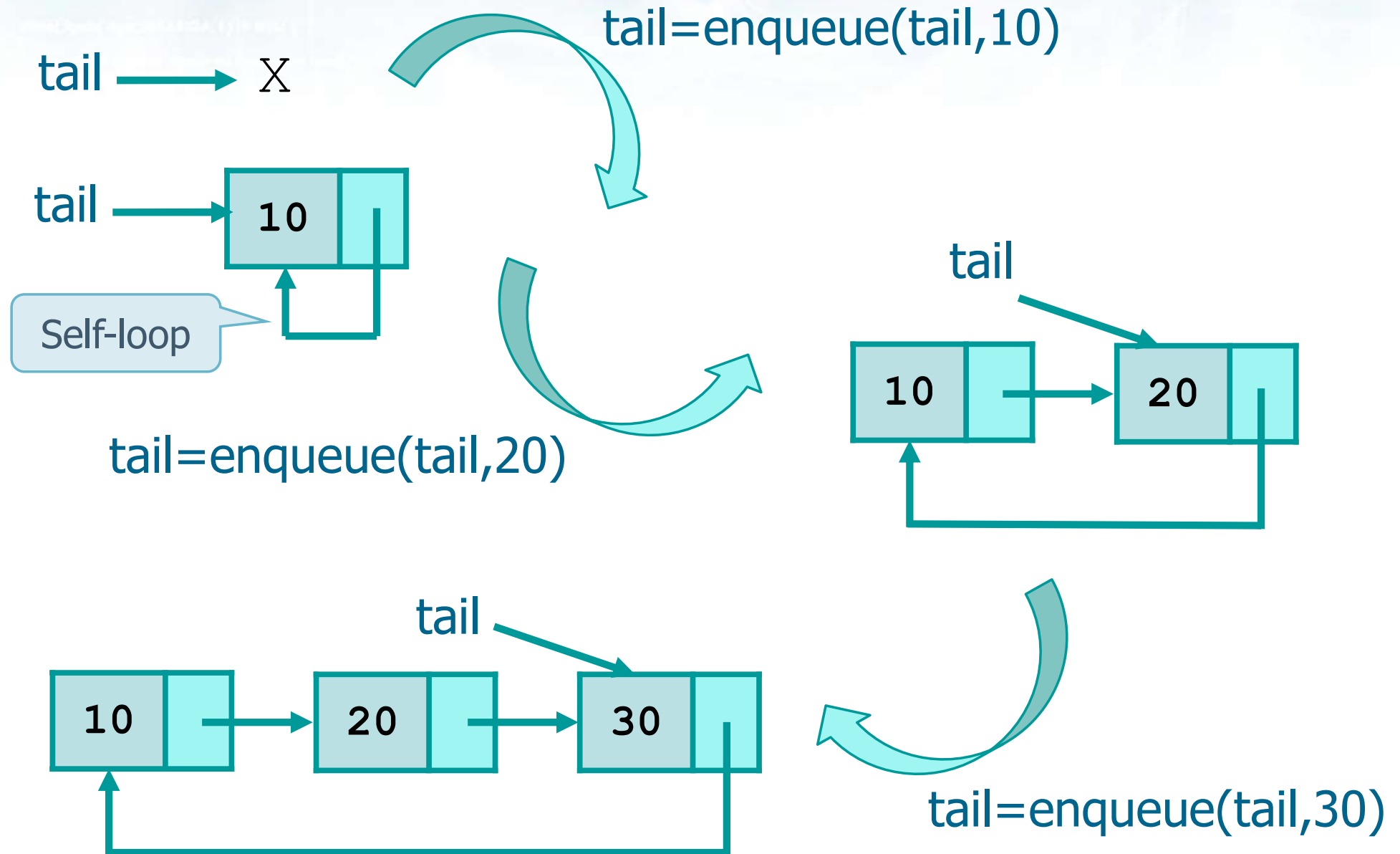
We analyze the second one

Implementation

Type definition,
initialization, and
function calls

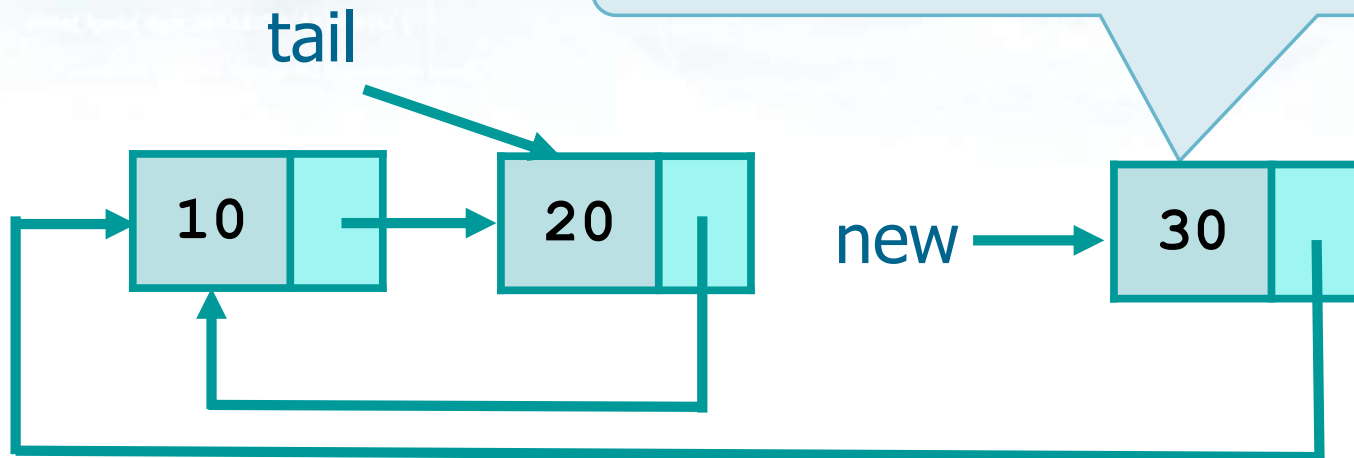
```
struct e *tail;  
int val, status;  
  
tail = NULL;  
  
do {  
    ...  
    tail = enqueue (tail, val);  
  
    ...  
    tail = dequeue (tail, &val, &status);  
} while (...);
```

Enqueue

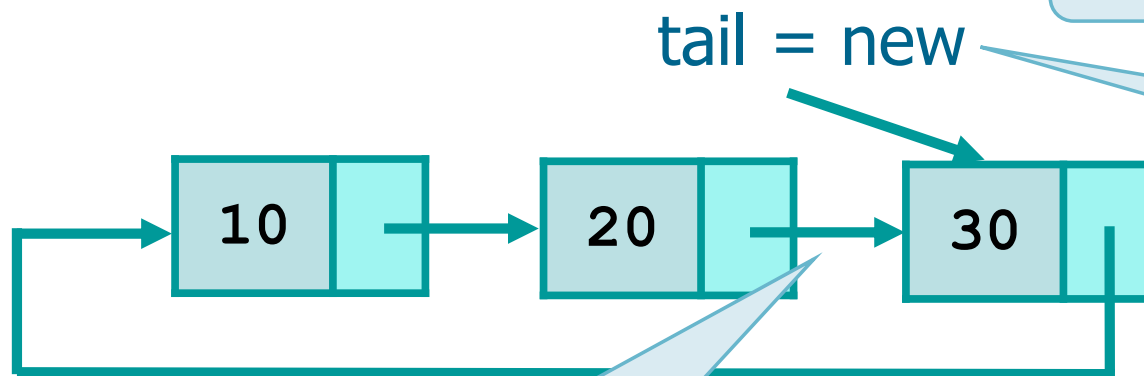


Enqueue

Step 1: `new = new_element()`



Step 2: `new->next = tail->next`



Step 4: `tail = new`

Step 3: `tail->next = new`

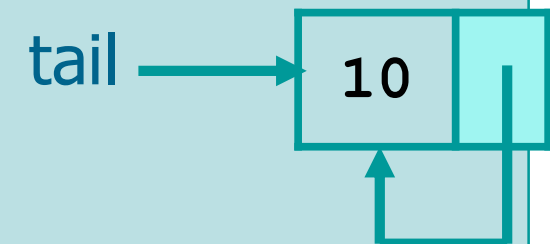
tail = new

Implementation

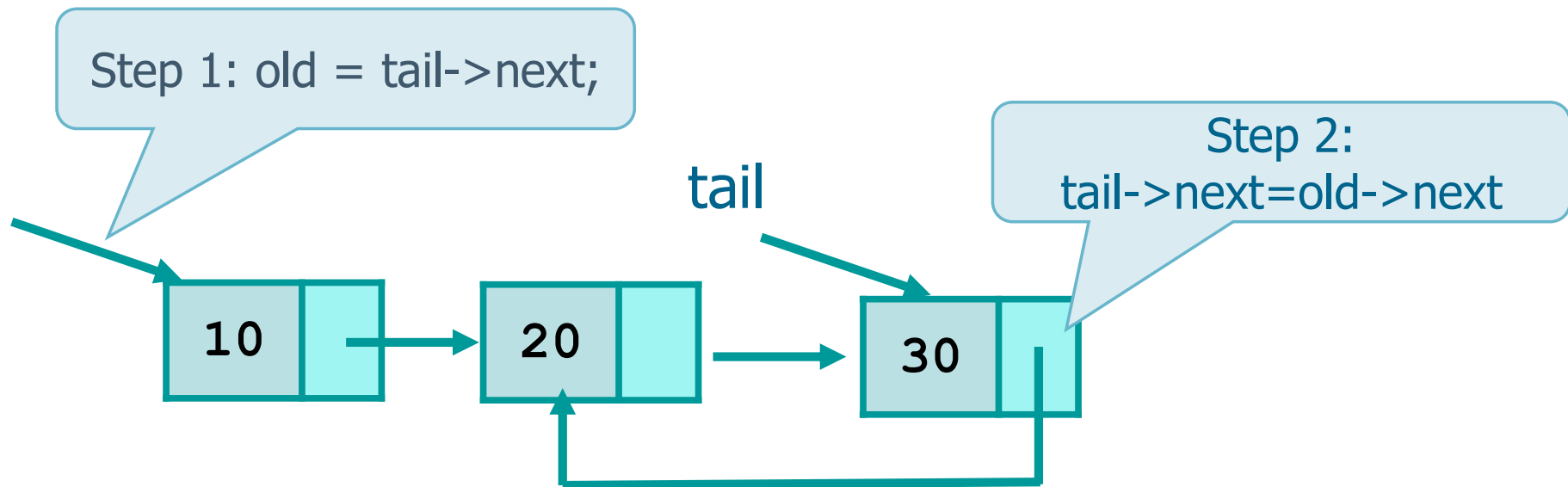
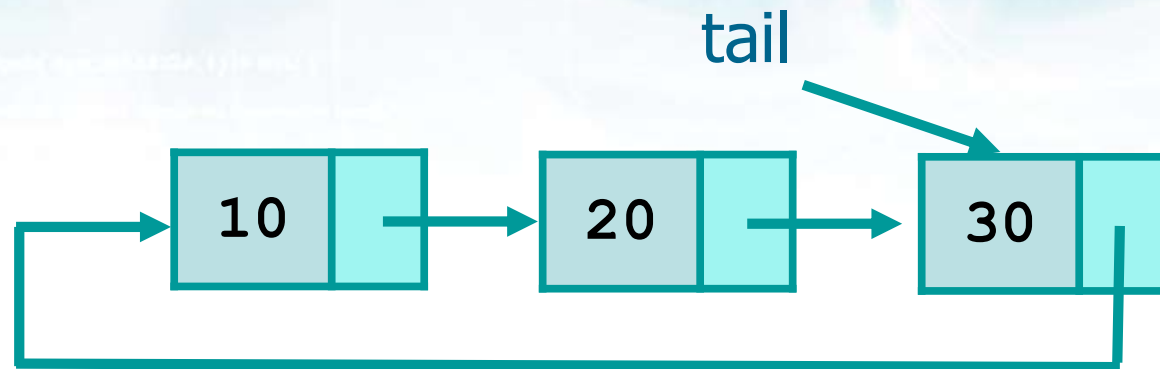
```
list_t *enqueue (list_t *tail, int val) {  
    list_t *new;  
  
    new = new_element ();  
    new->key = val;  
  
    if (tail==NULL) {  
        tail = new;  
        tail->next = tail;  
    } else {  
        new->next = tail->next;  
        tail->next = new;  
        tail = new;  
    }  
  
    return (tail);  
}
```

The first insertion (and the self-loop) must be implemented aside

tail → X



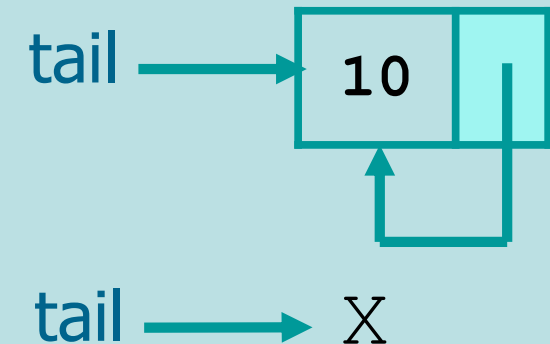
Deque



Implementation

```
list_t *dequeue (list_t *tail, int *val, int *status) {  
    list_t *old;  
    if (tail != NULL) {  
        *status = SUCCESS;  
        if (tail == tail->next) {  
            *val = tail->key;  
            free (tail);  
            tail = NULL;  
        } else {  
            old = tail->next;  
            *val = old->key;  
            tail->next = old->next;  
            free (old);  
        }  
    } else {  
        *status = FAILURE;  
    }  
    return (tail);  
}
```

The last extraction (and the self-loop) must be implemented aside



Ordered linked list

- ❖ A linked list can be also maintained sorted, i.e., ordered by increasing (or decreasing) values of its keys (integer values in our examples)
- ❖ With a sorted list
 - Insertions and extractions are usually performed in the middle of the list
 - Head and tail insertions are particular cases (first and last element in the sorted set) and must be implemented as corner-cases
 - A search can terminate unsuccessfully when a record with a key larger (or smaller) than the search key is found

Implementation

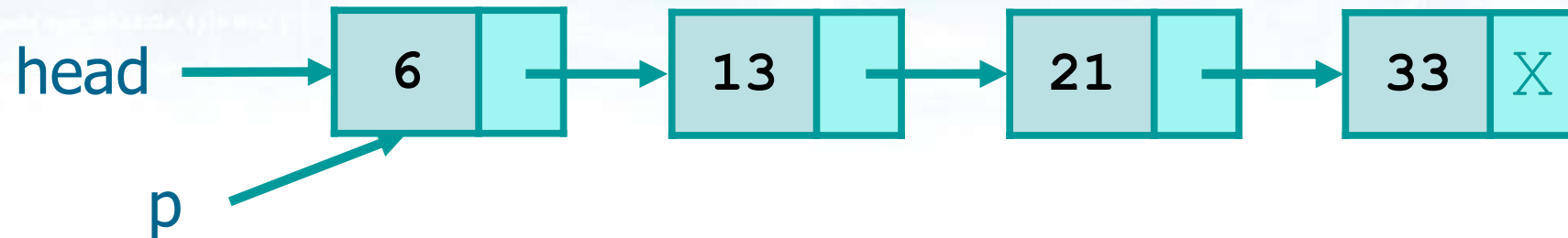
Type definition,
initialization, and
function calls

```
do {  
    ...  
    head = insert (head, val);  
  
    ...  
    search (head, val);  
  
    ...  
    head = extract (head, val);  
} while ( ... );
```

Search

- ❖ A search can terminate
 - Successfully, when we find the key
 - Unsuccessfully when a record with a key larger (or smaller) than the search key is found
- ❖ In any case
 - We can stop the search as soon as the key we are looking for become larger than the current node's key
 - This make the search more efficient
 - Nevertheless, the search still has a linear cost ($O(n)$) in the number of elements stored into the list

Implementation

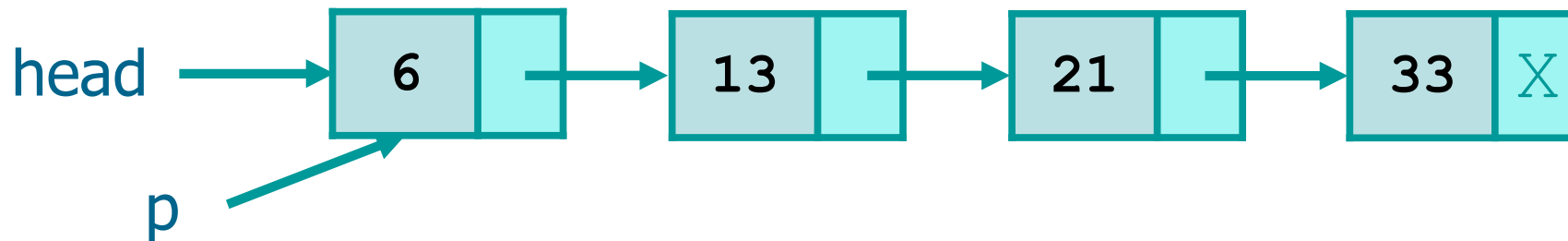


```
void search (list_t *head, int val) {  
    list_t *p;  
  
    p=head;  
    while (p!=NULL) {  
        if (p->val==val)  
            return p;  
        if (p->val<val)  
            p=p->next;  
    }  
  
    return (NULL) ;  
}
```

We can stop before; constant time advantage; the process is always linear in the number of elements

Insertion

- ❖ To insert an element into a list, we have to
 - Individuate its correct position
 - Insert the element in that position
- ❖ Unfortunately, when we look for a key, we stop our search too late



```
p=head;
while (p!=NULL && p->val<val)
    p=p->next;
... insertion ...
```

$p == \text{NULL}$ OR $p \rightarrow \text{val} \geq \text{val}$
Too late to insert !!!

Insertion

- ❖ There are several approaches to solve the problem
 - Use two pointers to individuate two consecutive elements
 - Move them along the list in a synchronized way
 - Use the rightmost to compare and the leftmost to insert
 - Use the pointer of the pointed element to make the comparison
 - Reach the element referenced by the pointed element to compare, use the direct pointer to insert
- ❖ Do not forget to deal with corner-cases (e.g., empty list)

Implementation

```
list_t *insert (list_t *head, int val) {  
    list_t *p, *q=head;  
  
    p = new_element ();  
    p->val = val;  
    p->next = NULL;  
  
    if (head==NULL || val<head->val) {  
        p->next = head;  
        return p;  
    }  
    while (q->next!=NULL && q->next->val<val) {  
        q = q->next;  
    }  
    p->next = q->next;  
    q->next = p;  
    return head;  
}
```

Create a new element

Head insertion

q is used to insert
q->next is used to compare

Implementation

Example: val=15

```
list_t *insert (list_t *head, int val) {
```

```
    list_t *p, *q=head;
```

q->next->val

q->next



```
    if (head->val == val) {
```

```
        p->next = head;
```

```
        return p;
```

```
    }  
    while (q->next!=NULL && q->next->val<val) {
```

```
        q = q->next;
```

```
    }  
    p->next = q->next;
```

```
    q->next = p;
```

```
    return head;
```

```
}
```


Implementation

Example: val=15

```
list_t *insert (list_t *head, int val) {  
    list_t *p, *q=head;  
  
    p = new_element ();  
    p->val = val;  
    p->next =  
  
    if (head==NULL || val<head->val) {  
        p->next = head;  
        return p;  
    }  
  
    while (q->next!=NULL && q->next->val<val) {  
        q = q->next;  
    }  
    p->next = q->next;  
    q->next = p;  
    return head;  
}
```

The diagram illustrates the insertion of a new node (15) into a linked list. The original list has nodes 6, 13, 21, and 33. A new node with value 15 is being inserted between 13 and 21. The diagram shows the pointers head, q, and p, and the next pointers of the nodes. The node 33 has a null next pointer (X).

- head** points to the first node (6).
- q** points to the node with value 13.
- p** points to the new node with value 15.
- The node with value 6 points to the node with value 13.
- The node with value 13 points to the node with value 21.
- The node with value 21 points to the node with value 33.
- The node with value 33 has a null next pointer (X).
- The new node with value 15 has its next pointer pointing to the node with value 21.

Extraction

- ❖ To extract (or delete) an element from a list, we have to proceed as for insertion
 - We need two pointers or a look-ahead of one element (the pointer of the pointed element)
 - We need to cope with corner-cases
 - Empty list, element on the head, no element

Implementation

```
list_t *extract (list_t *head, int val) {  
    list_t *p, *q=head;  
    if (head == NULL) {  
        fprintf(stderr, "Error: empty list\n");  
        return NULL;  
    }  
    if (val == head->val) {  
        p = head->next;  
        free(head);  
        return p;  
    }  
    while (q->next!=NULL && q->next->val<val) {  
        q = q->next;  
    }  
}
```

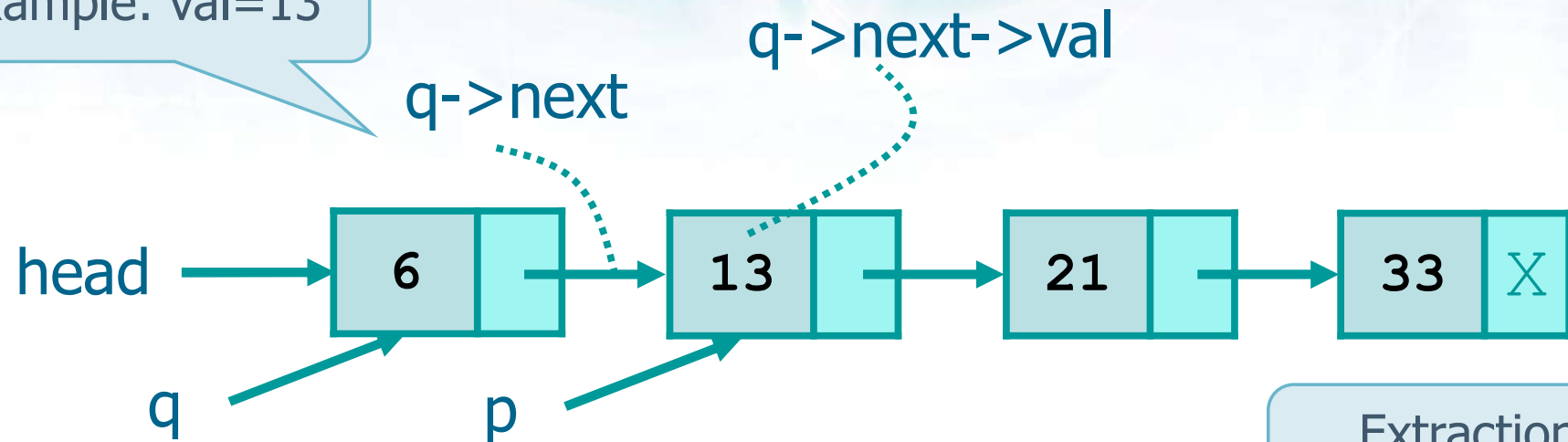
Empty list

Head extraction

Search
(using q->next)

Implementation

Example: val=13



Extraction
(using q)

```
if (q->next!=NULL && q->next->val==val) {  
    p = q->next;  
    q->next = p->next;  
    free(p);  
} else {  
    fprintf(stderr, "Element NOT found.\n");  
}  
return head;  
}
```

Double-linked lists

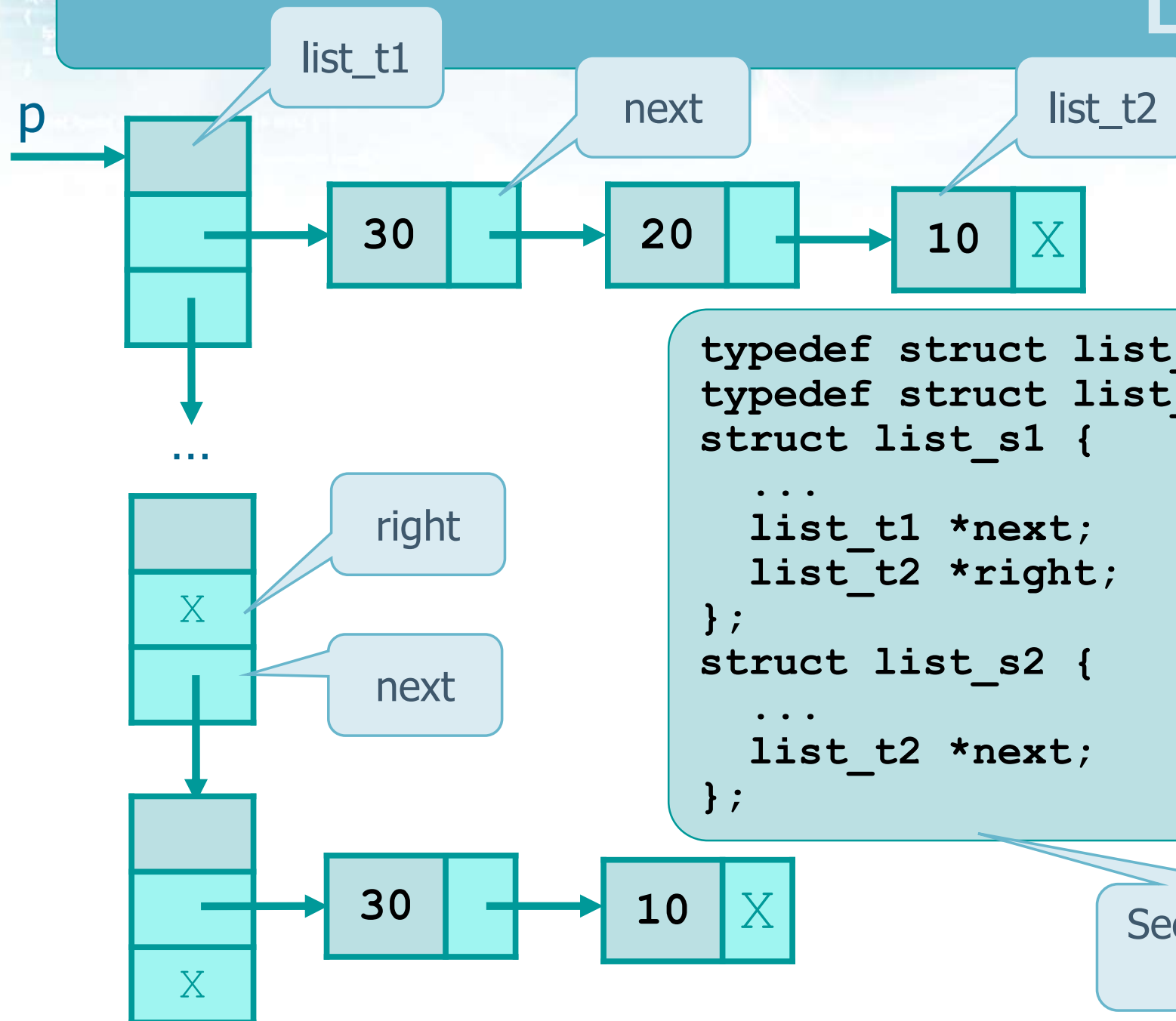


```
typedef struct list_s list_t;
struct list_s {
    ...
    list_t *left, *right;
};
```

- ❖ We need to manipulate two pointers to insert or extract elements

See laboratory to practice

List of lists



```
typedef struct list_s1 list_t1;
typedef struct list_s2 list_t2;
struct list_s1 {
    ...
    list_t1 *next;
    list_t2 *right;
};
struct list_s2 {
    ...
    list_t2 *next;
};
```

See laboratory to practice