Linked Lists

Common Linked Lists

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

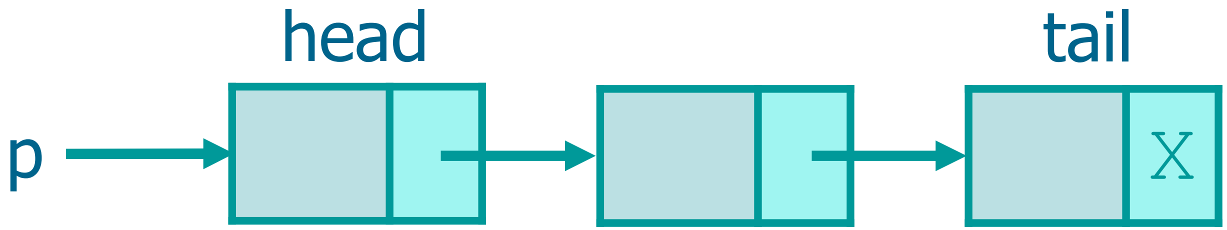
Introduction

- Basic linked list operations can be used to maintaint 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 linked lists with different formats
  - Doubly-linked lists
  - List of lists

LIFO linked list

LAST IN FIRST OUT

- 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

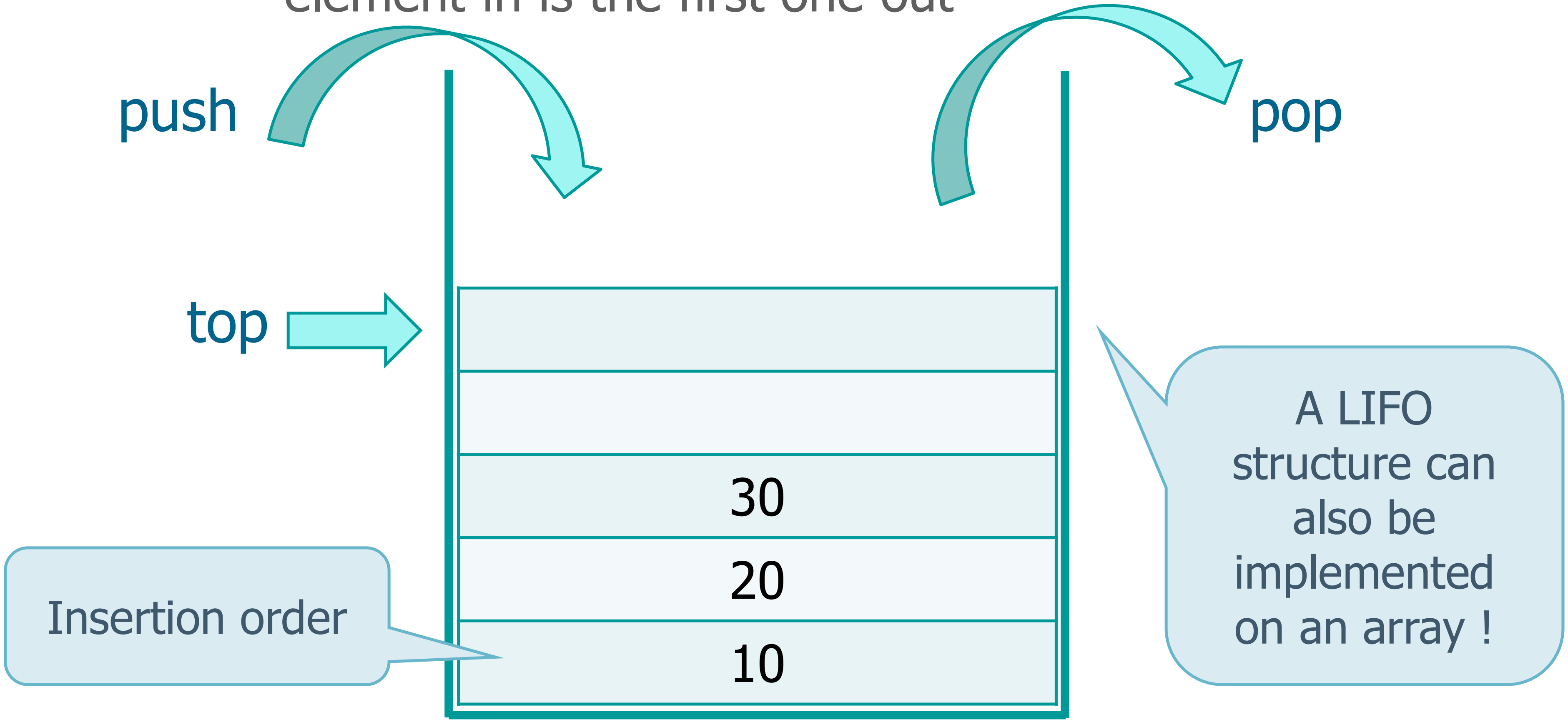
Push and Pop have a cost O(1)

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

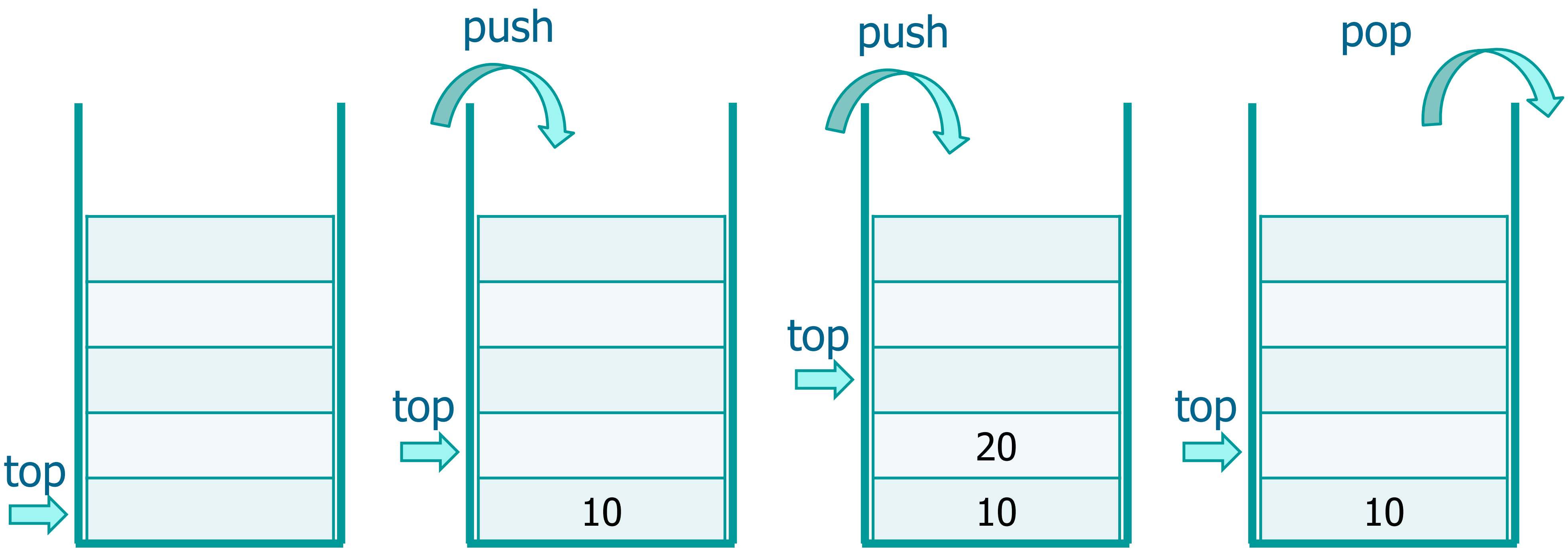
LIFO linked list

- 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

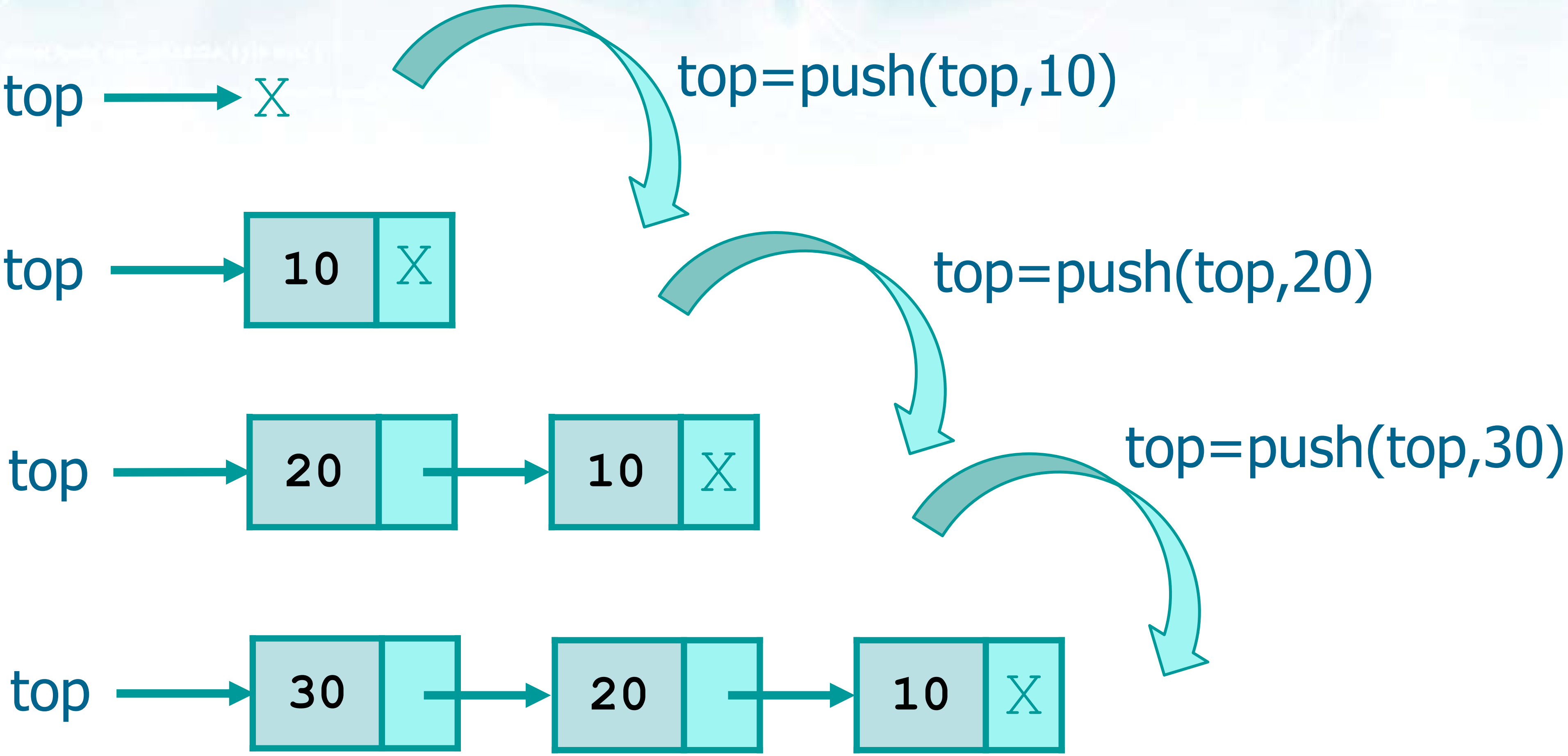
```
list_t *top;
int val, status;

top = NULL;

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

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

} while (...);
```

Type definition, initialization, and function calls

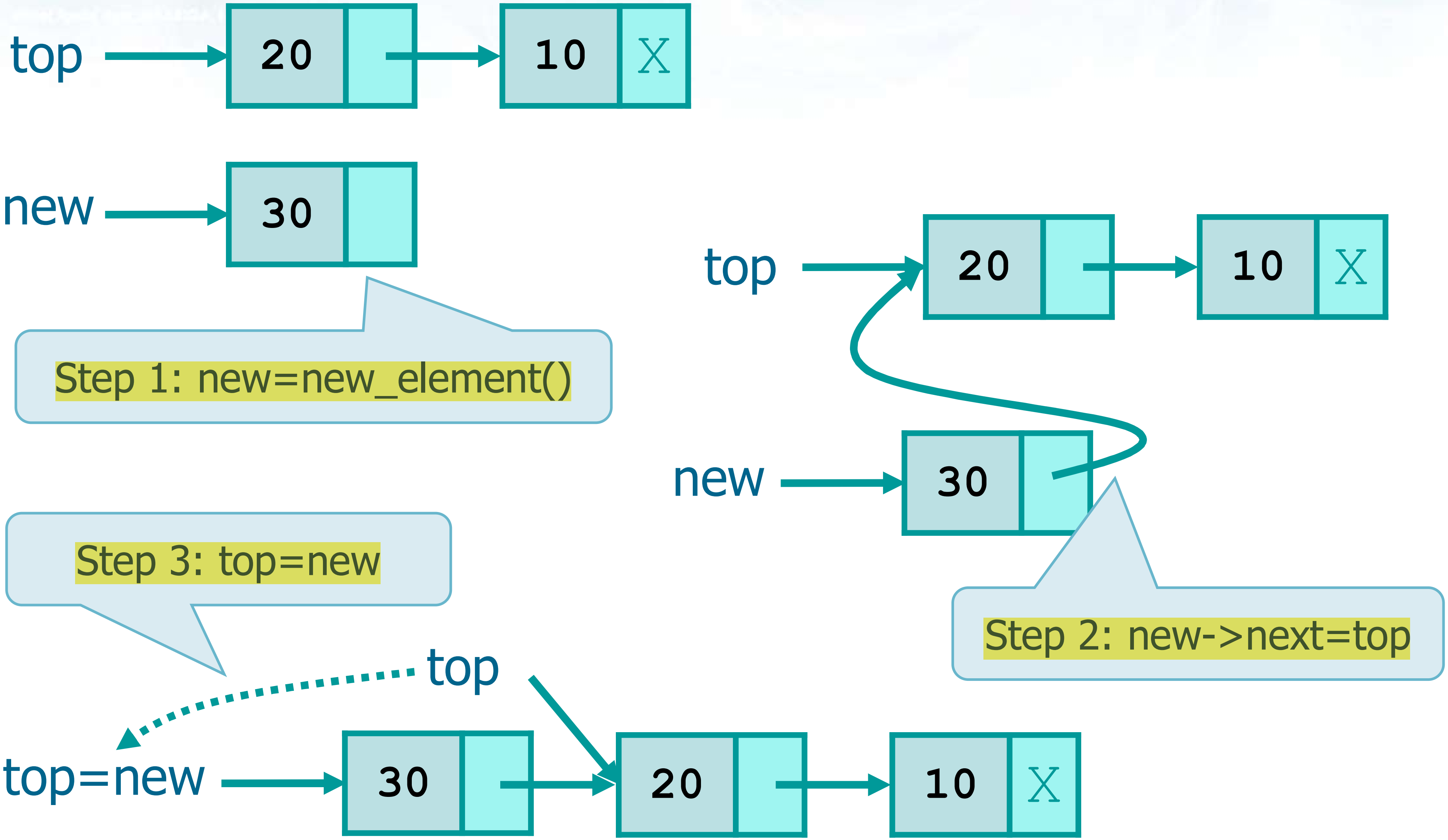
We suppose to exit if we cannot insert an element, otherwise we can return the operation status

If the stack is empty we cannot pop an element

Stack insertion: Push

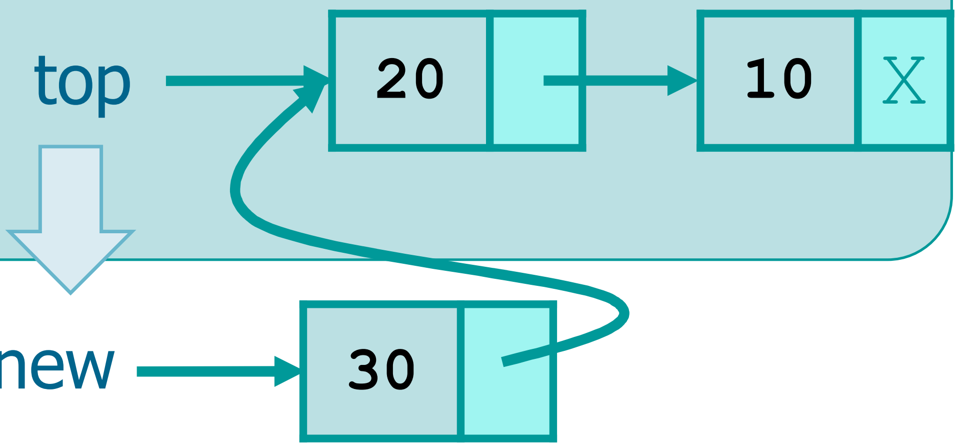


Stack insertion: Push

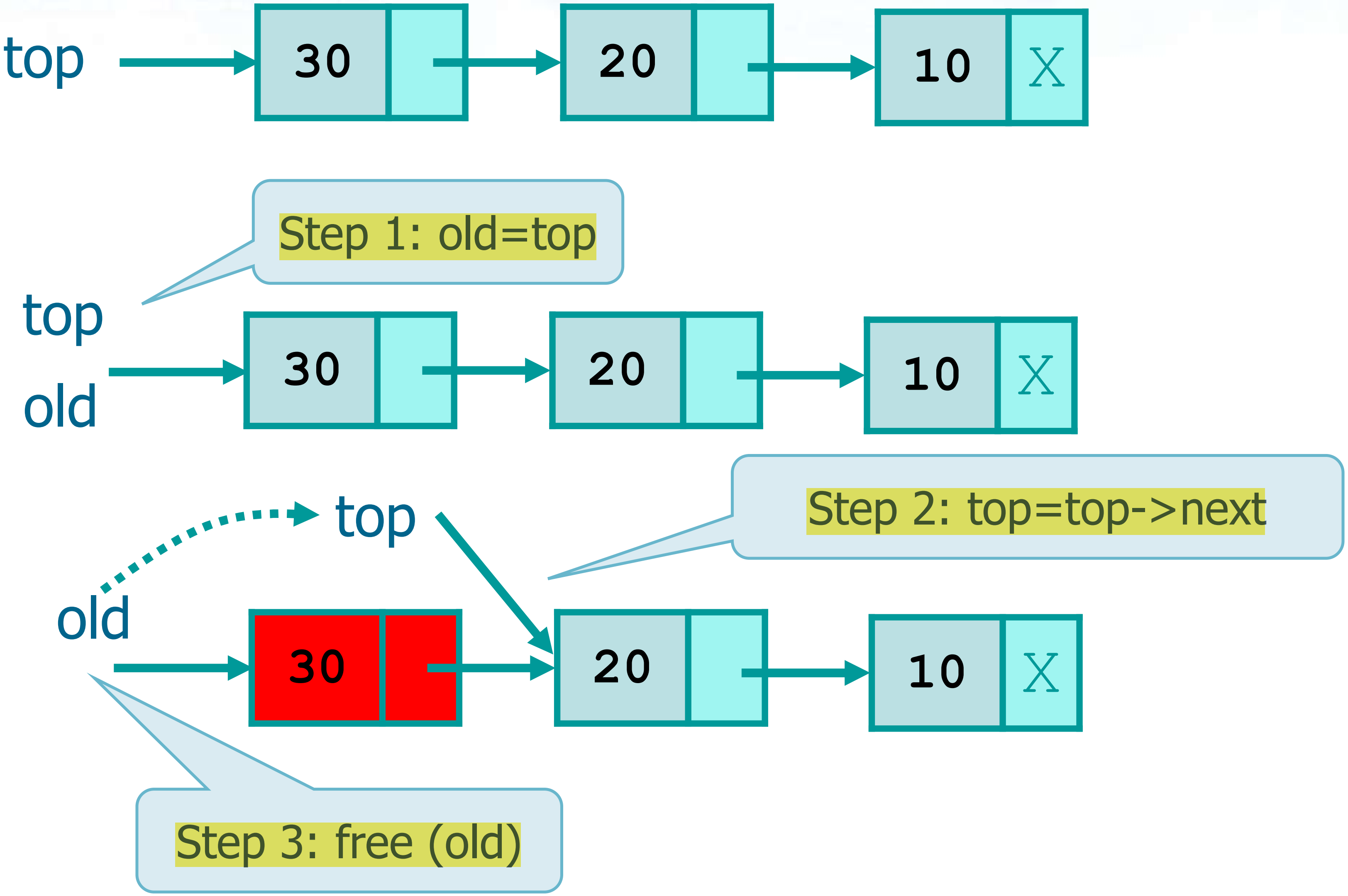


Implementation

```
        list_t *push (list_t *top, int val) {
          list_t *new;

Step 1:   new = new_element ();

          new->key = val;
Step 2:   new->next = top;
Step 3:   top = new;

          return (top);
        }
```



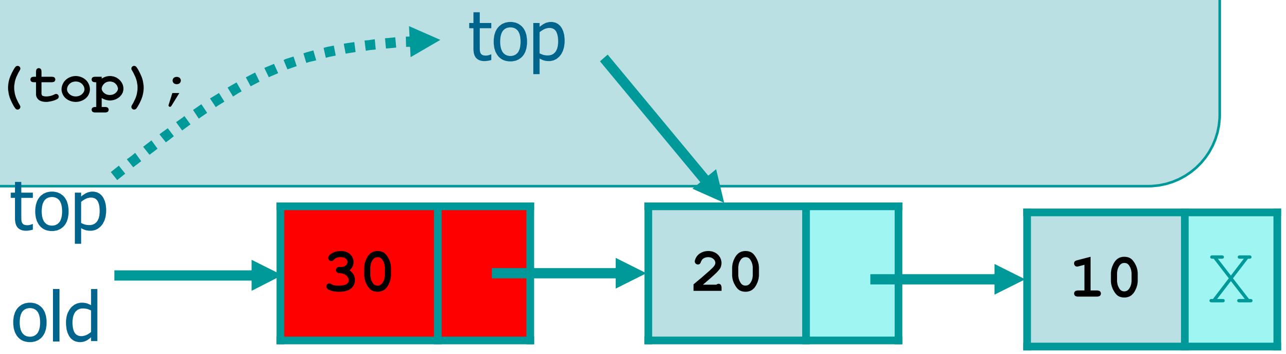
Stack extraction: Pop



Implementation

```
        list_t *pop (
          list_t *top, int *val, int *status) {
          list_t *old;

          if (top != NULL) {
            *status = SUCCESS;
            *val = top->key;
Step 1:     old = top;
Step 2:     top = top->next;
Step 3:     free (old);
          } else {
            *status = FAILURE;
          }

          return (top);
        }
```

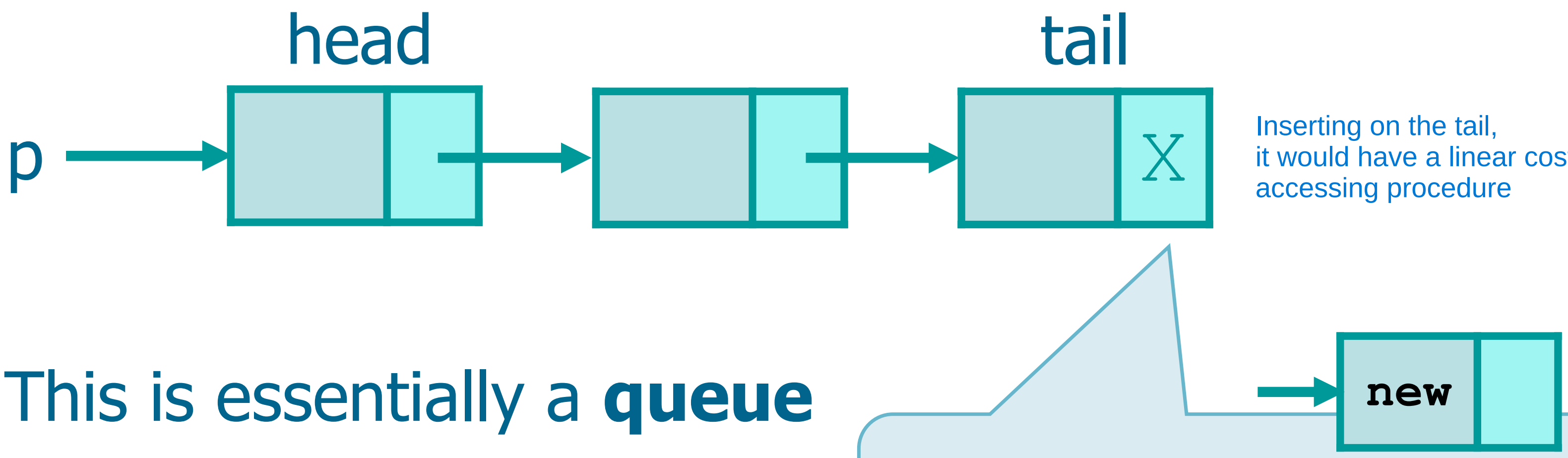
SUCCESS and FAILURE are pre-defined constant



FIFO linked list

FIRST IN FIRST OUT

- 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**



Inserting on the tail,
it would have a linear cost because of the accessing procedure

  - This is essentially a **queue**

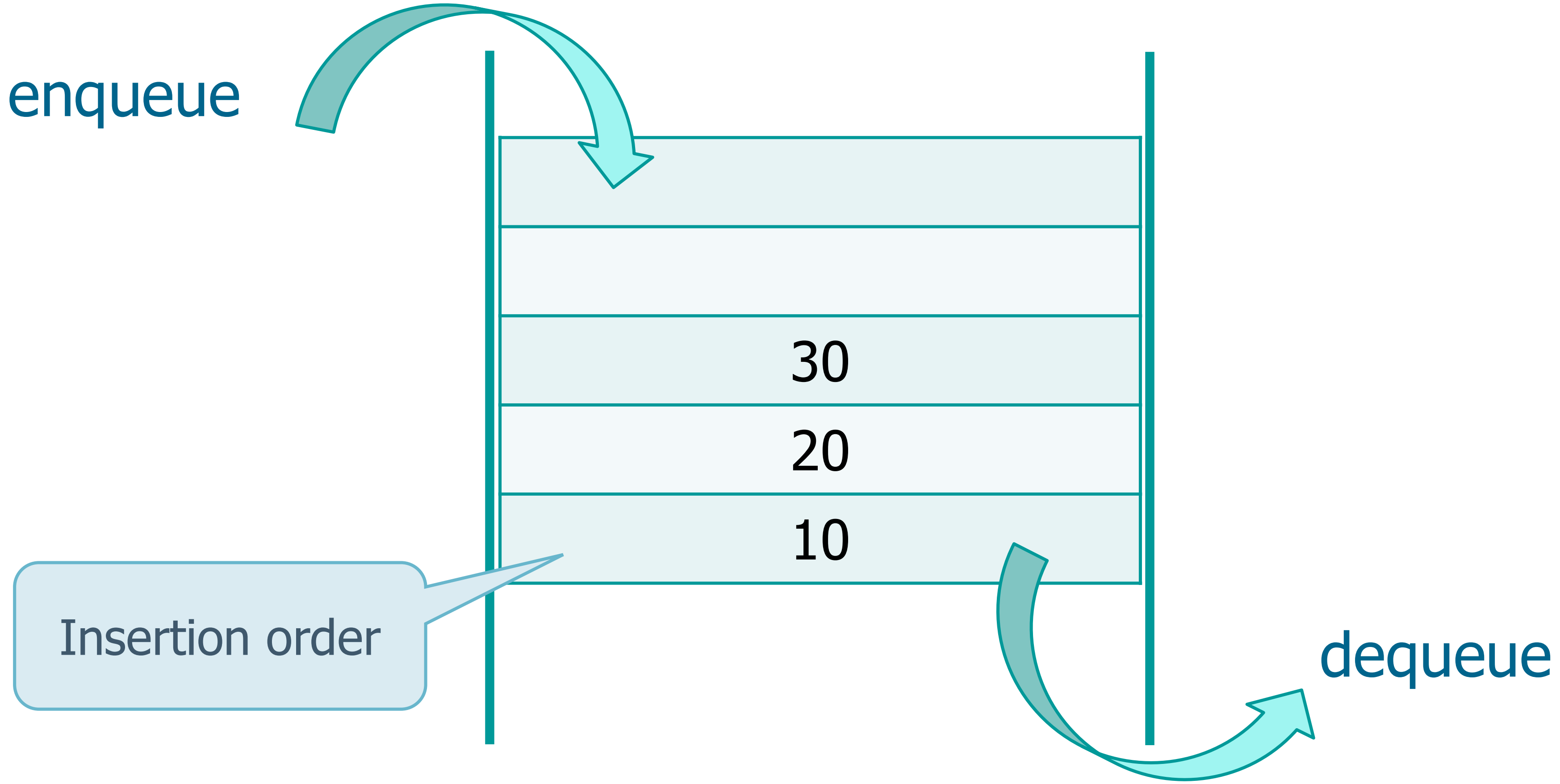
How do we insert on the tail?
Cost O(n)? Hug?

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

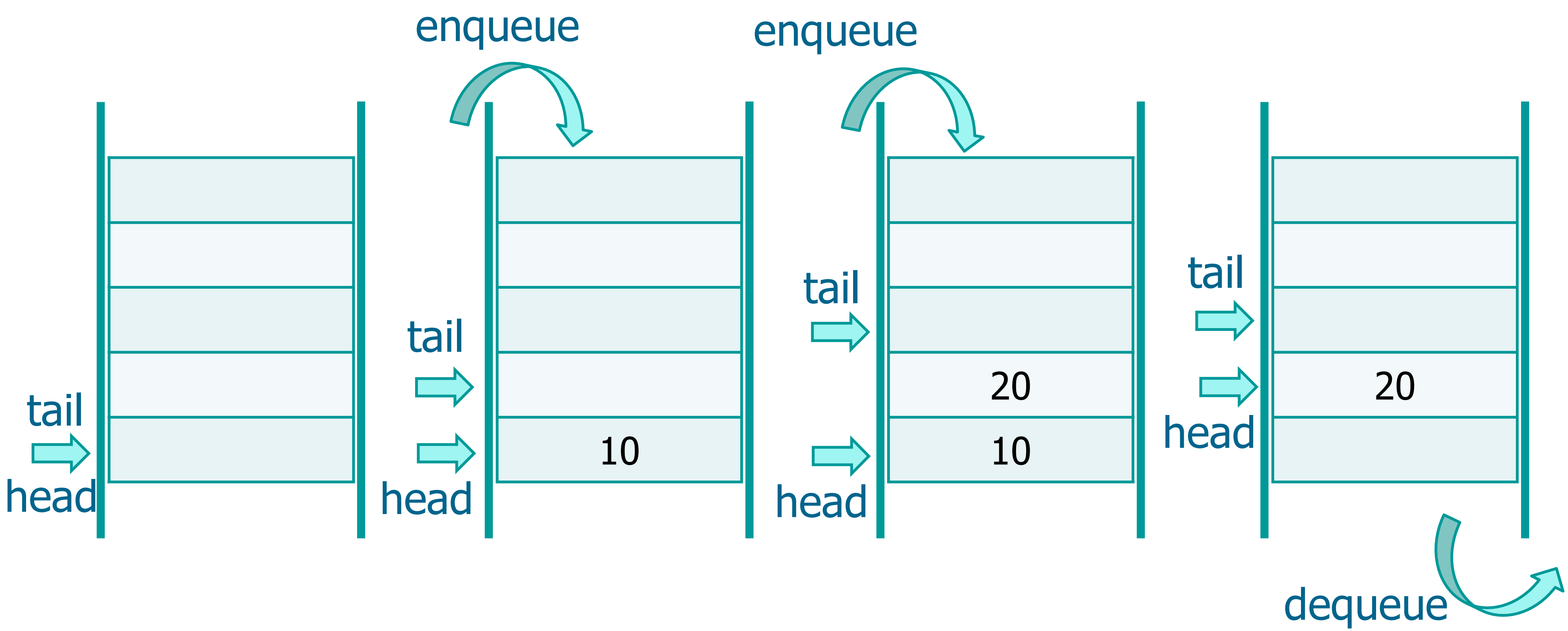
FIFO linked list

- 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



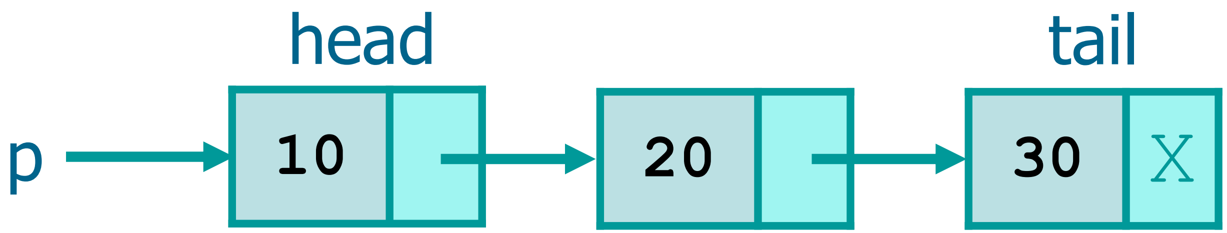
LIFO linked list

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

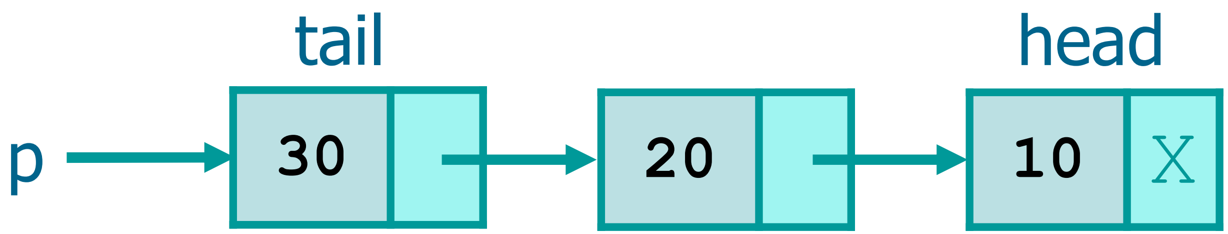


FIFO linked list

- 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)



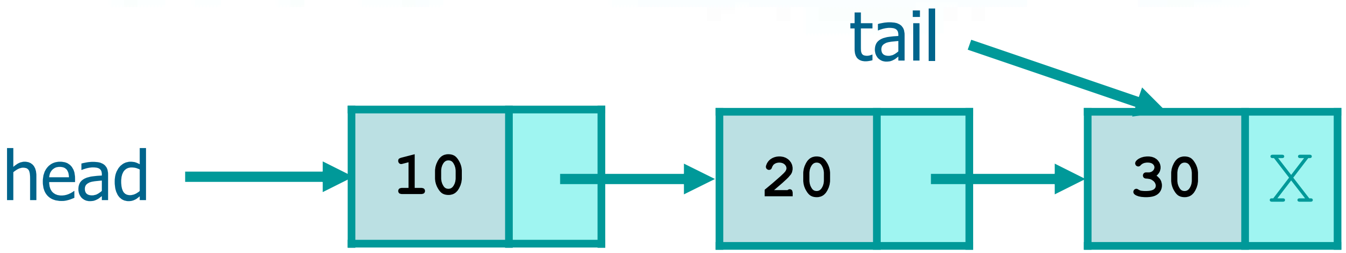
  - We could also "invert" the logic



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

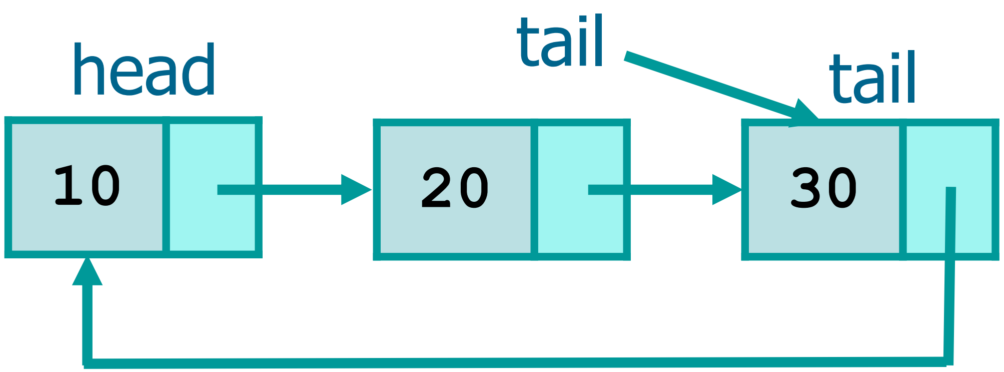
FIFO linked list

- 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

circular list:
no need for double pointer



Both implementations are possible, as, head==tail->next

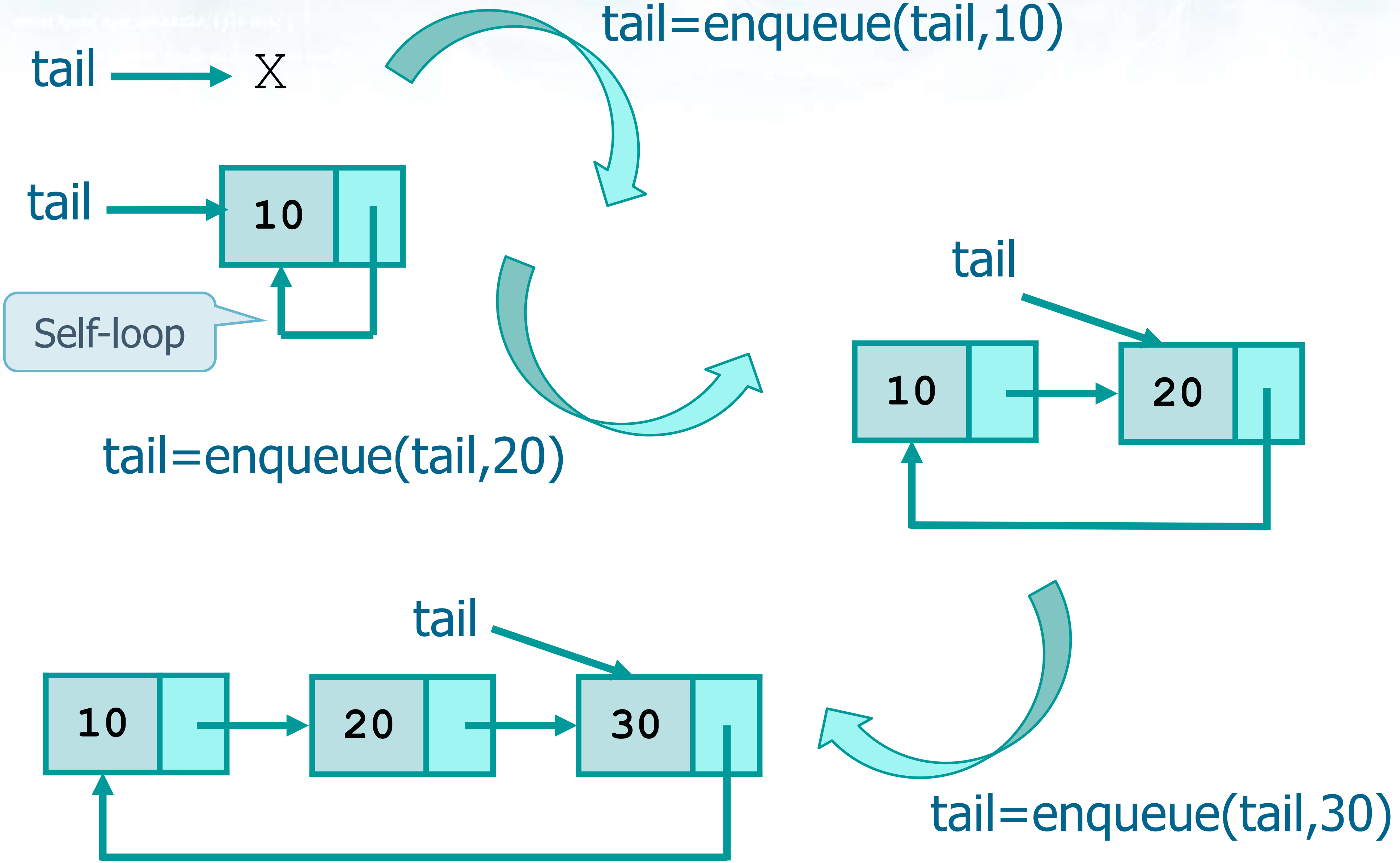
We analyze the second one

Implementation

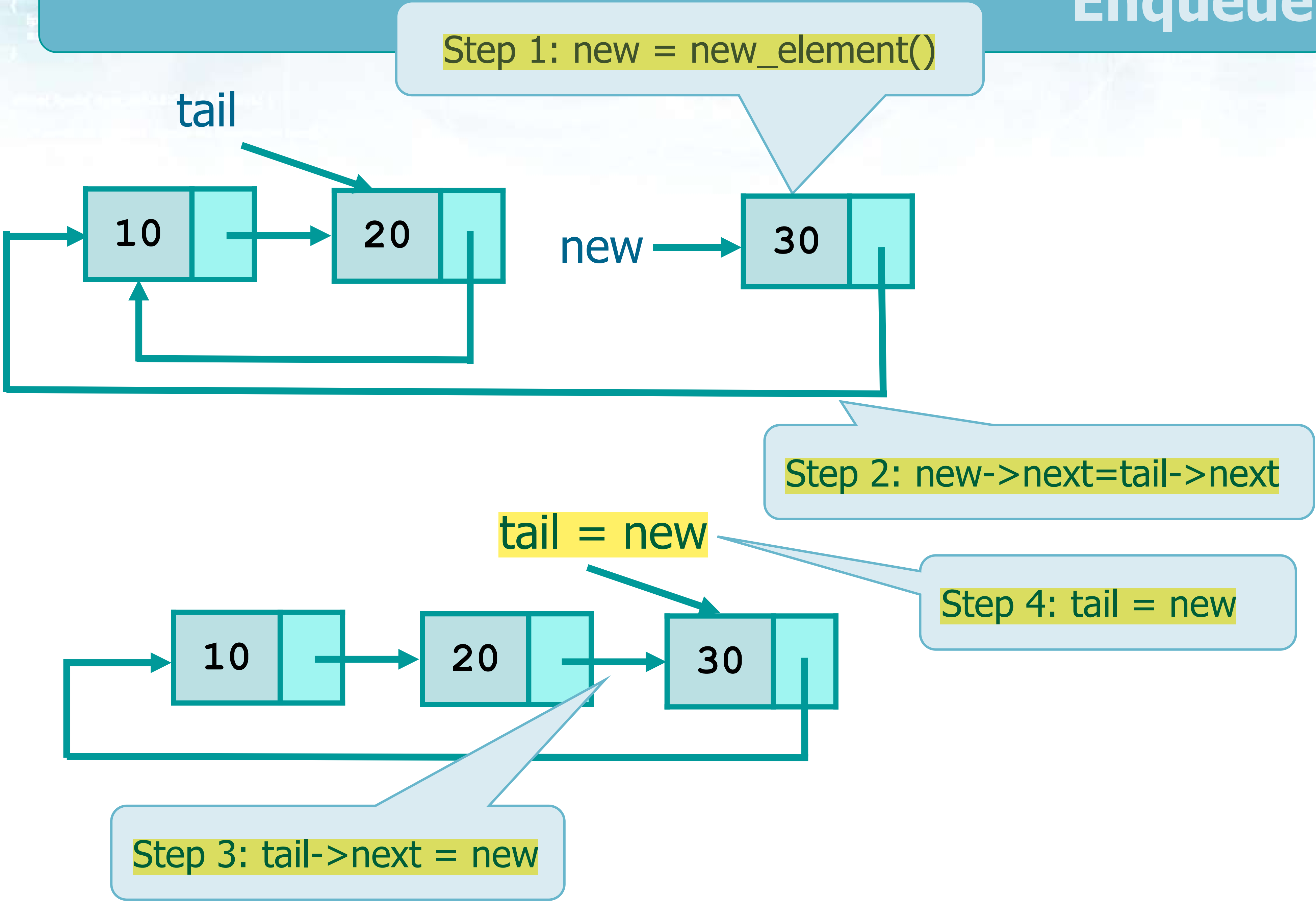
Type definition, initialization, and function calls

```
struct e *tail;
int val, status;

tail = NULL;

do {
  ...
  tail = enqueue (tail, val);

  ...
  tail = dequeue (tail, &val, &status);

} while (...);
```

Enqueue


tail
X
tail=enqueue(tail,10)
tail
10
Self-loop
tail
10
20
tail=enqueue(tail,20)
tail
10
20
30
tail=enqueue(tail,30)

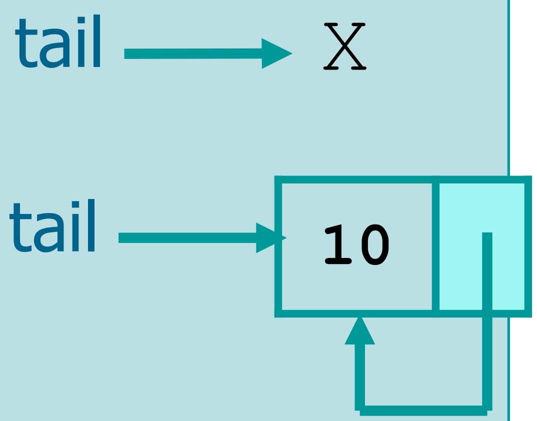
Enqueue


Step 1: new = new_element()
tail
10
20
new
30
Step 2: new->next=tail->next
tail = new
Step 4: tail = new
10
20
30
Step 3: tail->next = new

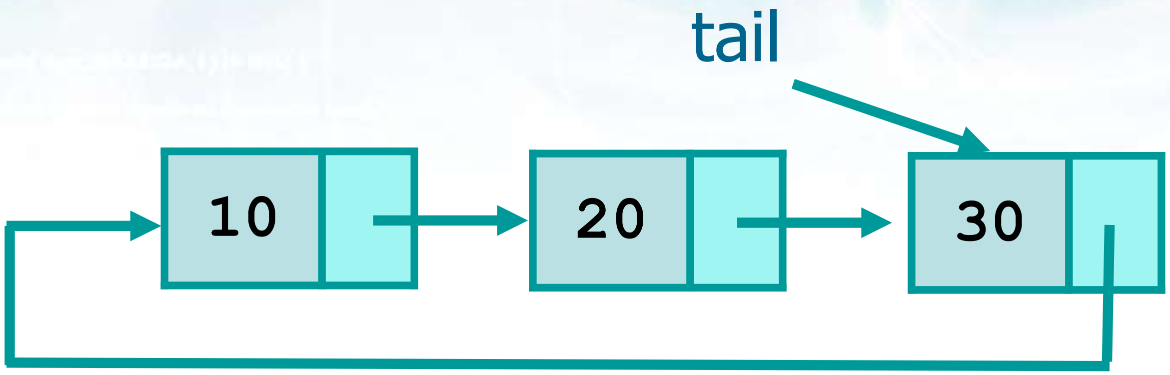
Implementation

```
         list_t *enqueue (list_t *tail, int val) {
           list_t *new;

Step 1:    new = new_element ();
           new->key = val;

           if (tail==NULL) {
             tail = new;
             tail->next = tail;
           } else {
Step 2:      new->next = tail->next;
Step 3:      tail->next = new;
Step 4:      tail = new;
           }

           return (tail);
         }
```

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

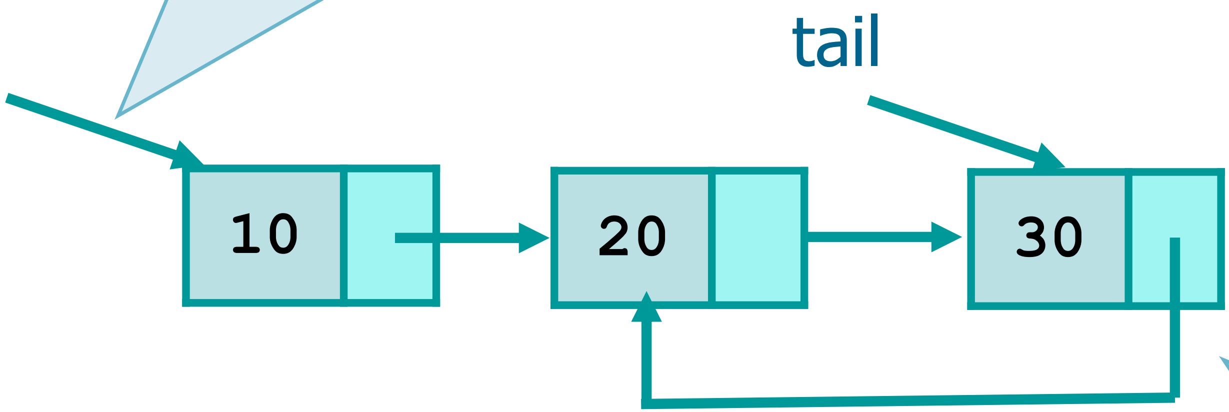


Dequeue



Step 1: old = tail->next;

Step 3: from key value, free the element

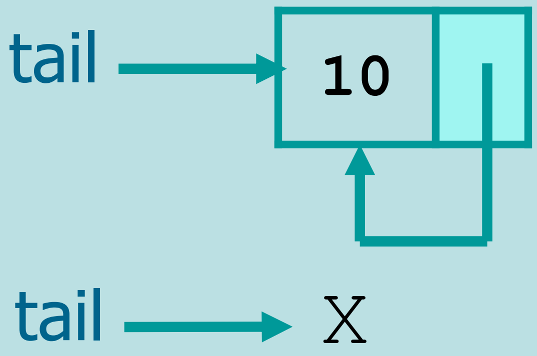


Step 2:
tail->next=old->next

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 {
Step 1:         old = tail->next;
                *val = old->key;
Step 2:         tail->next = old->next;
Step 3:         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

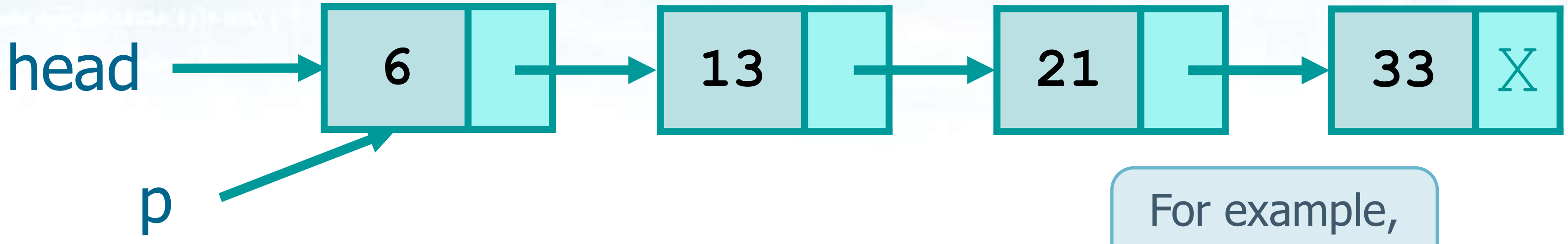
Type definition, initialization, and function calls

```
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



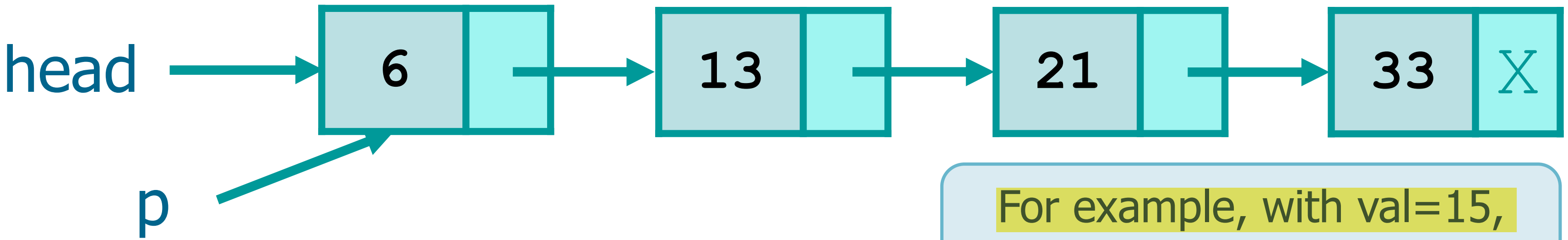
For example, val=15

```
void search (list_t *head, int val) {
  list_t *p;

  p=head;
  while (p!=NULL && p->val<val) {
    p = p->next;
  }
  if (p!=NULL && p->val==val) {
    ... found ...
  }

  return;
}
```

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



For example, with val=15, we stop with p on 21

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

p==NULL OR p->val>=val
Too late to insert !!!

Insertion

- The are several approaches to solve the problem
  - Use two pointers to individuate two consecutive elements
    - Move them along the list is 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

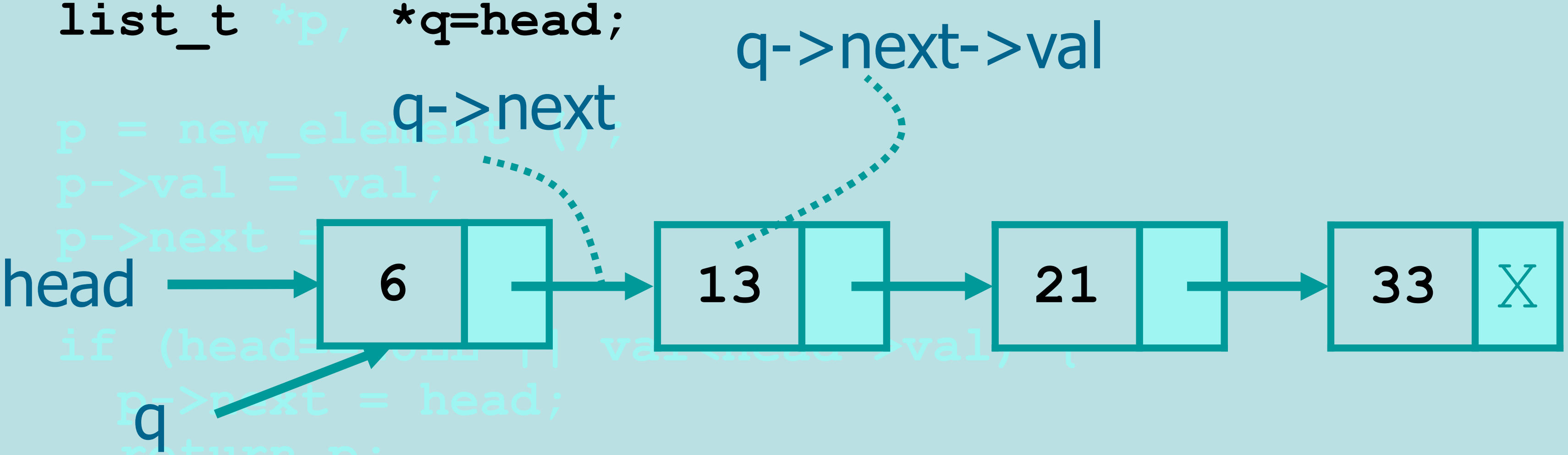
q->next is used to compare

q is used to insert

Implementation

Example: val=15

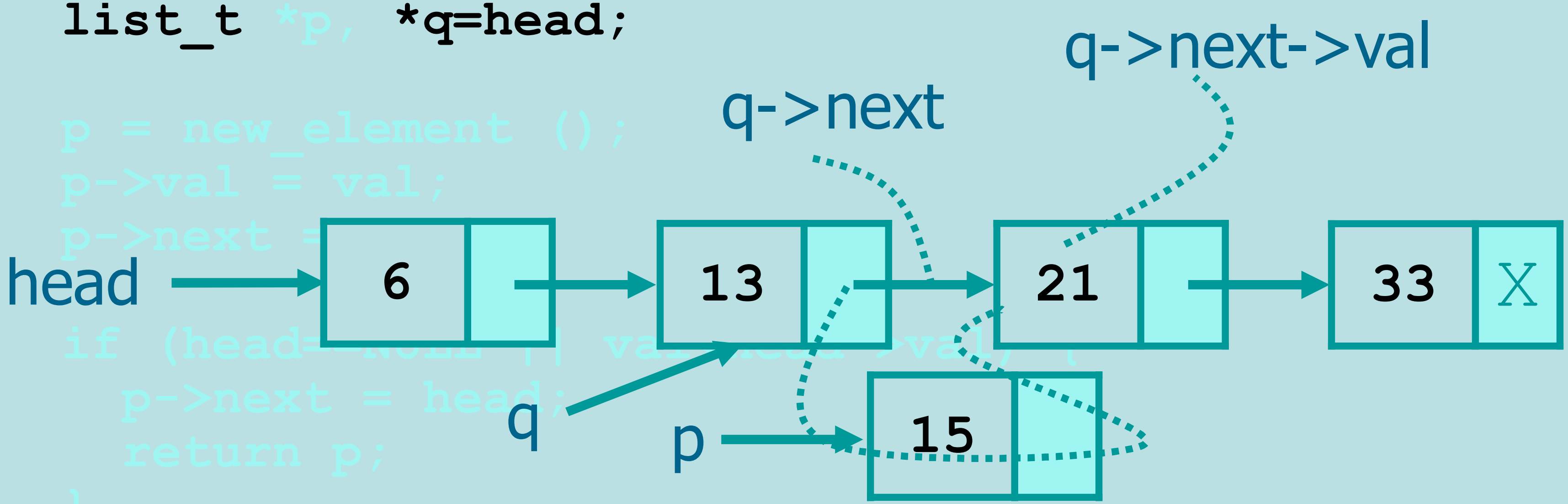
```
list_t *insert (list_t *head, int val) {
  list_t *p, *q=head;

  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;
}
```



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;
}
```



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
    - Emply 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;
  }
```

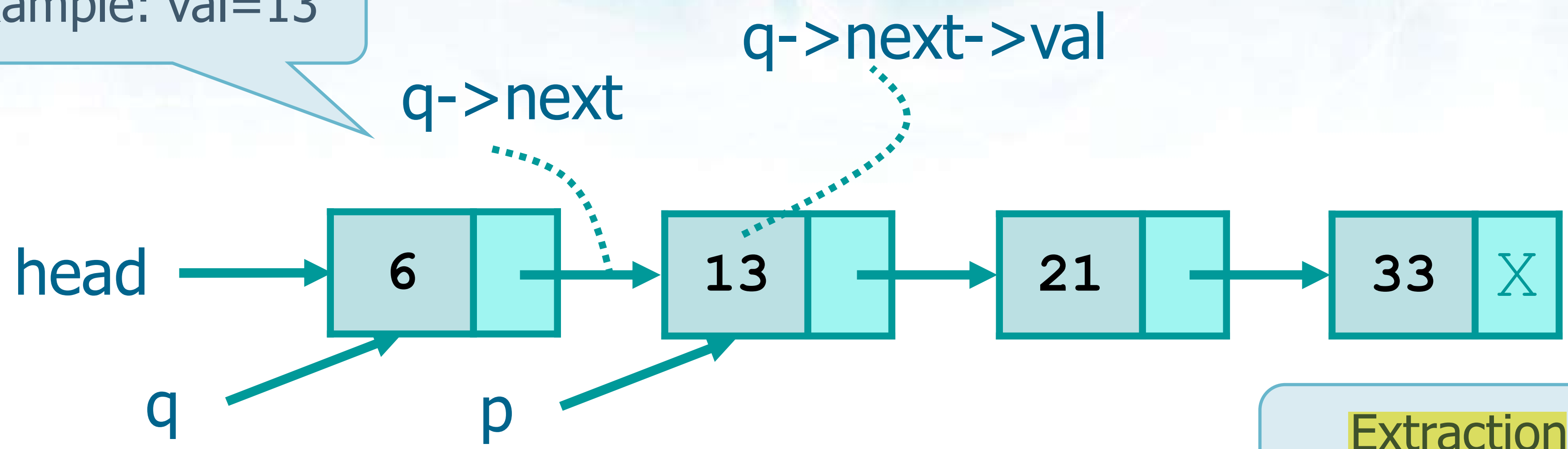
Empty list

Head extraction

Search
(using q->next)

Implementation

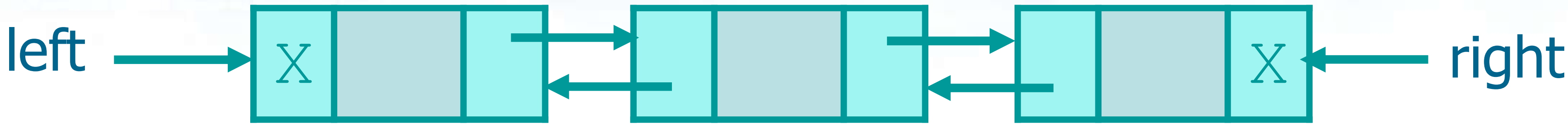
Example: val=13



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

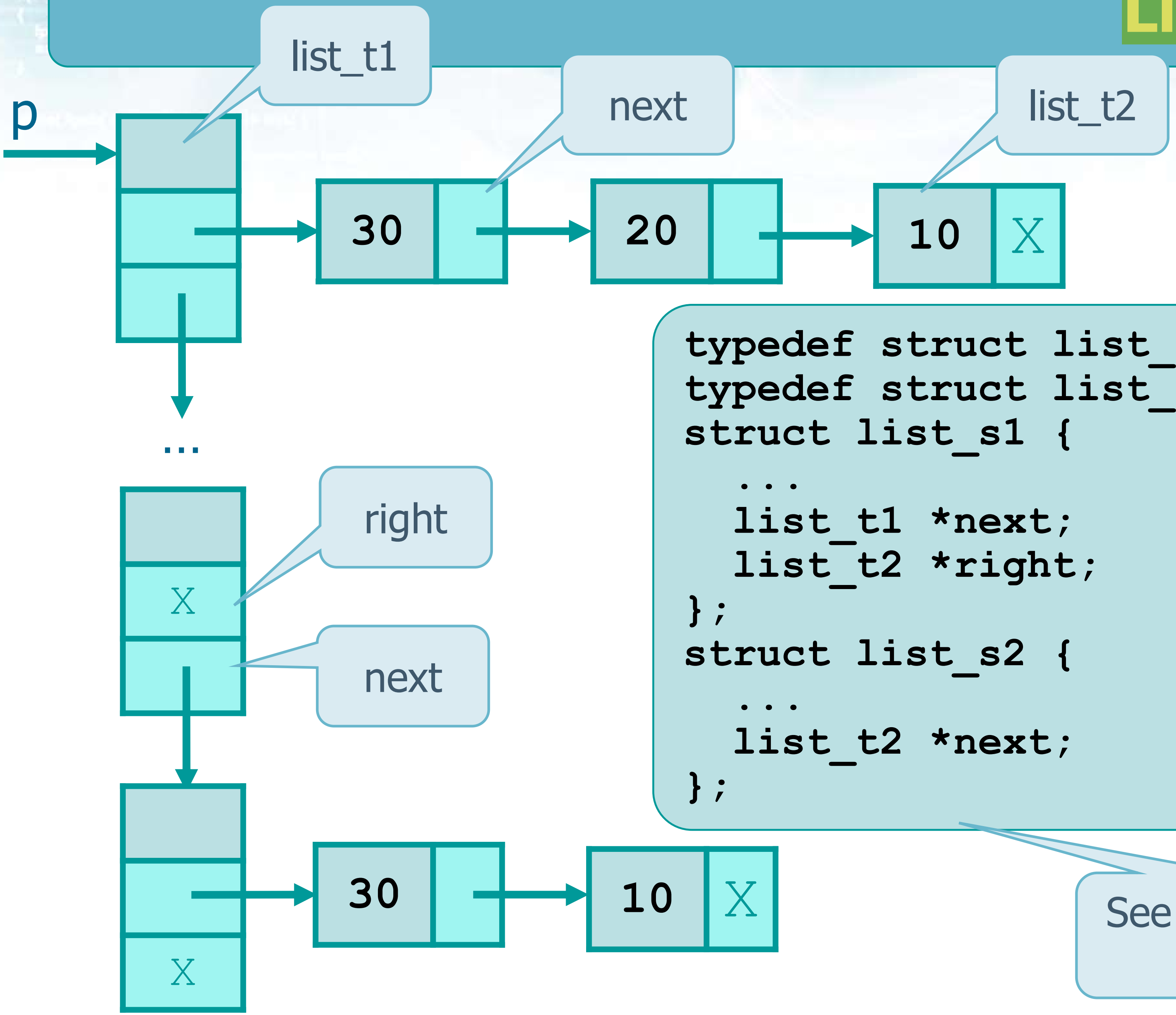



```
typedef struct list_s list_t;
struct list_s {
  ...
  list_t *left, *right;
};
```

- 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