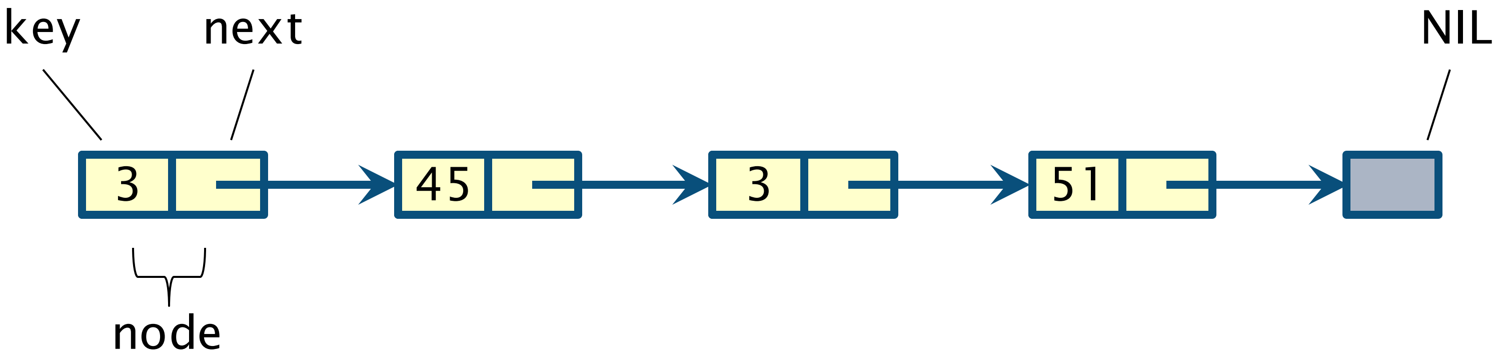
(SINGLY) LINKED LIST

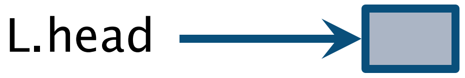
Singly linked list

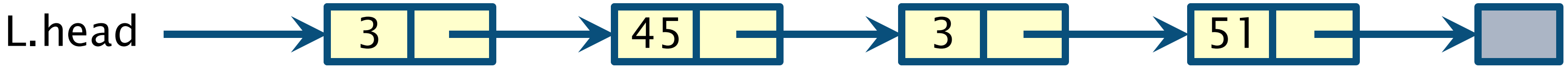
- A collection of objects, arranged in a linear order
  - The order in a linked list is (implicitly) determined by using a **pointer** in each object
- Each element (or node) x of a (singly) linked list L has:
  1. an attribute x.key, holding the “value” of the node; and
  2. a pointer attribute x.next
  - x.next points to *successor* of x in L
  - If x is the *last* element of L (called the tail), then x.next = NIL



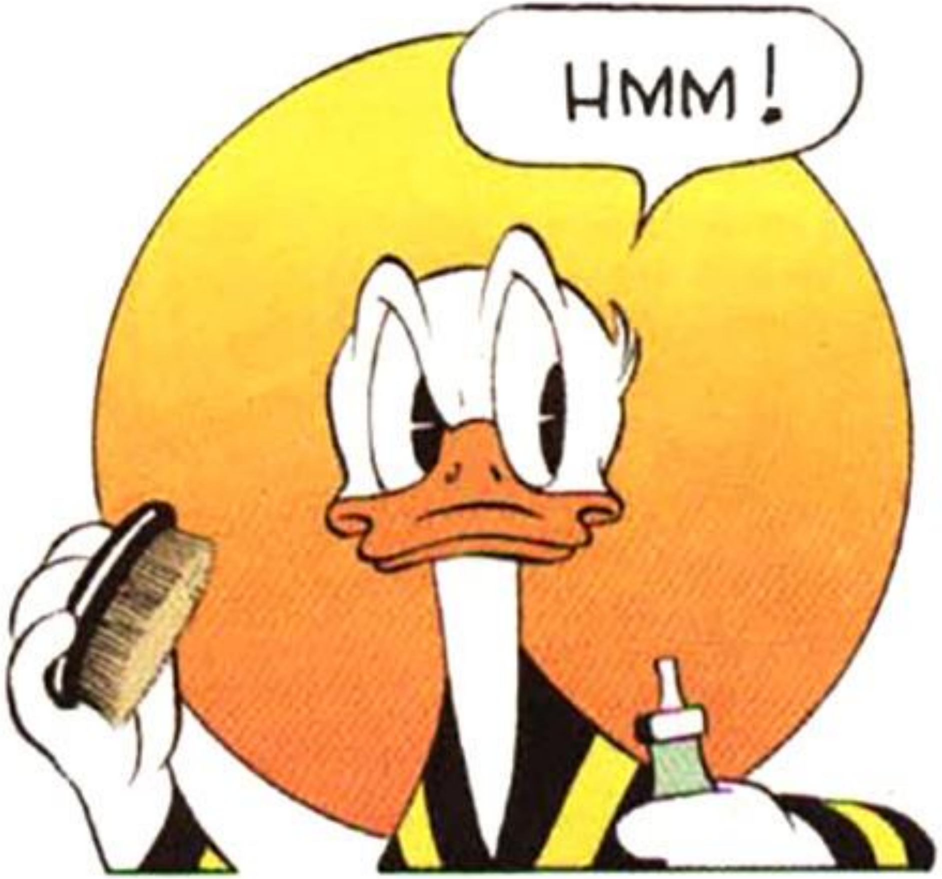
Singly linked list

- A collection of objects, arranged in a linear order
  - The order in a linked list is (implicitly) determined by using a **pointer** in each object
- Each element (or node) x of a (singly) linked list L has:
  1. an attribute x.key, holding the "value" of the node; and
  2. a pointer attribute x.next
     - x.next points to *successor* of x in L
     - If x is the *last* element of L (called the tail), then x.next = NIL
- An attribute L.head points to the *first* element of L
  - If the list is empty, then L.head = NIL





Algorithms *for* Data Structures

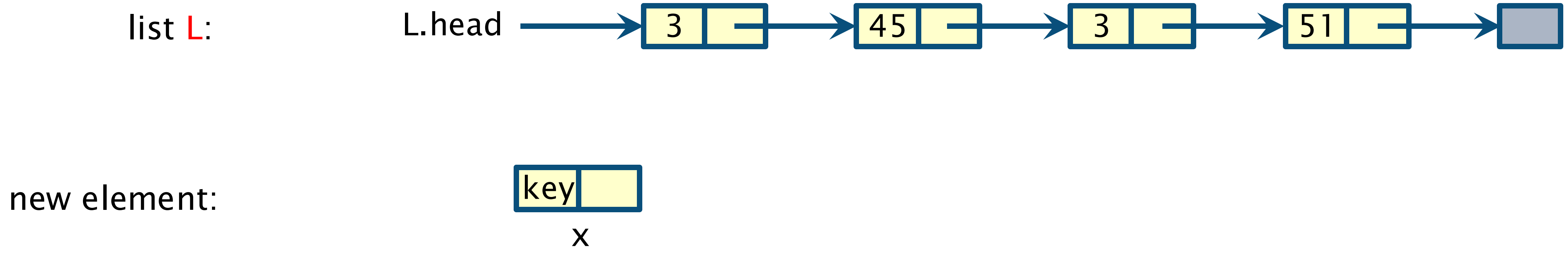


- Given a linked list data structure:
- How do we perform the following operations?
  - Insert element at the head of the list
  - Insert element at the tail of the list
  - Search for (the value of) an element
  - Delete an element
  - Etc.
- We need to design the *algorithms* for performing these operations
- That will allow us to *implement* these data structures in a given programming language

Insertion

- Insertion at the head
  - Allocate a new node with desired key
  - Update two pointers

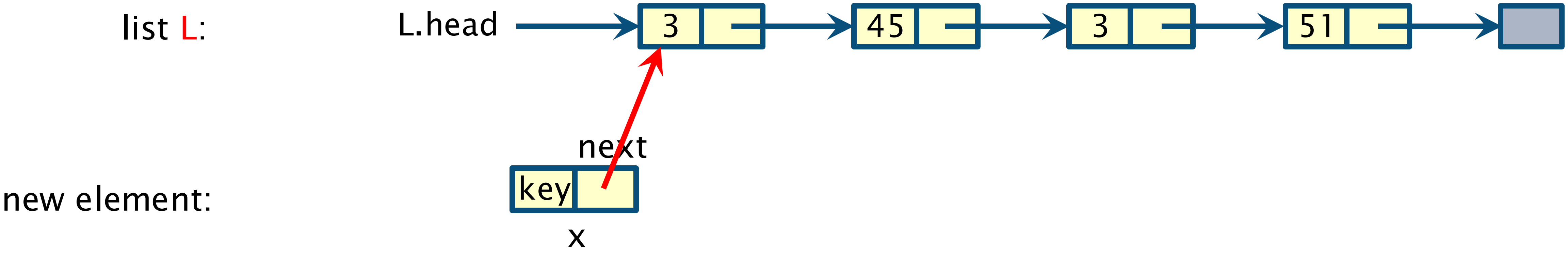
```
INSERT(L,x)
  x.next = L.head
  L.head = x
```



Insertion

- Insertion at the head
  - Allocate a new node with desired key
  - Update two pointers

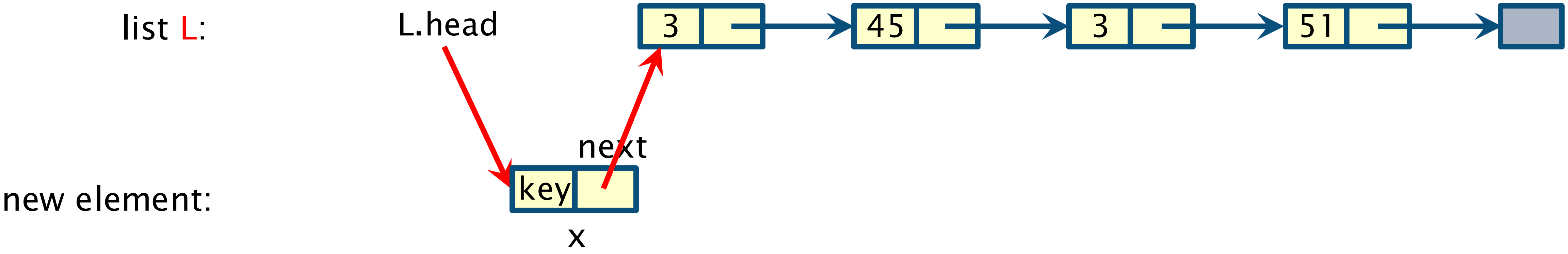
```
INSERT(L,x)
   x.next = L.head
   L.head = x
```



Insertion

- Insertion at the head
  - Allocate a new node
  - Update two pointers
- Complexity ?

```
INSERT(L,x)
  x.next = L.head
  L.head = x
```

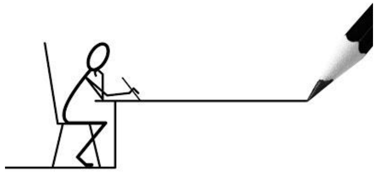


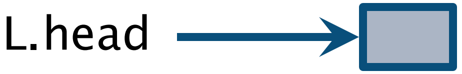
Insertion

- Insertion **at the head**
  - Allocate a new node
  - Update two pointers
- Complexity **O(1)**

```
INSERT(L,x)
  x.next := L.head
  L.head := x
```

- Example
  - Add nodes with keys 2 and 3 (in this order) to an empty list



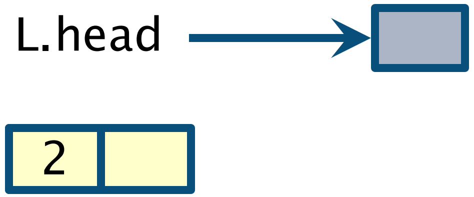


Insertion

- Insertion at the head
  - Allocate a new node
  - Update two pointers
- Complexity O(1)

```
INSERT(L,x)
  x.next := L.head
  L.head := x
```

- Example
  - Add nodes with keys 2 and 3 (in this order) to an empty list



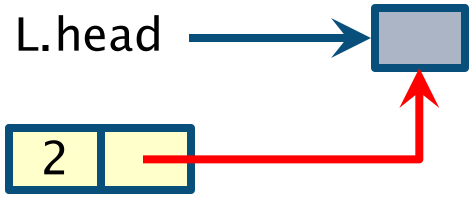
  - Create new node with key = 2

Insertion

- Insertion at the head
  - Allocate a new node
  - Update two pointers
- Complexity O(1)

```
INSERT(L,x)
  x.next := L.head
  L.head := x
```

- Example
  - Add nodes with keys 2 and 3 (in this order) to an empty list



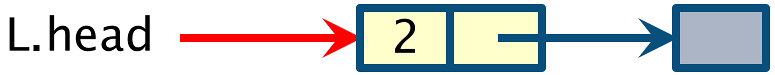
  - Update next

Insertion

- Insertion at the head
  - Allocate a new node
  - Update two pointers
- Complexity O(1)

```
INSERT(L,x)
  x.next := L.head
  L.head := x
```

- Example
  - Add nodes with keys 2 and 3 (in this order) to an empty list



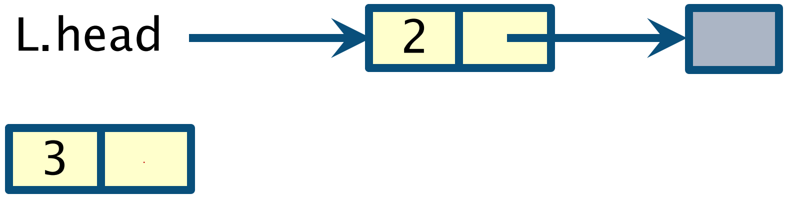
  - Update head

Insertion

- Insertion at the head
  - Allocate a new node
  - Update two pointers
- Complexity O(1)

```
INSERT(L,x)
   x.next := L.head
   L.head := x
```

- Example
  - Add nodes with keys 2 and 3 (in this order) to an empty list



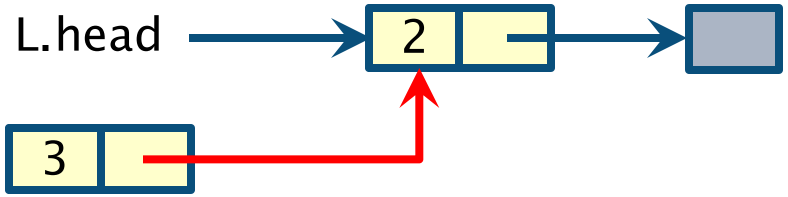
  - Create new node with key = 3

Insertion

- Insertion **at the head**
  - Allocate a new node
  - Update two pointers
- Complexity **O(1)**

```
INSERT(L,x)
  x.next := L.head
  L.head := x
```

- Example
  - Add nodes with keys 2 and 3 (in this order) to an empty list



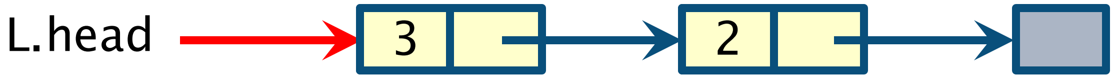
  - Update next

Insertion

- Insertion at the head
  - Allocate a new node
  - Update two pointers
- Complexity O(1)

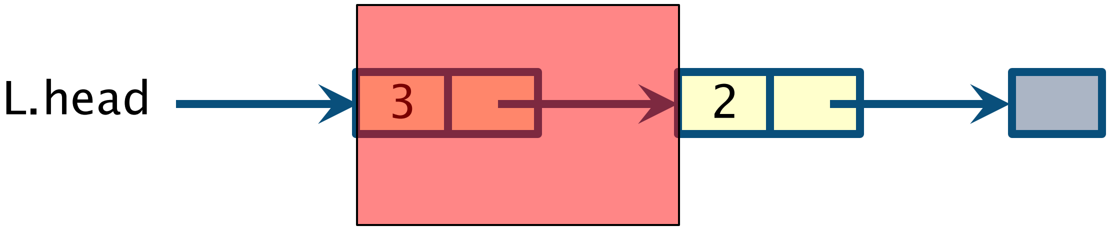
```
INSERT(L,x)
  x.next := L.head
  L.head := x
```

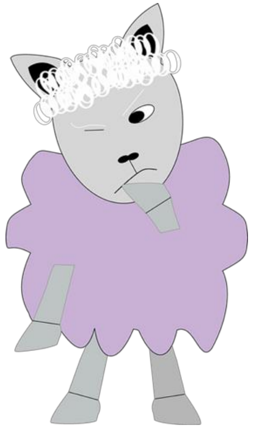
- Example
  - Add nodes with keys 2 and 3 (in this order) to an empty list



  - Update head

Deletion



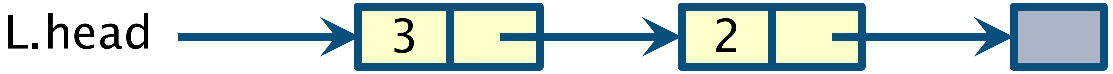


Can you perform the operation(s) needed for deletion?

Deletion

- Deletion at the head
  - Update `L.head`
    - Be careful about “corner cases”; e.g. what if list is empty?
  - Deallocate memory of node being deleted
- Deallocation is performed by the *garbage collector* in Python
- Complexity O(1)

```
DELETE-HEAD(L)
  if L.head != NIL
    L.head := L.head.next
```



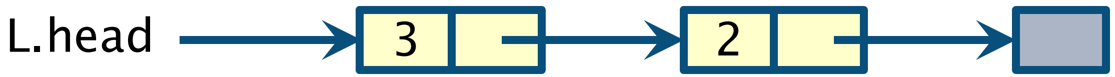
Deletion

- Deletion at the head
  - Update L.head
  - Deallocate memory of node being deleted

```
DELETE-HEAD(L)
  if L.head != NIL
    L.head := L.head.next
```

- Complexity O(1)

- Example
  - Delete (at the head) *three* times on the list below:



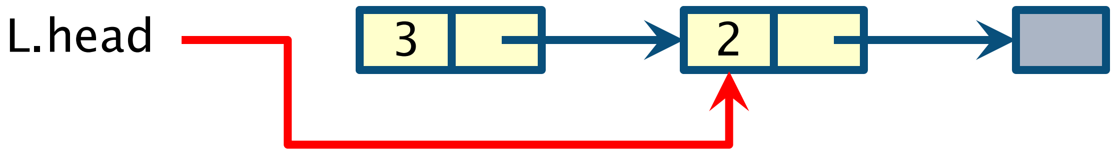
Deletion

- **Deletion at the head**
  - Update L.head
  - Deallocate memory of node being deleted

```
DELETE-HEAD(L)
  if L.head != NIL
    L.head := L.head.next
```

- **Complexity O(1)**

- **Example**
  - Delete (at the head) *three* times on the list below:



1) L.head is updated

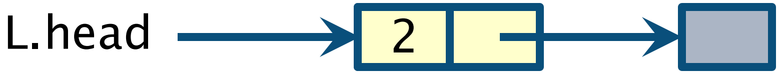
Deletion

- Deletion at the head
  - Update L.head
  - Deallocate memory of node being deleted

```
DELETE-HEAD(L)
  if L.head != NIL
    L.head := L.head.next
```

- Complexity O(1)

- Example
  - Delete (at the head) *three* times on the list below:



1) 3 is garbage collected

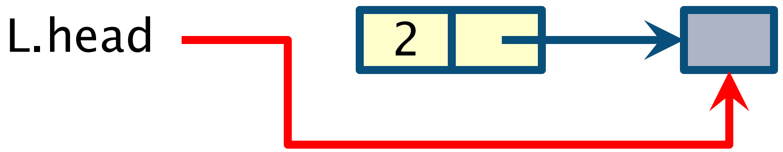
Deletion

- Deletion at the head
  - Update L.head
  - Deallocate memory of node being deleted

```
DELETE-HEAD(L)
  if L.head != NIL
    L.head := L.head.next
```

- Complexity O(1)

- Example
  - Delete *three* times on the list below:



2) L.head is updated

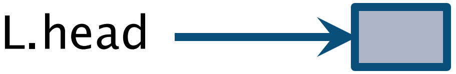
Deletion

- Deletion at the head
  - Update L.head
  - Deallocate memory of node being deleted

```
DELETE-HEAD(L)
  if L.head != NIL
    L.head := L.head.next
```

- Complexity O(1)

- Example
  - Delete *three* times on the list below:



3) 2 is garbage collected

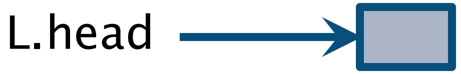
Deletion

- Deletion at the head
  - Update L.head
  - Deallocate memory of node being deleted

```
DELETE-HEAD(L)
  if L.head != NIL
    L.head := L.head.next
```

- Complexity O(1)

- Example
  - Delete *three* times on the list below:



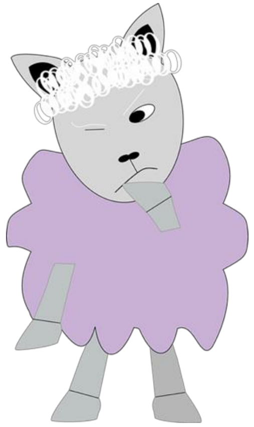
3) L.head = NIL

Search

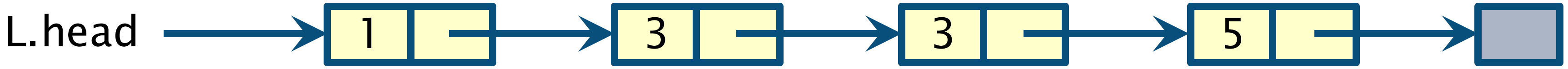
This Photo by Unknown Author is licensed under CC BY-SA

Search

- Given value k, find the first element with key k in list L by a simple *linear search*
  - If such an object exists, return a pointer to it
  - If not, then return NIL



- Example
  - Find k=3 in the list below:



Can you perform the operations needed for searching?

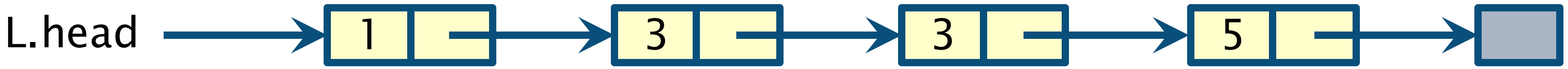
Search

- Find the first element with key k in list L by a simple linear search
  - If found, return a pointer to this element
  - If no object with key k appears in the list, then return NIL

```
SEARCH(L,k)
  i := L.head
  while i != NIL and i.key != k
    i := i.next
  return i
```

- Complexity O(n)

- Example
  - Find k=3 in the list below



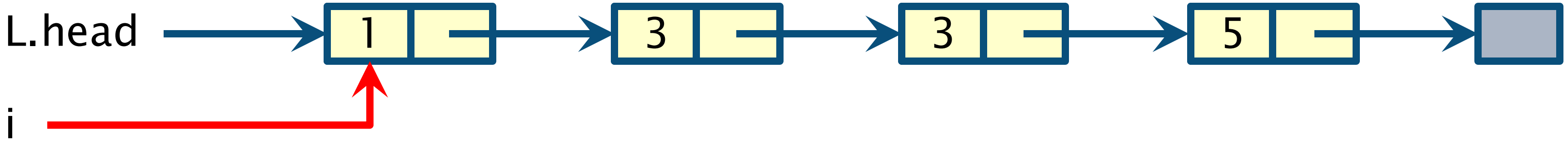
Search

- Find the first element with key k in list L by a simple linear search
  - If found, return a pointer to this element
  - If no object with key k appears in the list, then return NIL

- Complexity O(n)

```
SEARCH(L,k)
  i := L.head
  while i != NIL and i.key != k
    i := i.next
  return i
```

- Example
  - Find k=3 in the list below



  - Initialize cursor i

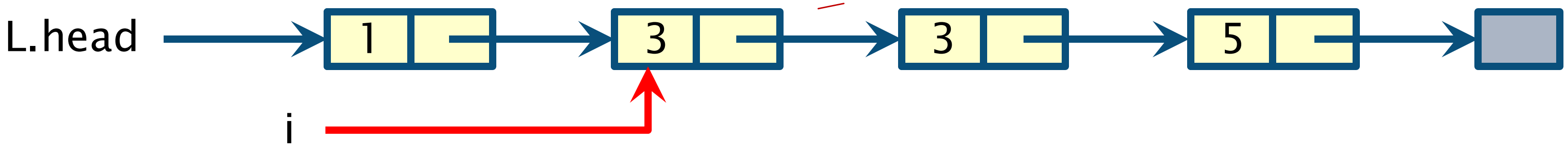
Search

- Find the first element with key k in list L by a simple linear search
  - If found, return a pointer to this element
  - If no object with key k appears in the list, then return NIL

- Complexity O(n)

```
SEARCH(L,k)
  i := L.head
  while i != NIL and i.key != k
    i := i.next
  return i
```

- Example
  - Find k=3 in the list below



  - 1!= 3, so we update cursor i

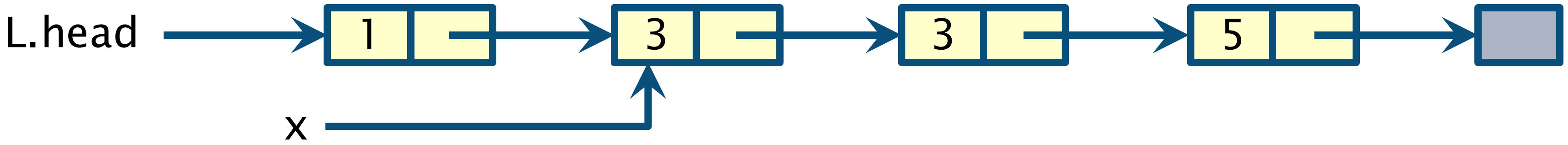
Search

- **Find the first element with key k in list L by a simple linear search**
  - If found, return a pointer to this element
  - If no object with key k appears in the list, then return NIL

- **Complexity O(n)**

```
SEARCH(L,k)
  i := L.head
  while i != NIL and i.key != k
    i := i.next
  return i
```

- **Example**
  - Find k=3 in the list below



  - 3=3, so we return cursor i

Insertion at the tail

Why?

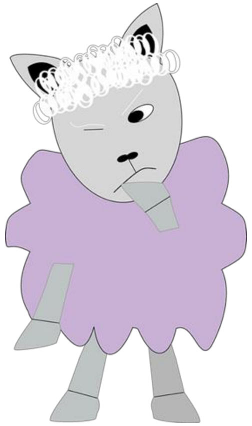
To implement *FIFO* data structures, like *queues:*

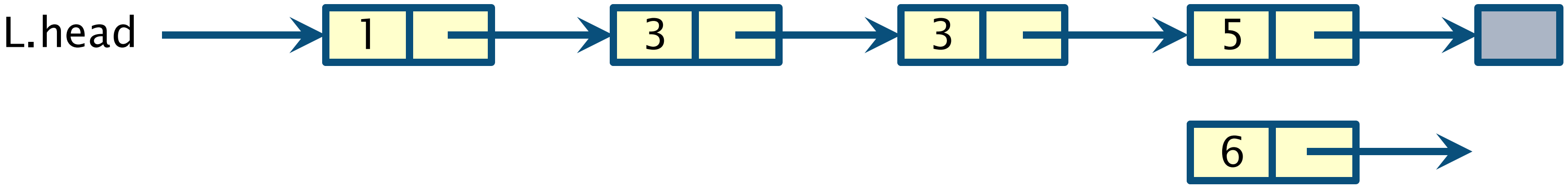
- Insert (write) at one end, delete (read) at the other

Insertion at the tail

- We saw that insertion at the head is O(1)
  - What about insertion at the tail?

- Example
  - Insert “6” at the *tail* of the following linked list





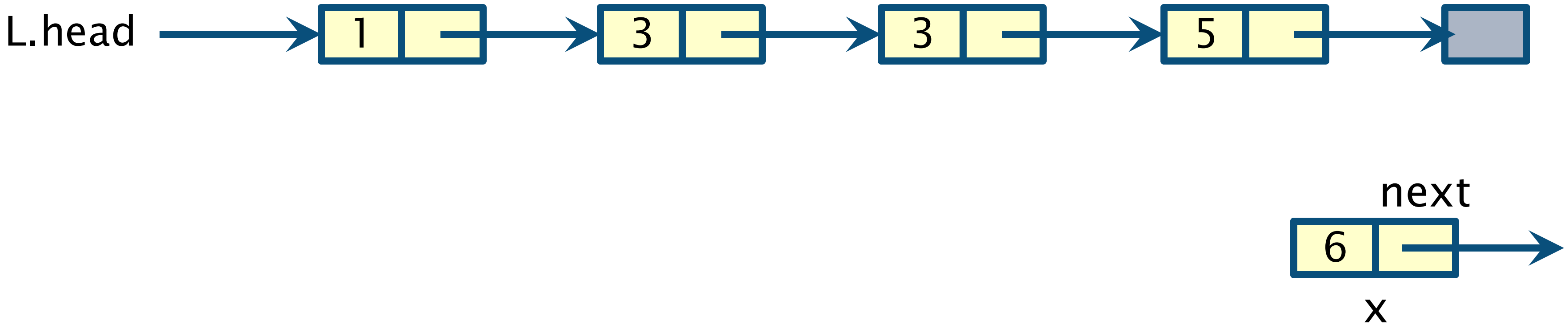
Can you perform the operations needed for Insertion at tail? Complexity?

Insertion at the tail

- We saw that insertion **at the head** is **O(1)**

- Insertion **at the tail** requires to scan the entire list
  - Complexity **O(n)**

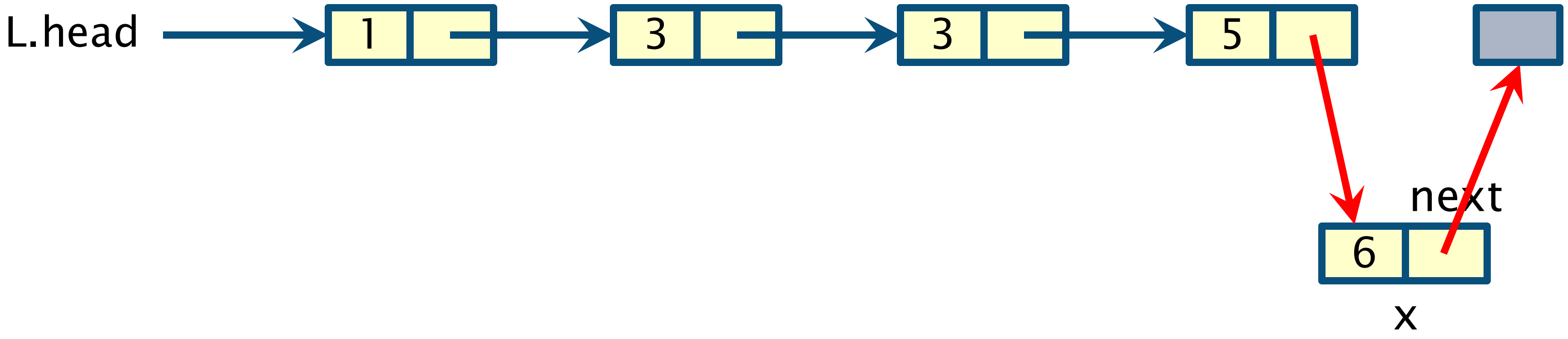
```
INSERT-TAIL(L,x)
  if L.head = NIL
    INSERT(L,x)
  else
    i := L.head
    while i.next != NIL
      i := i.next
    x.next := NIL
    i.next := x
```



Insertion at the tail

- We saw that insertion at the head is O(1)
- Insertion at the tail requires to scan the entire list
    - Complexity O(n)
- This is a significant drawback of using linked lists to implement queues

```
INSERT-TAIL(L,x)
  if L.head = NIL
    INSERT(L,x)
  else
    i := L.head
    while i.next != NIL
      i := i.next
    x.next := NIL
    i.next := x
```



Insert at head vs insert at tail

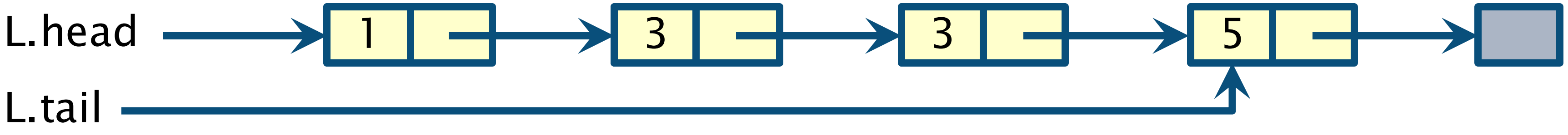
- Insert at head: O(1)
- Insert at tail: O(n)

Is there a way to make *both* operations run at O(1)?

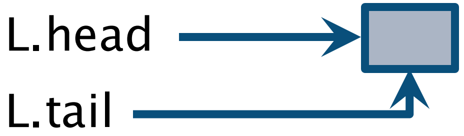
(We don't mind incurring a small cost in *space* complexity…)

Tail pointer

- We extend the definition of a (singly) linked list L to include an attribute L.tail pointing to its last element



- Empty lists: L.head = L.tail = NIL



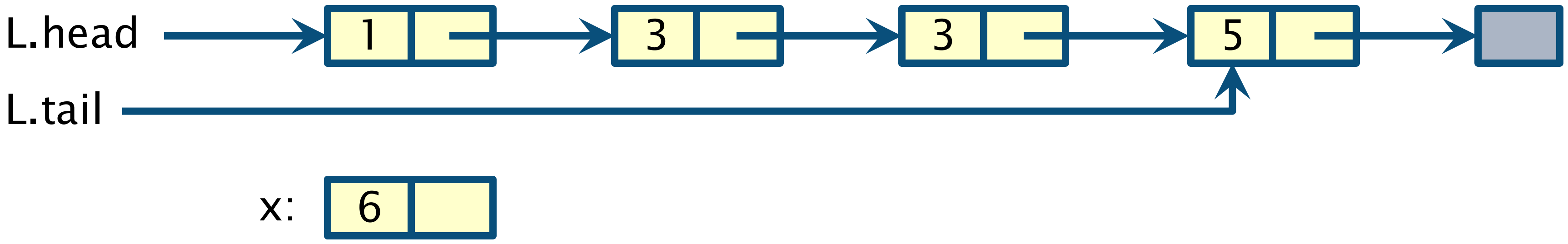
- INSERT and DELETE have to be adapted accordingly to update L.tail when needed

Insertion at the tail, *with* tail pointer

- Can be performed in O(1) time

```
INSERT-TAIL(L,x)
  x.next := NIL
  if L.tail = NIL
    L.head := x
  else
    L.tail.next := x
  L.tail := x
```

- Example
  - Insert “6” at the tail of the list below

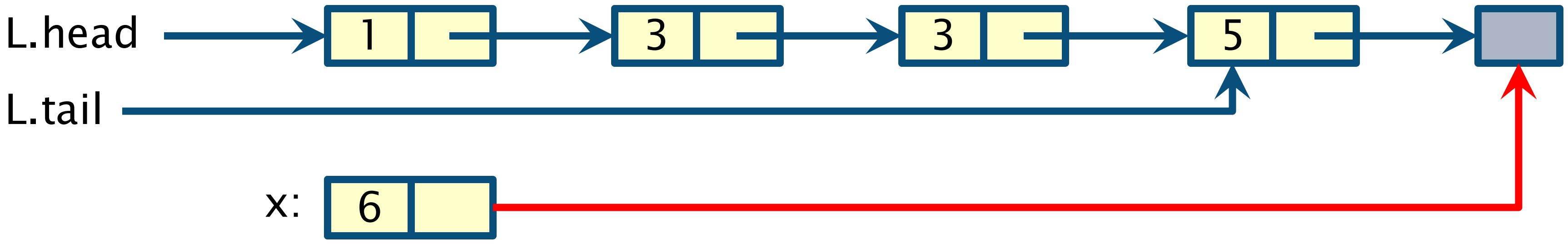


Insertion at the tail, *with* tail pointer

- Can be performed in O(1) time

```
INSERT-TAIL(L,x)
  x.next := NIL
  if L.tail = NIL
    L.head := x
  else
    L.tail.next := x
  L.tail := x
```

- Example
  - Insert "6" at the tail of the list below



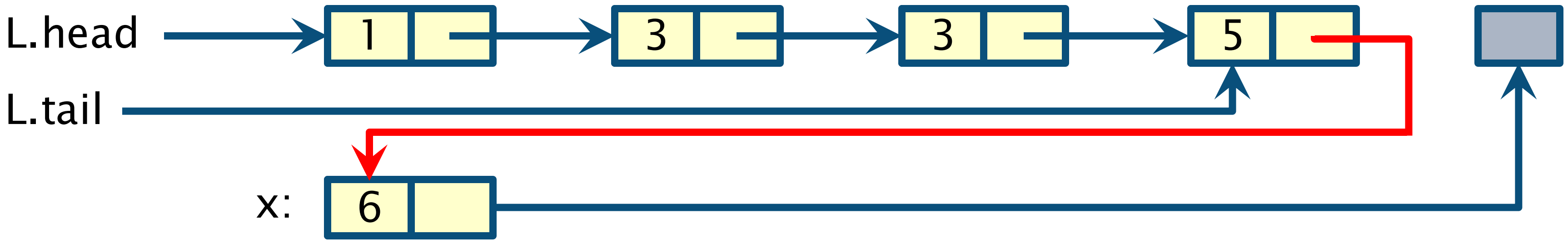
  - Update x.next

Insertion at the tail with tail pointer

- Can be performed in **constant** time

```
INSERT-TAIL(L,x)
  x.next := NIL
  if L.tail = NIL
    L.head := x
  else
    L.tail.next := x
  L.tail := x
```

- Example
  - Insert “6” at the tail of the list below



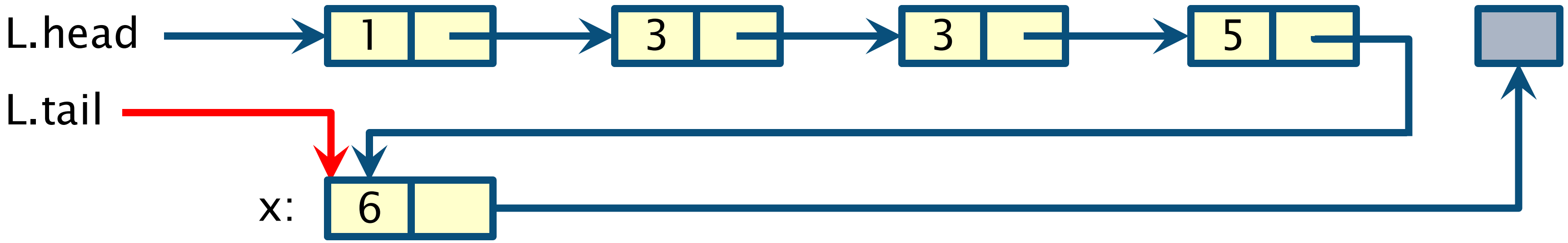
  - Update L.tail.next

Insertion at the tail with tail pointer

- Can be performed in **constant** time

```
INSERT-TAIL(L,x)
  x.next := NIL
  if L.tail = NIL
    L.head := x
  else
    L.tail.next := x
  L.tail := x
```

- Example
  - Insert x.key = 6 at the tail of the list below



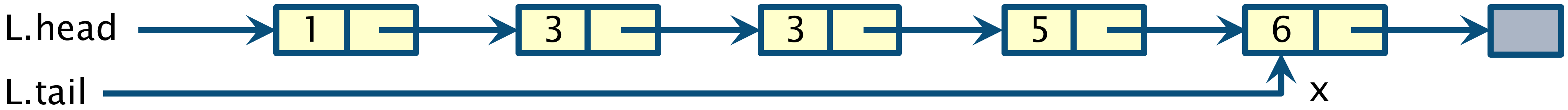
  - Update L.tail

Insertion at the tail with tail pointer

- Can be performed in **constant** time

```
INSERT-TAIL(L,x)
  x.next := NIL
  if L.tail = NIL
    L.head := x
  else
    L.tail.next := x
  L.tail := x
```

- Example
  - Insert x.key = 6 at the tail of the list below

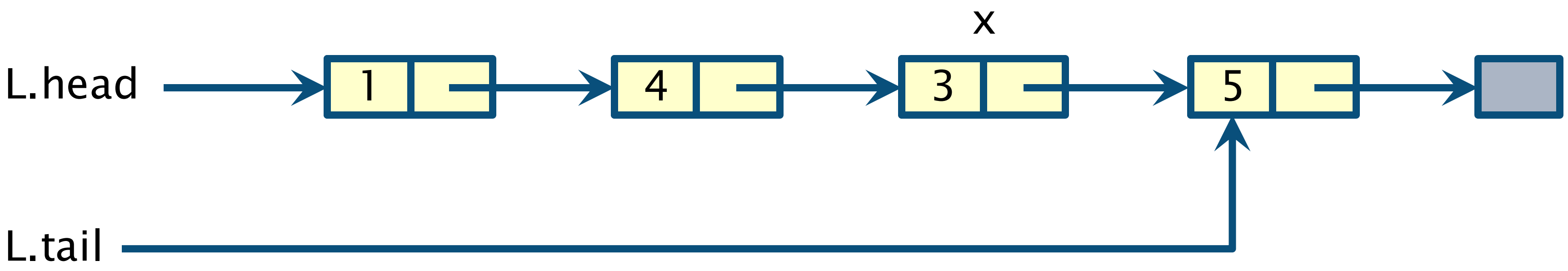


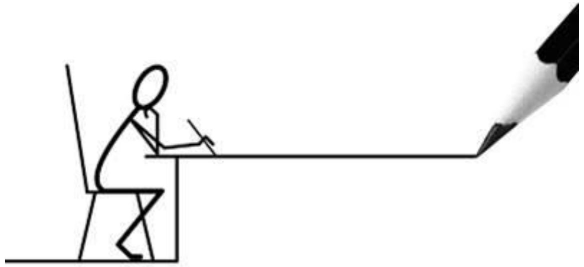
  - Termination

Deletion with tail pointer

- Remove an element x from a linked list L

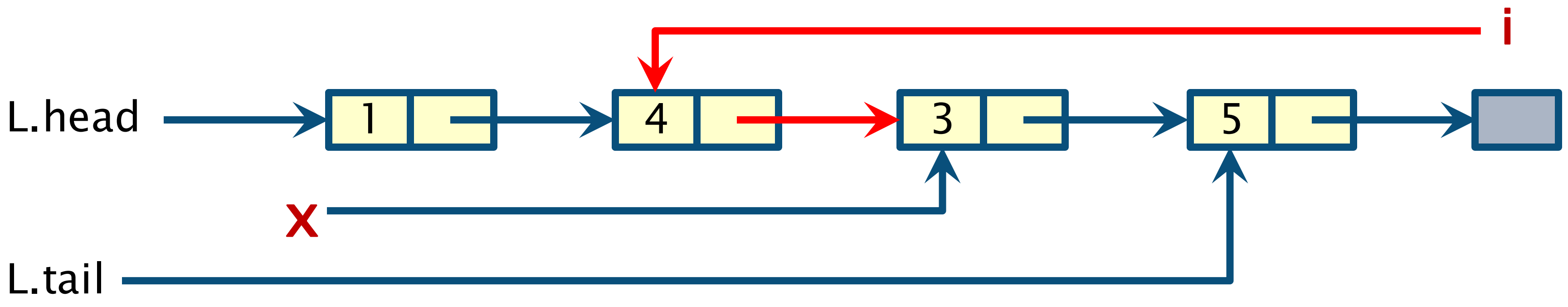
- What steps will you follow to delete node x from the list below?





Deletion with tail pointer

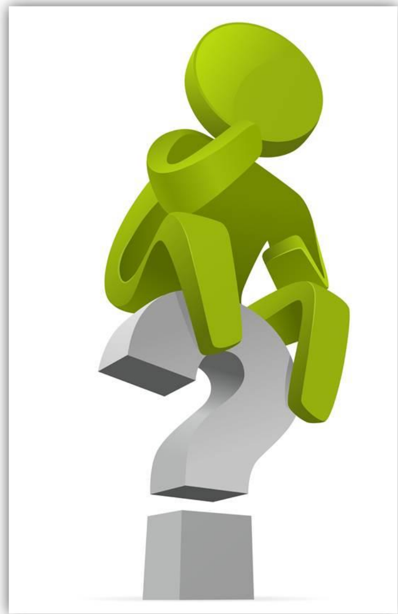
- Remove an element x from a linked list L
  - Note: pointer to node x must be retrieved first (for instance by calling SEARCH for a given key value)
  - Assume that we have such a pointer



- We also need a pointer i to the predecessor of x in order to update i.next := x.next
  - Scan the list again and return i when i.next = x
    - O(n) complexity

Singly linked lists: main disadvantage

List can only be traversed in *one* fixed direction (head → tail)

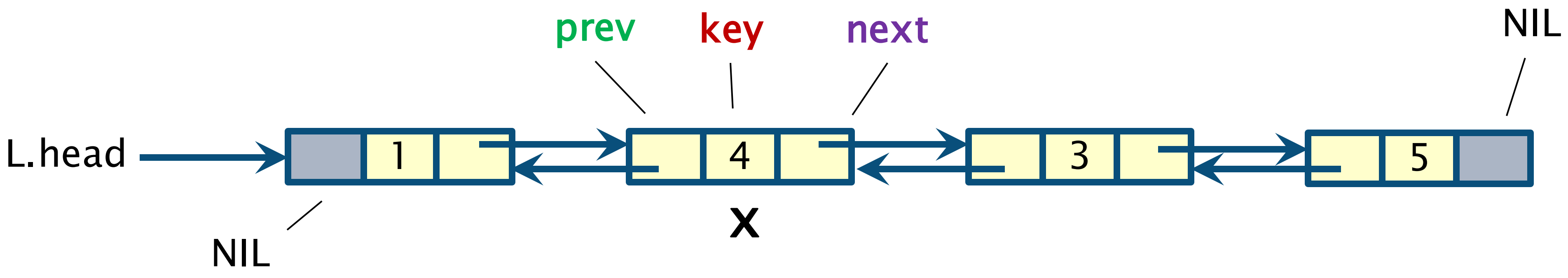


How can we overcome this?

DOUBLY LINKED LISTS

Doubly linked lists

- Extend definition of singly linked lists so that each node has an additional pointer attribute prev
  - Given a node x, x.prev points to the previous node in the list
  - If x.prev = NIL, x has no predecessor and is therefore the head of the list

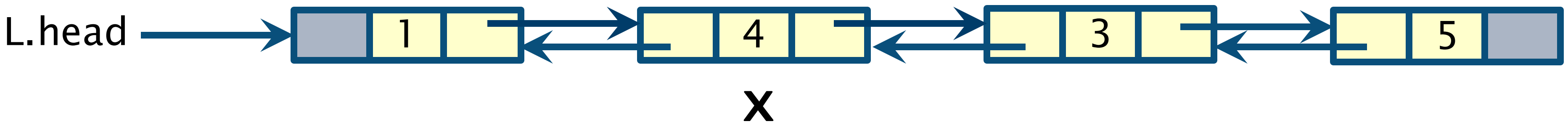


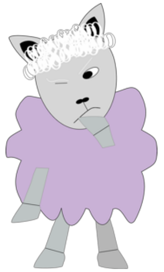
- Pros: key operations are simpler to implement and more efficiennt
- Cons: memory overhead → O(n) for maintaining the prev pointers

Deletion in doubly linked lists

- Can be performed in constant time
  - ➢ We don’t need to traverse the list anymore!

- Example
  - – Delete x from the list below





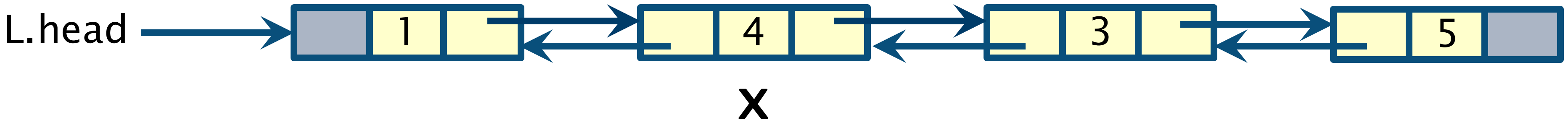
– Important: Think about *corner cases* (x might be the head/tail element)

Deletion in doubly linked lists

- Can be performed in constant time

```
DELETE(L,x)
  if x.prev != NIL
    x.prev.next := x.next
  else
    L.head:= x.next
  if x.next != NIL
    x.next.prev := x.prev
```

- Example
  - Delete x from the list below

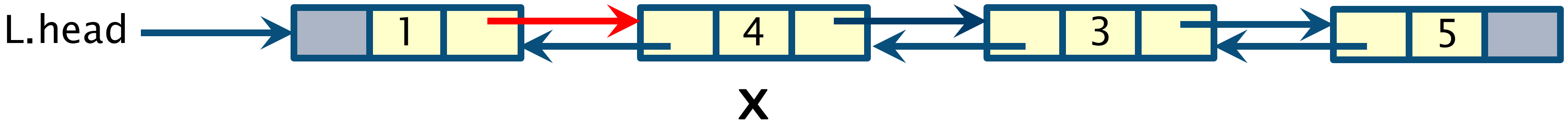


Deletion in doubly linked lists

- Can be performed in constant time

```
DELETE(L,x)
  if x.prev != NIL
    x.prev.next := x.next
  else
    L.head:= x.next
  if x.next != NIL
    x.next.prev := x.prev
```

- Example
  - Delete x from the list below



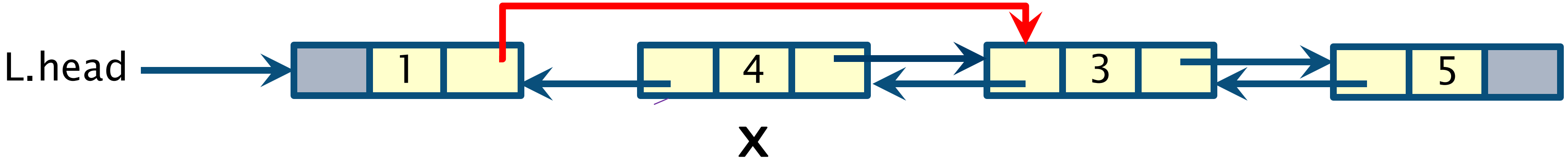
  - Update x.prev.next

Deletion of an element in doubly linked lists

- Can be performed in constant time

```
DELETE(L,x)
  if x.prev != NIL
    x.prev.next := x.next
  else
    L.head:= x.next
  if x.next != NIL
    x.next.prev := x.prev
```

- Example
  - Delete x from the list below



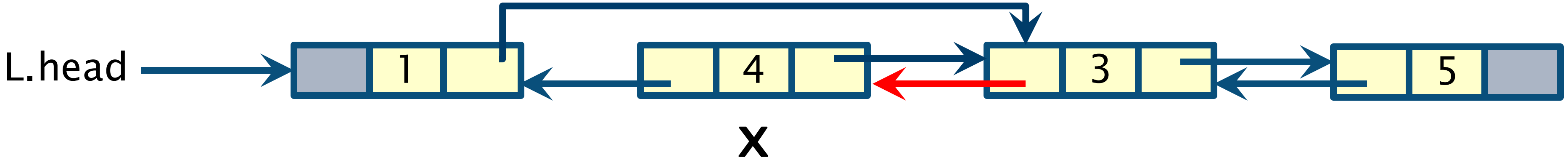
  - Update x.prev.next

Deletion of an element in doubly linked lists

- Can be performed in constant time

```
DELETE(L,x)
  if x.prev != NIL
    x.prev.next := x.next
  else
    L.head:= x.next
  if x.next != NIL
    x.next.prev := x.prev
```

- Example
  - Delete x from the list below



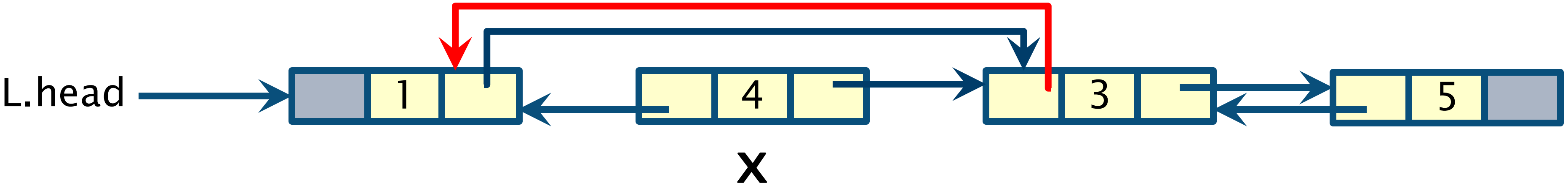
  - Update x.next.prev

Deletion of an element in doubly linked lists

- Can be performed in constant time

```
DELETE(L,x)
  if x.prev != NIL
    x.prev.next := x.next
  else
    L.head:= x.next
  if x.next != NIL
    x.next.prev := x.prev
```

- Example
  - Delete x from the list below



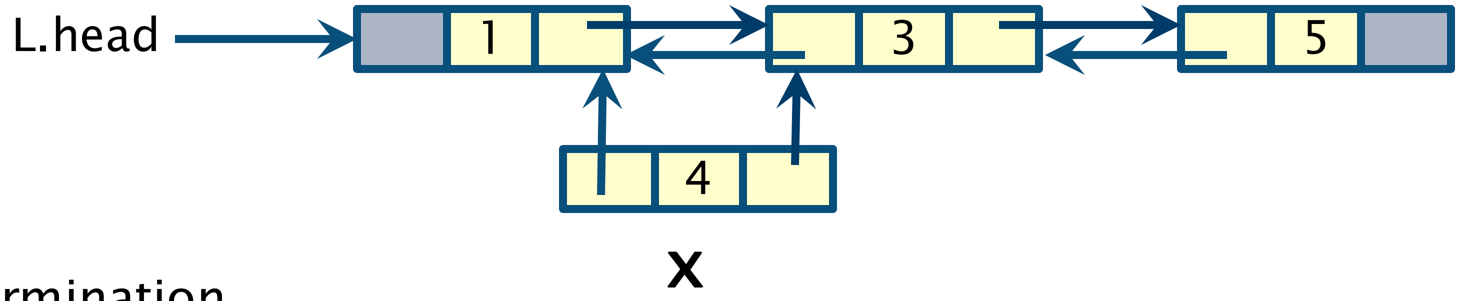
  - Update x.next.prev

Deletion of an element in doubly linked lists

- Can be performed in **constant** time

```
DELETE(L,x)
  if x.prev != NIL
    x.prev.next := x.next
  else
    L.head:= x.next
  if x.next != NIL
    x.next.prev := x.prev
```

- Example
  - Delete x from the list below



  - Termination

Linked List operations we have reviewed

1. Singly Linked List – Insertion at the head
2. Singly Linked List – Deletion at the head
3. Singly Linked List – Search for a given key
4. Singly Linked List – Insertion at the tail
5. Singly Linked List with tail pointer – Insertion at the tail
6. Doubly Linked List – Deletion