

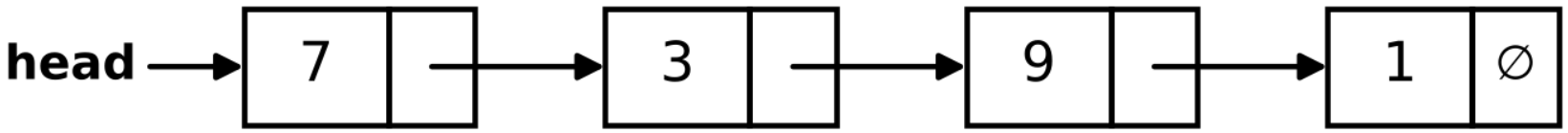
Course Algorithms and Data Structures · Lecture 6

Linked Lists

Textbook: A. Levitin, Section 1.4

A chain of nodes

Like a treasure hunt: each note tells you something, and where the next note is.



- Each **node** stores **data** and **next**, a link to the next node
- **head** points to the first node
- The last link is **null** (∅). An empty list: head = null

Traversal

```
ALGORITHM PrintList(head)
    p ← head
    while p ≠ null do
        print p.data
        p ← p.next
```

p ← p.next is the list version of i ← i + 1

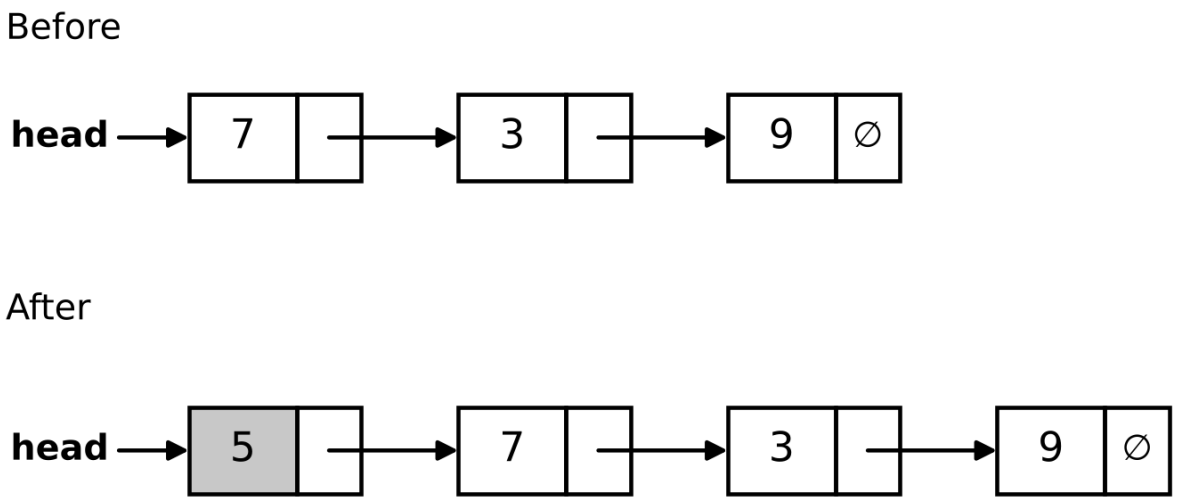
Search

```
ALGORITHM ListSearch(head, K)
    p ← head
    while p ≠ null and p.data ≠ K do
        p ← p.next
    return p
```

Sequential search again: O(n). **No direct access** to the i-th element: getting there costs O(i). No binary search.

Insert at the front: O(1)

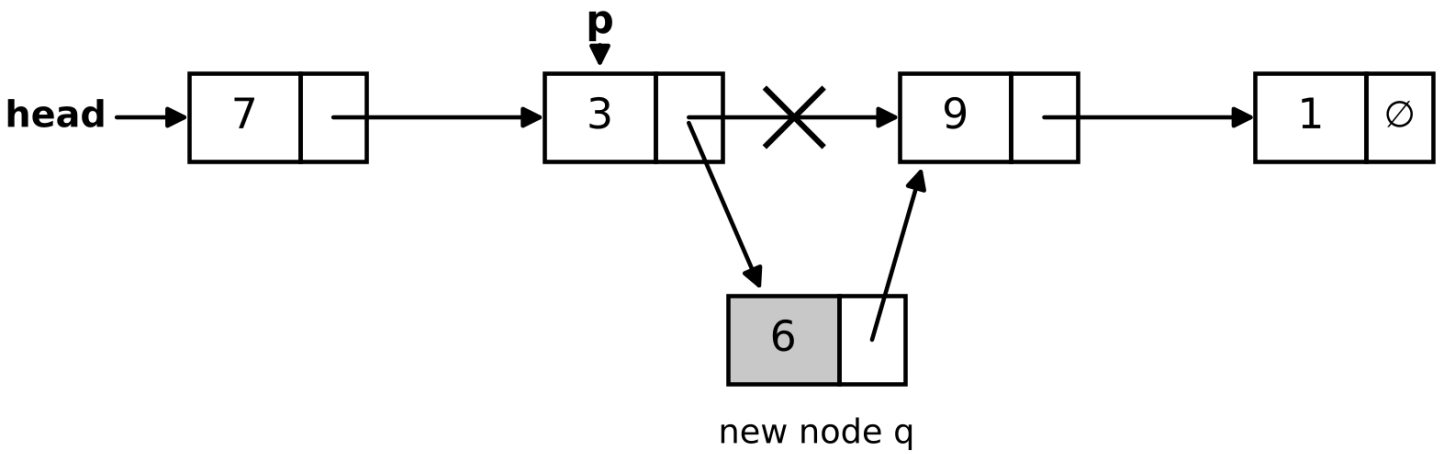
```
q ← new node
q.data ← x
q.next ← head
head ← q
```



Nothing is shifted. In an array, inserting at the front moves all n elements.

Insert after node p: O(1)

```
ALGORITHM InsertAfter(p, x)
    q ← new node;  q.data ← x
    q.next ← p.next     //1. new node points to p's successor
    p.next ← q          //2. p points to the new node
```



The order matters!

Right

q.next ← p.next
p.next ← q

Wrong

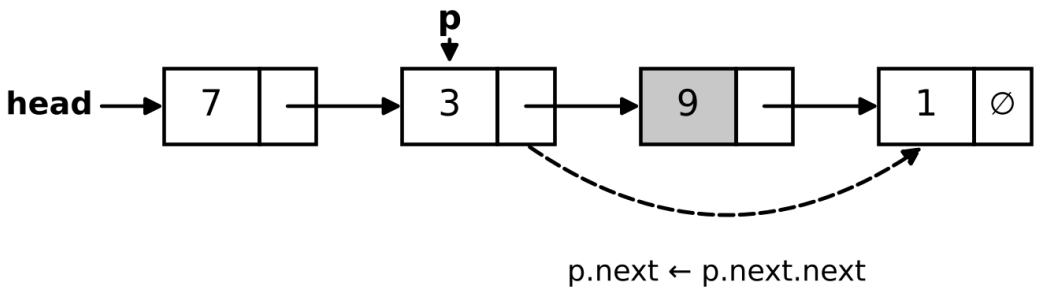
p.next ← q
q.next ← p.next (the rest of the list is lost)

Always connect the new node first, then redirect the old link.

Deletion: skip the node

```
ALGORITHM DeleteFront(head)
    if head ≠ null
        head ← head.next

ALGORITHM DeleteAfter(p)
    if p.next ≠ null
        p.next ← p.next.next
```



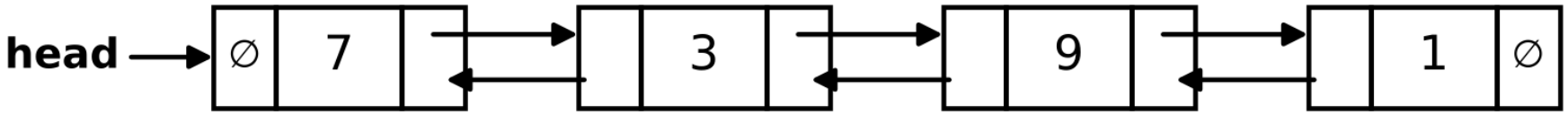
Both O(1). But we need the node **before** the one we delete.

Delete by value

```
ALGORITHM DeleteValue(head, K)
    if head = null return head        //empty list
    if head.data = K return head.next //delete the first node
    p ← head
    while p.next ≠ null and p.next.data ≠ K do
        p ← p.next
    if p.next ≠ null
        p.next ← p.next.next
    return head
```

Special cases: the empty list and the first node.

Doubly linked lists




```
ALGORITHM DeleteNode(p)     //p is in the middle
    p.prev.next ← p.next
    p.next.prev ← p.prev
```

- **Gain:** walk both ways; delete a node directly in O(1)
- **Cost:** an extra link in every node; more links to update

Linked list or array?

Operation	Array	Linked list
Access the i-th element	O(1)	O(n)
Search	O(n), O(log n) if sorted	O(n)
Insert / delete at the front	O(n)	O(1)
Insert / delete in the middle (place known)	O(n)	O(1)
Insert at the end	O(1)	O(1) with a tail pointer
Extra memory	none	a link per node

Common mistakes

- **Losing the list** by changing links in the wrong order
- **Following a null link**: check p ≠ null before p.next
- Forgetting **the empty list** and **changes at the head**
- Expecting A[i]-style access: in a list it costs O(i)

Summary

- A linked list is a chain of nodes with data and a link; it is reached through its head.
- Insertion and deletion are O(1) at the right place; access by index and search are O(n).
- A doubly linked list also links back, so it can be walked both ways.
- Arrays and linked lists have opposite strengths.

Review questions

1. What does a node of a singly linked list contain? What are head and null?
2. Why can we insert at the front of a linked list in O(1), but not at the front of an array?
3. Draw head → 4 → 8 → 15 after InsertFront(head, 2) and then DeleteAfter on the node containing 8.

Reading: Levitin, Section 1.4; Bhargava, Chapter 2.