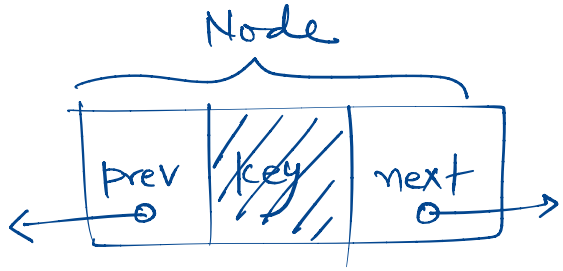
Linked Lists

- A data structure where objects are arranged in a linear order
  - ↳ a pointer in each object determines the order

- Two types of Linked Lists (LL)
  - → Singly Linked List
  - → Doubly Linked List

- A doubly Linked List node has attributes: key : data
  - a node { next : Pointer to next obj.
  - Prev : Pointer to Previous obj



- For an element x:

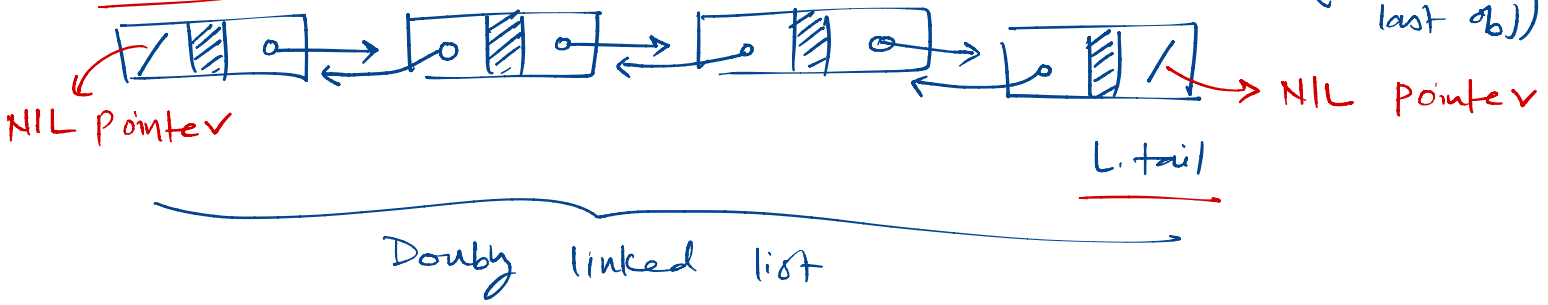
  $x.next$ points to the successor of x

  $x.prev$ " " " predecessor of x

  if $x.prev == NIL$ ⇒ No predecessor
  ⇒ x is the first element of LL
  ↳ Head of the list

  if $x.next == NIL$ ⇒ NO successor
  ⇒ x is the last element of LL
  ↳ Tail of the list

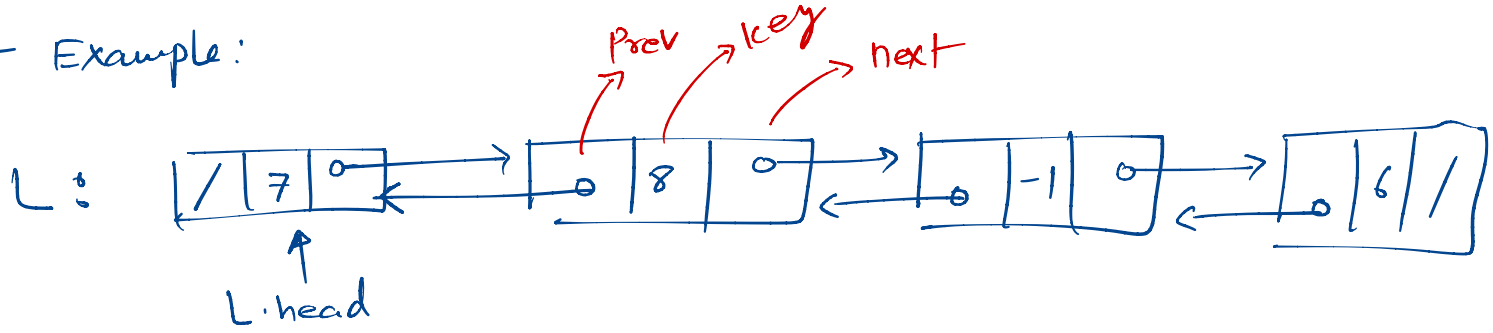
— Attributes of a doubly linked list : L.head (Points to first obj)
(" " last obj)



Doubly linked list

— A linked list may be : — singly or doubly
— sorted or unsorted
— circular or not

— Example :



— Searching a linked list
↳ find the first element with a given key
↳ Return NIL if not found

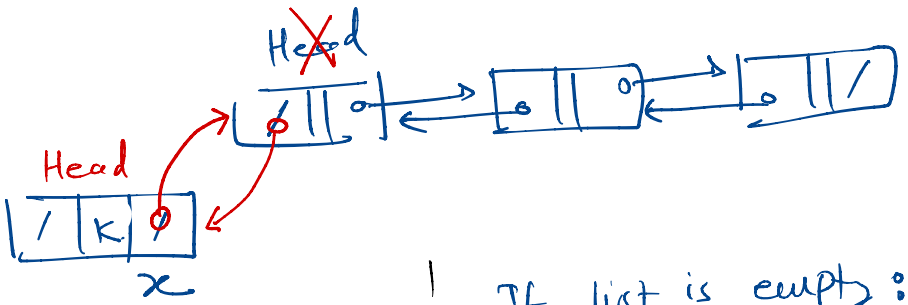
```
List-Search (L, k)
x = L.head
while x ≠ NIL and x.key ≠ k
    x = x.next
return x
```

→ $O(n)$

n: # of elements in the list

— Inserting an element/node in a given linked list

a) Prepending



List - Prepend (L, x)

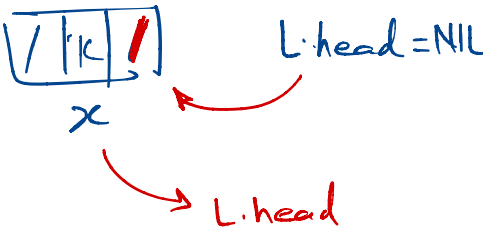
x.next = L.head

if L.head ≠ NIL

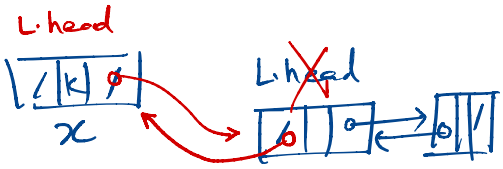
L.head.prev = x

L.head = x

If list is empty:



if list is not empty:



List - insert (x, y) : Insert node x after node y

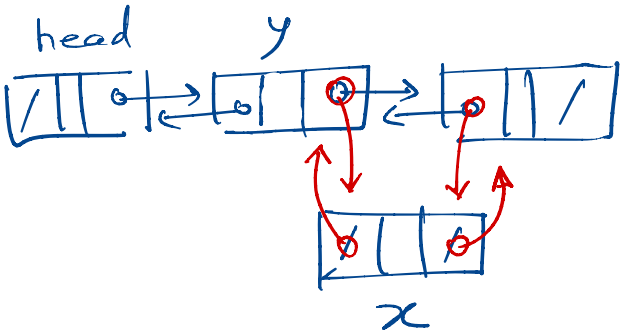
x.next = y.next

x.prev = y

if y.next ≠ NIL

| y.next.prev = x

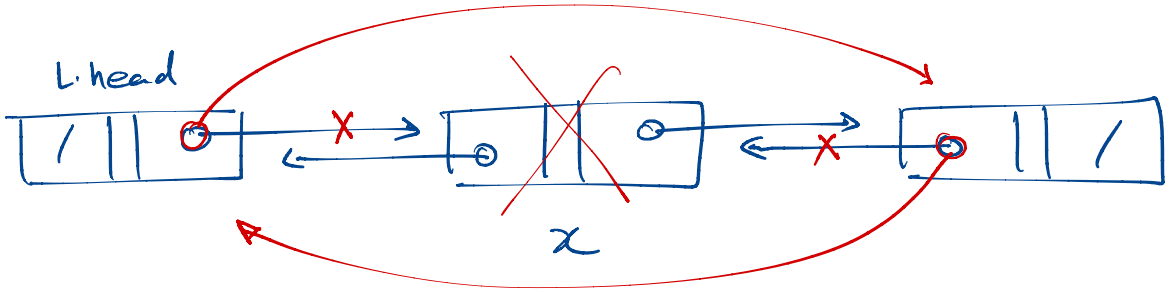
y.next = x



- Deleting a node from a linked list L
  - ↳ removes an element x from L
  - ↳ Splices x out of the list by updating pointers
  - ↳ First call list-Search to retrieve x for a given key

List-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
```



→ Sentinels

↳ a dummy object that allows simplifying boundary conditions

↳ Object L.nil that represents NIL but also contains other attributes of an LL node.

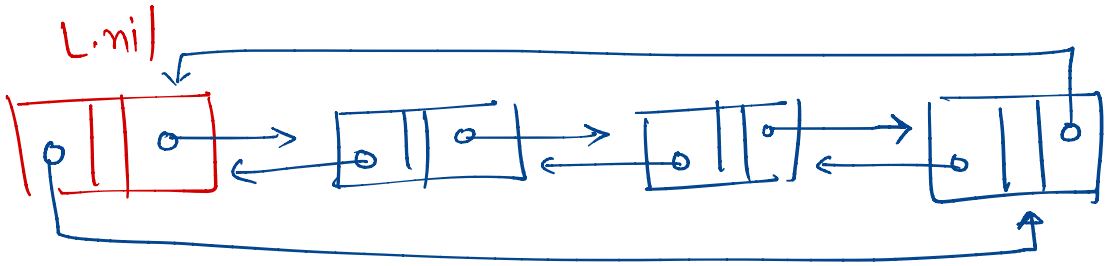
→ Replace references to NIL by L.nil

→ Turn our doubly linked list into a Circular doubly linked list

→ Sentinel L.nil lies between head & tail

⇒ L.nil.next : points to head of list

L.nil.prev : points to tail of list



List-Delete'(L, x)

$x.prev.next = x.next$

$x.next.prev = x.prev$

List-Delete (L, x)

if $x.prev \neq NIL$
  $x.prev.next = x.next$
else $L.head = x.next$
if $x.next \neq NIL$
  $x.next.prev = x.prev$

Old list-Delete without a sentinel

List-Search'(L, k)

```
L.nil.key = k
x = L.nil.next
while (x.key ≠ k)
        x = x.next
if x = L.nil
        return NIL
else return x
```

List-Insert'(x, y)

```
x.next = y.next
x.prev = y
y.next.prev = x
y.next = x
```

List-insert(x, y)

```
x.next = y.next
x.prev = y
if y.next ≠ NIL
    y.next.prev = x
y.next = x
```

Previous List-Insert without ~~—~~ Sentinel

— No prepend function needed. In order to prepend at head, use $y = L.nil$

Todo: Verify the claim!

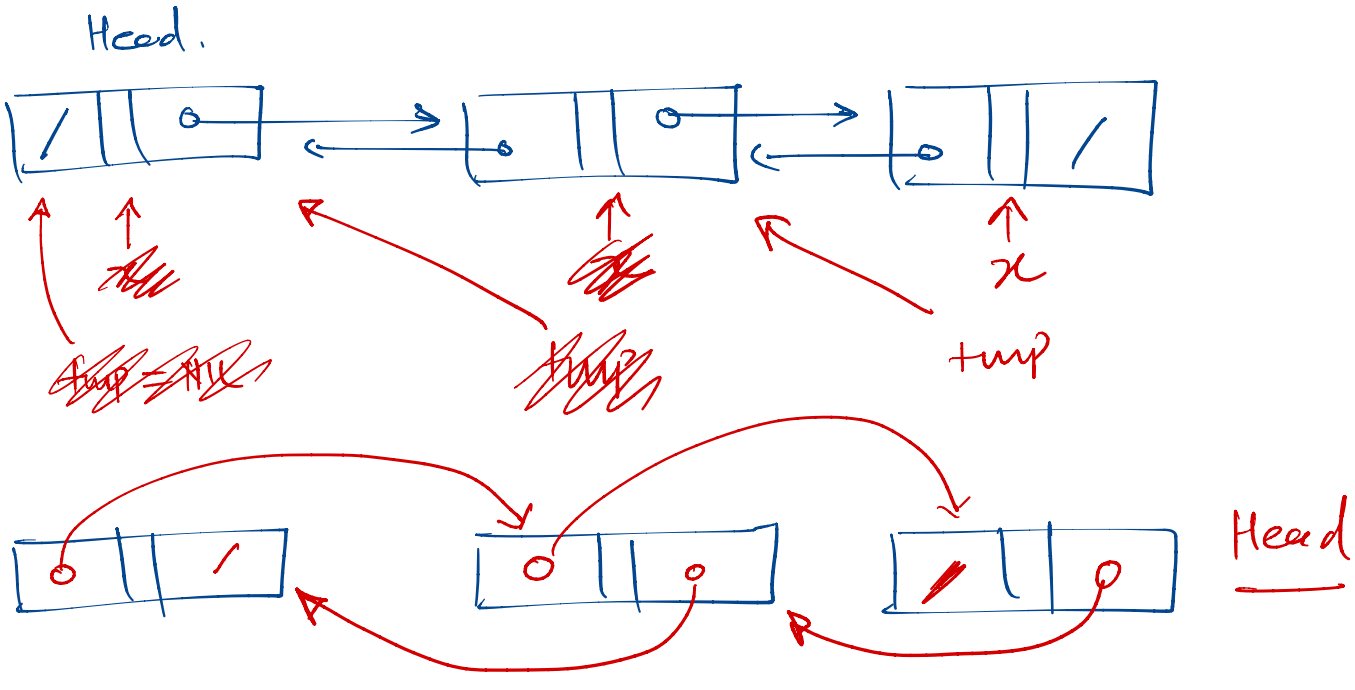
→ Reversing a doubly linked list

↳ Swap prev & next pointers

↳ Assign last node to head

List - Reverse (L)

```
x = L.head
While (x != NIL)
    tmp = x->prev
    x.prev = x.next
    x.next = tmp
    x = x.prev
End While
L.head = tmp.Prev
```



→ Joining two lists

↳ Given lists L_1 & L_2, return a concatenation of L_1 & L_2.

List-Join (L_1, L_2)

$L_1.tail.next = L_2.head$

$L_2.head.prev = L_1.tail$

return $L_1.head$
