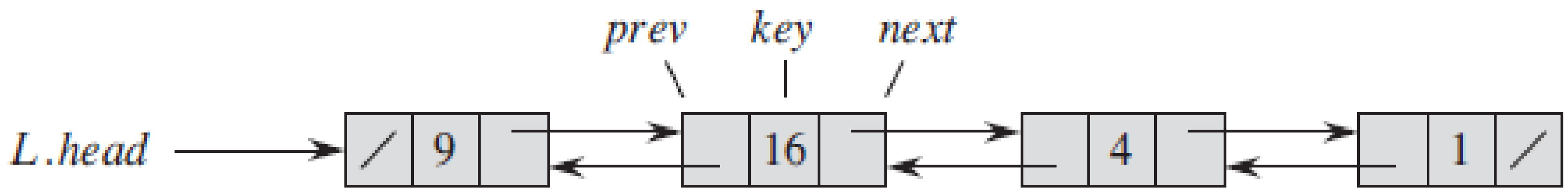
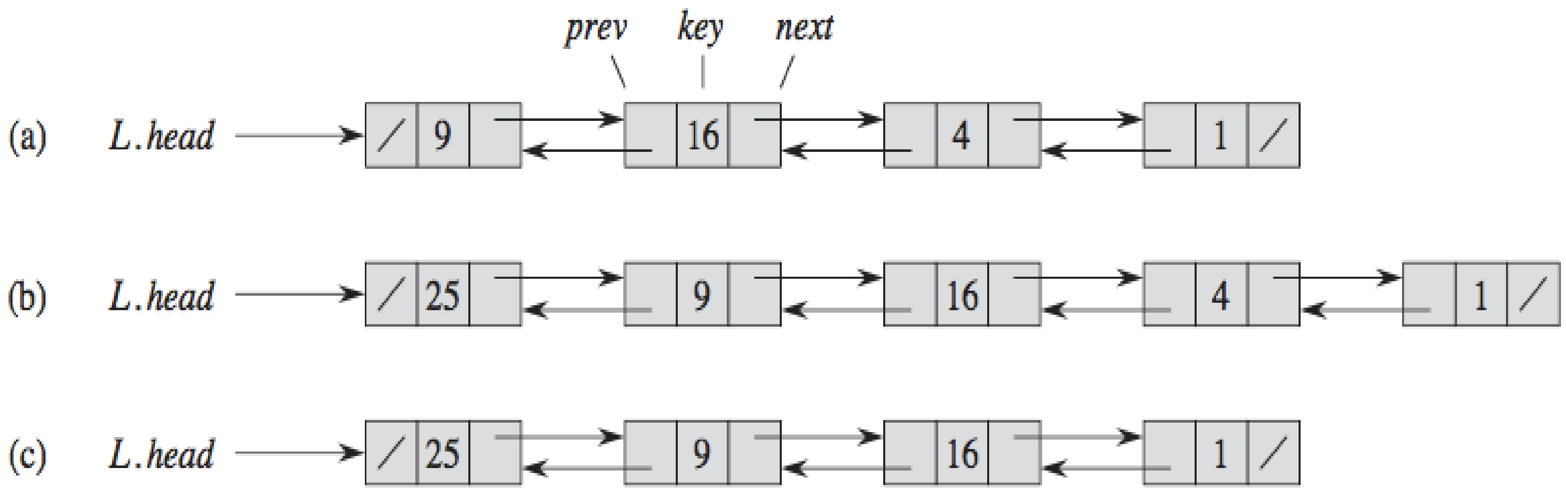
Doubly Linked List (DLL)

Doubly Linked List (DLL)

- Each node has three fields
- Data (key), next, prev
- What is the advantage?
- Traversing the DLL
- Counting the DLL



Doubly Linked List



(a) DLL representing the set {1,4,9,16)
(b) After inserting 25
(c) After deleting 4

DLL insert operation

```
LIST-INSERT(L, x)
x.next = L.head
if L.head ≠ NIL
    L.head.prev = x
L.head = x
x.prev = NIL
```

- Inserting an element at the front of the linked list.

- Running time ?

DLL Deletion

```
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
```

- Given a pointer to x, LIST-DELETE removes *x* out of the list by updating pointers.
- To delete an element with a given key, first call LIST-SEARCH to retrieve a pointer to the element.
- Running time ?

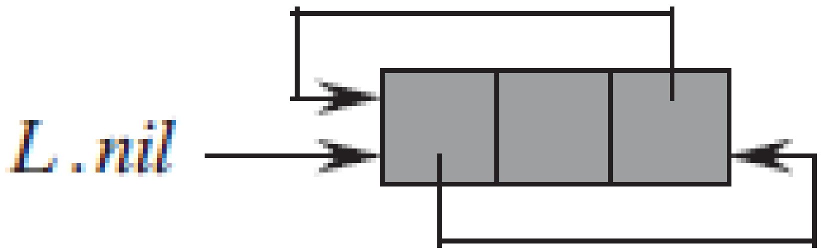
DLL DELETE (without boundary conditions)

LIST-DELETE$'(L, x)$

1 $x.prev.next = x.next$
2 $x.next.prev = x.prev$

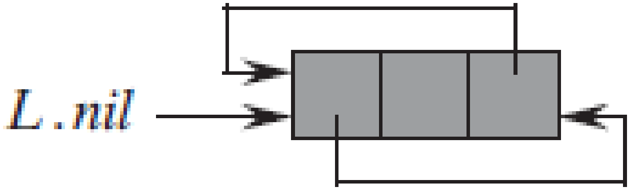
Sentinels

- A ***sentinel*** is a dummy object - to simplify boundary conditions.
- Eg: Suppose that we provide with list L an object L.*nil* that represents NIL but has all the attributes of the other objects in the list.



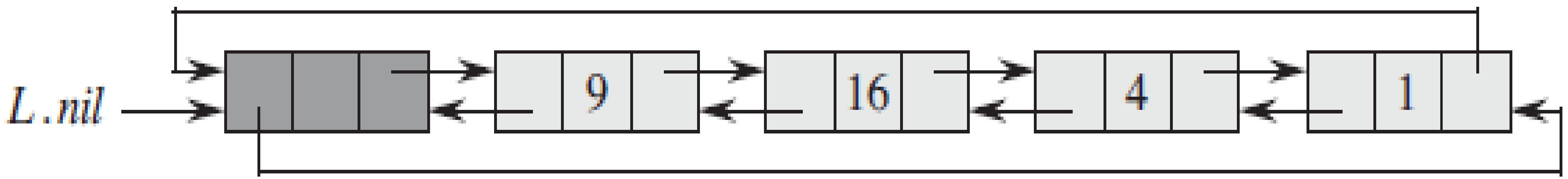
- The above node is pointed by L.nil and it has the attributes, next, prev and data fields like other objects.
- Wherever we have a reference to NIL in list code, we replace it by a reference to the sentinel L.*nil*.

Circular Doubly Linked List



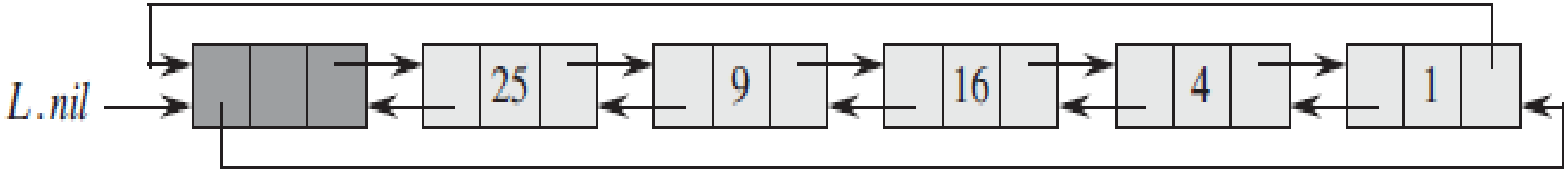
- ***Circular doubly linked list with a sentinel*** - sentinel L.*nil* lies between the head and tail.
- The attribute L.*nil*.*next* points to the head of the list, and L.*nil*.*prev* points to the tail.
- Similarly, both the *next* attribute of the tail and the *prev* attribute of the head point to L.*nil*.
- Since L.*nil*.*next* points to the head, we can eliminate the attribute L.*head* altogether, replacing references to it by references to L.*nil*.*next*.
- Empty list consists of just the sentinel, and both L.*nil*.*next* and L.*nil*.*prev* point to L.*nil*.

CLL Search




```
LIST-SEARCH'(L, k)
x = L.nil.next
while x ≠ L.nil and x.key ≠ k
    x = x.next
return x
```

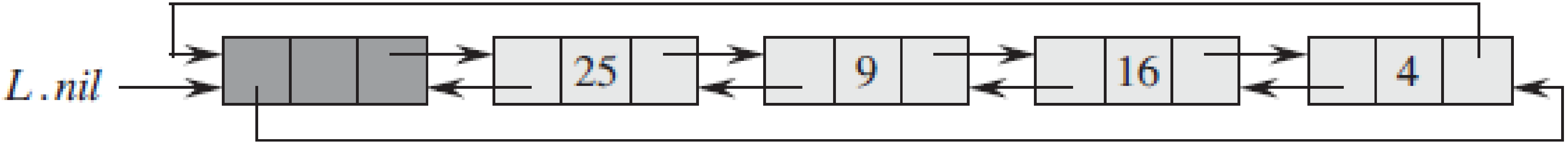
CLL Insert



LIST-INSERT$'(L, x)$

$x.next = L.nil.next$
$L.nil.next.prev = x$
$L.nil.next = x$
$x.prev = L.nil$

CLL Delete



LIST-DELETE′(L, x)

1 $x.prev.next = x.next$
2 $x.next.prev = x.prev$

Use of Sentinels in Linked List code

- Linked list code, for example, becomes simpler when we use sentinels, but we save only O(1) time in the CLL LIST-INSERT’ and CLL LIST-DELETE’ procedures.
- We should use sentinels judiciously. When there are many small lists, the extra storage used by their sentinels can represent significant wasted memory.
- Use sentinels only when they truly simplify the code.