Al-Mustaqbal University
College of Sciences
Intelligent Medical System Department

جامعة المستقبل
AL MUSTAQBAL UNIVERSITY

كلية العلوم

قسم الانظمة الطبية الذكية

Intelligent Medical Systems Department

Subject: Data Structure

Class: Second

Lecturer: Prof.Dr. Mehdi Ebady Manaa

Lecture: (8)

Doubly Linked Lists

Doubly Linked Lists

Let's examine another variation on the linked list: the *doubly linked* list. What's the advantage of a doubly linked list? A potential problem with ordinary linked lists is that it's difficult to traverse backward along the list. A statement like `current=current.next` steps to the next link, but there's no corresponding way to go to the previous link.

For example, imagine a text editor in which a linked list is used to store the text. Each text line on the screen is stored as a `String` object embedded in a link. When the editor's user moves the cursor downward on the screen, the program steps to the next link to manipulate or display the new line. But what happens if the user moves the cursor upward? In an ordinary linked list, you'd need to return `current` (or its equivalent) to the start of the list and then step all the way down again to the new current link. This isn't very efficient. You want to make a single step upward.

The doubly linked list provides this capability. It allows you to traverse backward as well as forward through the list. The secret is that each link has two references to other links instead of one. The first is to the next link, as in ordinary lists. The second is to the previous link. This is shown in Figure 1.

Figure (1) Double link list

The beginning of the specification for the `Link` class in a doubly linked list looks like this:

```
class Link:
    def __init__(self, dData):
        self.dData = dData
        self.next = None
        self.previous = None
```

The downside of doubly linked lists is that every time you insert or delete a link you must deal with four links instead of two: two attachments to the previous link and two attachments to the following one. Also, of course, each link is a little bigger because of the extra reference.

Traversal

Two display methods demonstrate traversal of a doubly linked list. The `displayForward()` method is the same as the `displayList()` method we've seen in ordinary linked lists. The `displayBackward()` method is similar, but starts at the last element in the list and proceeds toward the start of the list, going to each element's `previous` field. This code fragment shows how this works:

```
current = last  # Start at the end
```

```
while current is not None:  # Until the start of the list
    current = current.previous  # Move to the previous link
```

Insertion

We've included several insertion routines in the `DoublyLinkedList` class. In addition, the `insertFirst()` method inserts at the beginning of the list, `insertLast()` inserts at the end, and `insertAfter()` inserts following an element with a specified key.

Unless the list is empty, the `insertFirst()` routine changes the `previous` field in the old `first` link to point to the new link, and changes the `next` field in the new link to point to the old first link. Finally it sets `first` to point to the new link. This is shown in Figure 2.

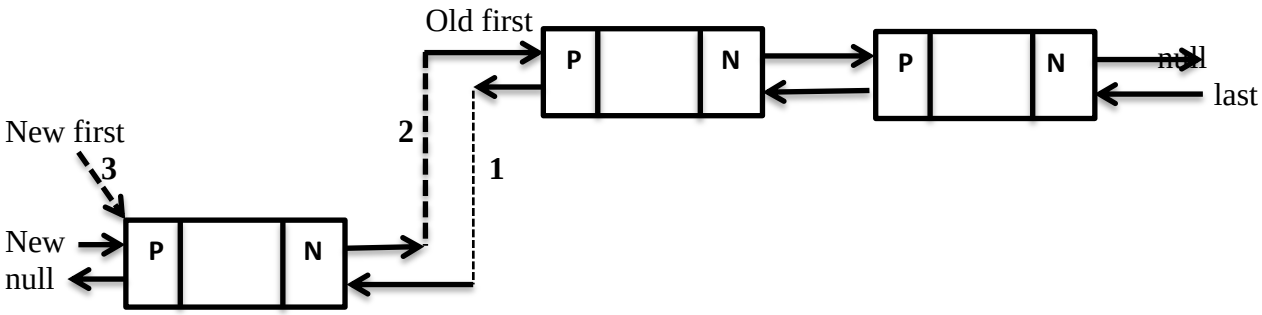


Figure (2) insertion at first of Double link list

If the list is empty, then the `last` field must be changed instead of the `first.previous` field. Here's the code in python :

```
if self.is_empty():  # If the list is empty
    self.last = new_link  # newLink <-- last
else:
    self.first.previous = new_link  # newLink <-- old first

new_link.next = self.first  # newLink --> old first
self.first = new_link  # first --> newLink
```

The code in above inserts a new link into the doubly linked list, either at the beginning (if the list is not empty) or as the first element (if the list is empty). The code properly adjusts the next and previous references to maintain the doubly linked structure. new_link.next = self.first: Here, it sets the next reference of the new_link to point to the old first element. This effectively connects the new_link as the next element after the old first. It's a bit more complicated because four connections must be made. First the link with the specified key value must be found. Then, assuming we're not at the end of the list, two connections must be made between the new link and the next link, and two more between `self` and the new link. This is shown in Figure 3.

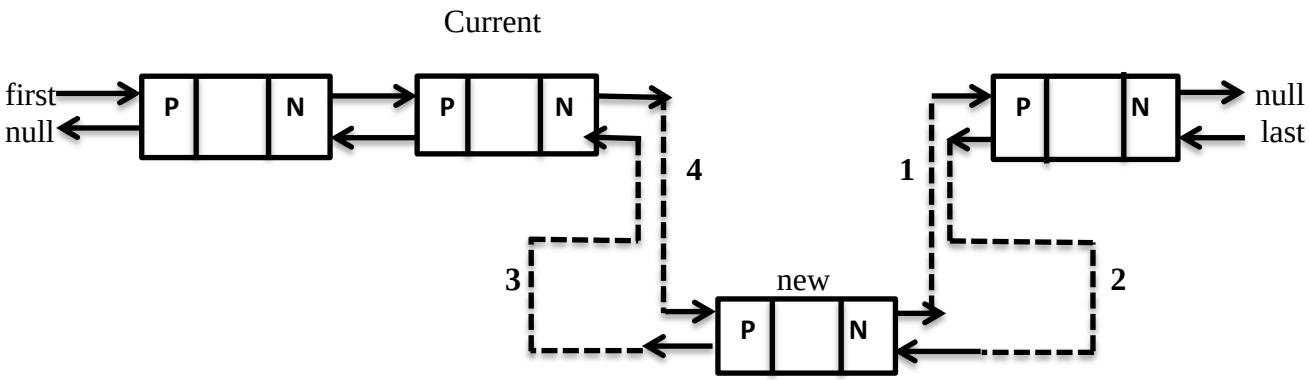


Figure (3) insertion aftert a location in Double link list

If the new link will be inserted at the end of the list, then its `next` field must point to `null`, and `last` must point to the new link. It inserts a **newLink** into a doubly linked list, and it properly updates the next and previous references to maintain the doubly linked structure.

```
if current == last:  # If it's the last link
    newLink.next = None  # newLink --> None
    last = newLink  # newLink <-- last
else:  # If it's not the last link
    newLink.next = current.next  # newLink --> old next
 #    newLink <-- old next
    current.next.previous = newLink
newLink.previous = current  # old current <-- newLink
current.next = newLink  # old current --> newLink
```

Deletion

There are three deletion routines: `deleteFirst()`, `deleteLast()`, and `deleteKey()`. The first two are fairly straightforward. In `deleteKey()`, the key being deleted is `current`. Assuming the link to be deleted is neither the first nor the last one in the list, then the `next` field of `current.previous` (the link before the one being deleted) is set to point to `current.next` (the link following the one being deleted), and the `previous` field of `current.next` is set to point to `current.previous`. This disconnects the current link from the list. Figure 4 shows how this disconnection looks, and the following two statements carry it out:

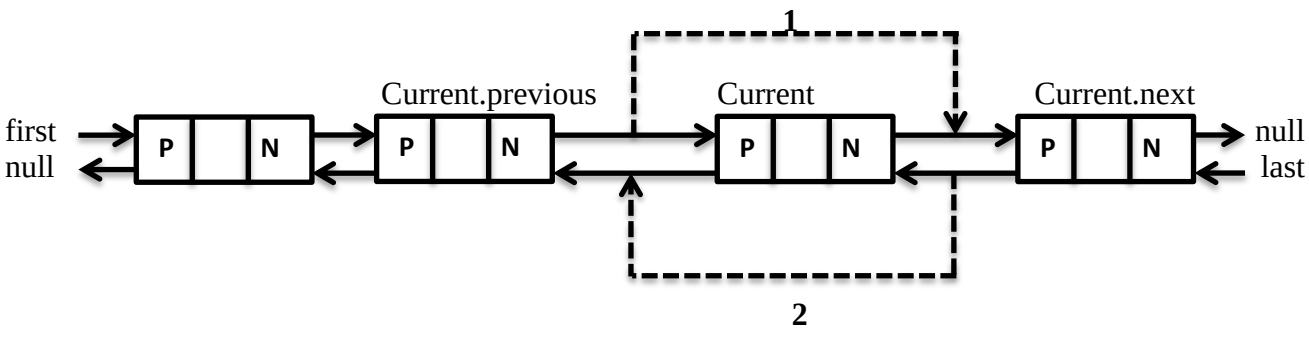


Figure (4) deletion in Double link list

```
current.previous.next = current.next
current.next.previous = current.previous
```

Special cases arise if the link to be deleted is either the first or last in the list, because `first` or `last` must be set to point to the next or the previous link. Here's the code from `deleteKey()` for dealing with link connections:

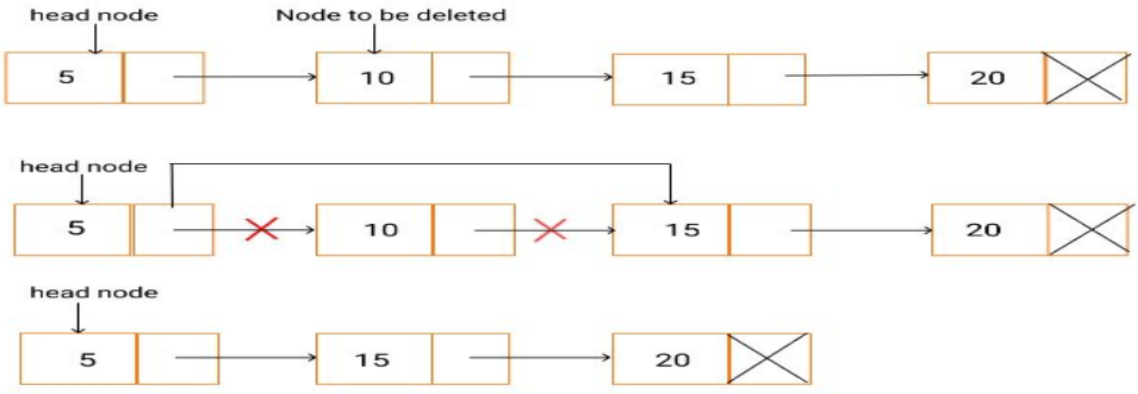
```
if current == first:  # first item?
    first = current.next  # first --> old next
else:  # not first
    current.previous.next = current.next  # old previous --> old
next

if current == last:  # last item?
    last = current.previous  # old previous <-- last
else:  # not last
    current.next.previous = current.previous  # old previous <--
old next
```

Deletion by Item Value

To delete the element by value:

- Find the node that contains the item with the specified value.
- delete the node.
- The reference of the node before the item is set to the node that exists after the item being deleted.

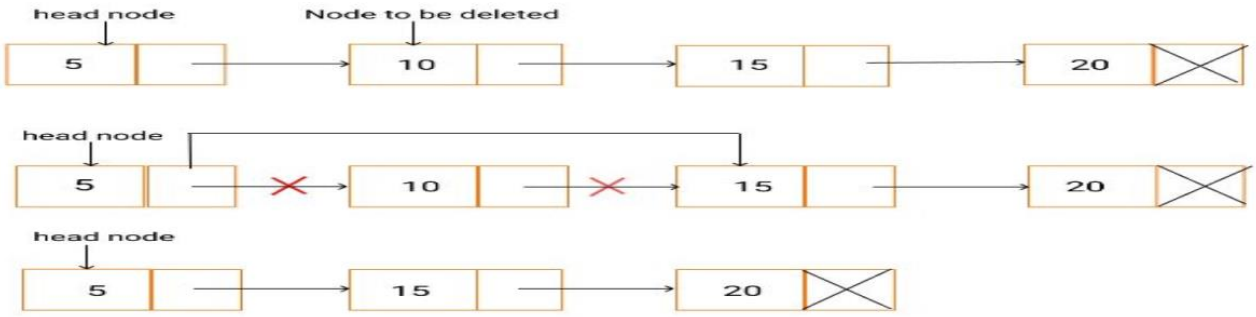



```
def delete_element_by_value(self, x):
   if self.start_node is None:
      print("The list has no element to delete")
      return

 #    Deleting the first node
```

```
if self.start_node.item == x:
    self.start_node = self.start_node.ref
    return
n = self.start_node
while n.ref is not None:
    if n.ref.item == x:
        break
    n = n.ref

if n.ref is None:
    print("Item not found in the list")
else:
    n.ref = n.ref.ref
```



HW.

1- How can count the number of nodes in linked list?
2- How can search to find node in linked list ?