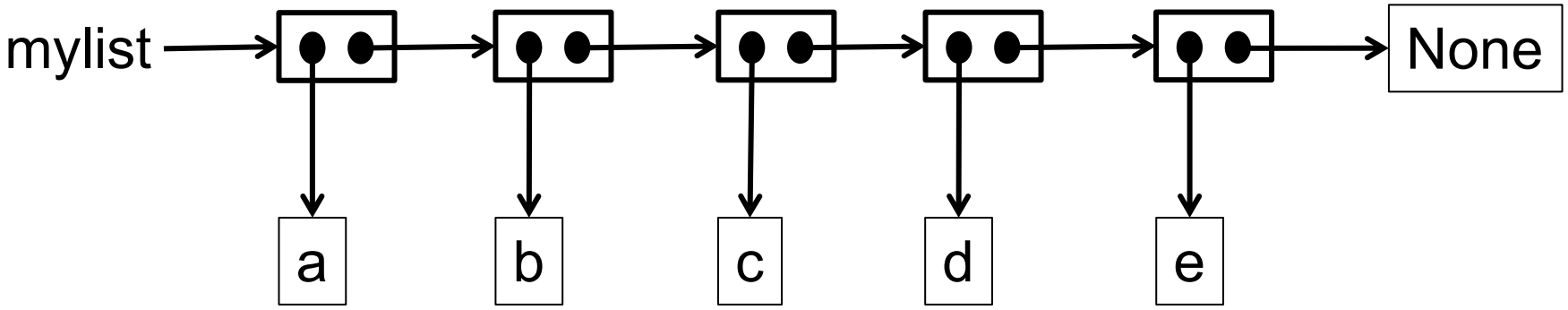
The Doubly Linked List

Revision: Singly Linked List




```
class SLLNode:
    def __init__(self, item, nextnode):
        self.element = item       #any object
        self.next = nextnode      #an SLLNode
```

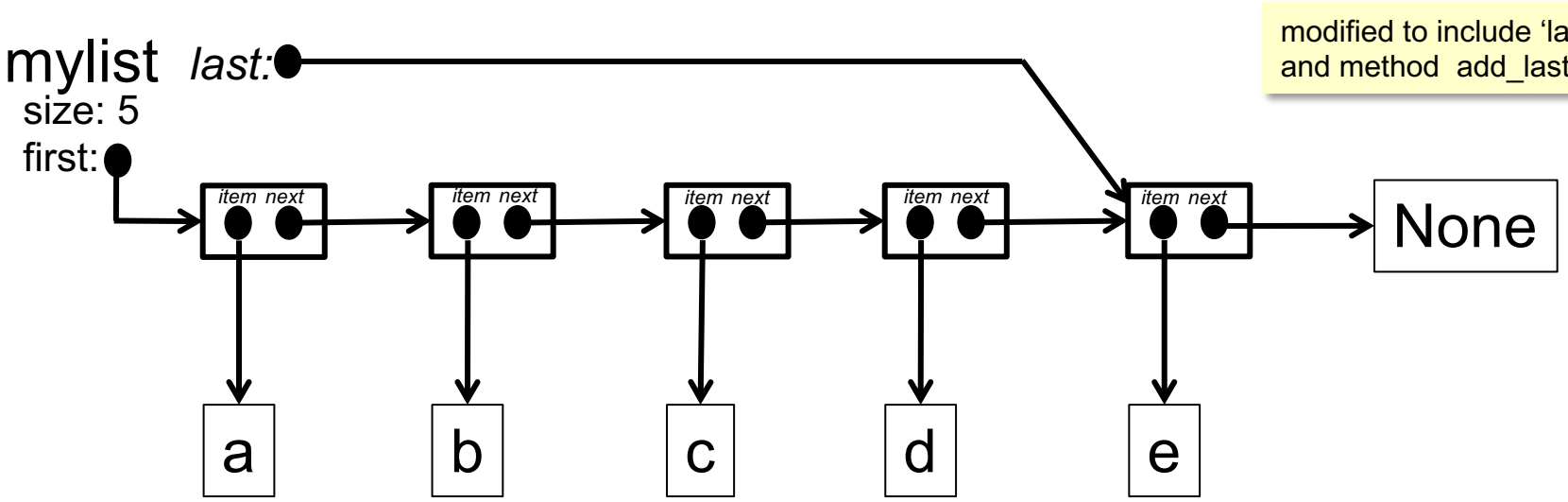
SLL
first # SLLNode
size # int

```
class SLinkedList:
    def __init__(self):
        self.first = None         #an SLLNode
        self.size = 0             #an integer

    def add_first(self, item):    #add at front of list
    def get_first(self):          #report the first element
    def remove_first(self):       #remove the first element
    def length(self):             #report the number of elements
```

SLLNode
item # Object
next # SLLNode

All of these methods can be implemented to run in O(1) time



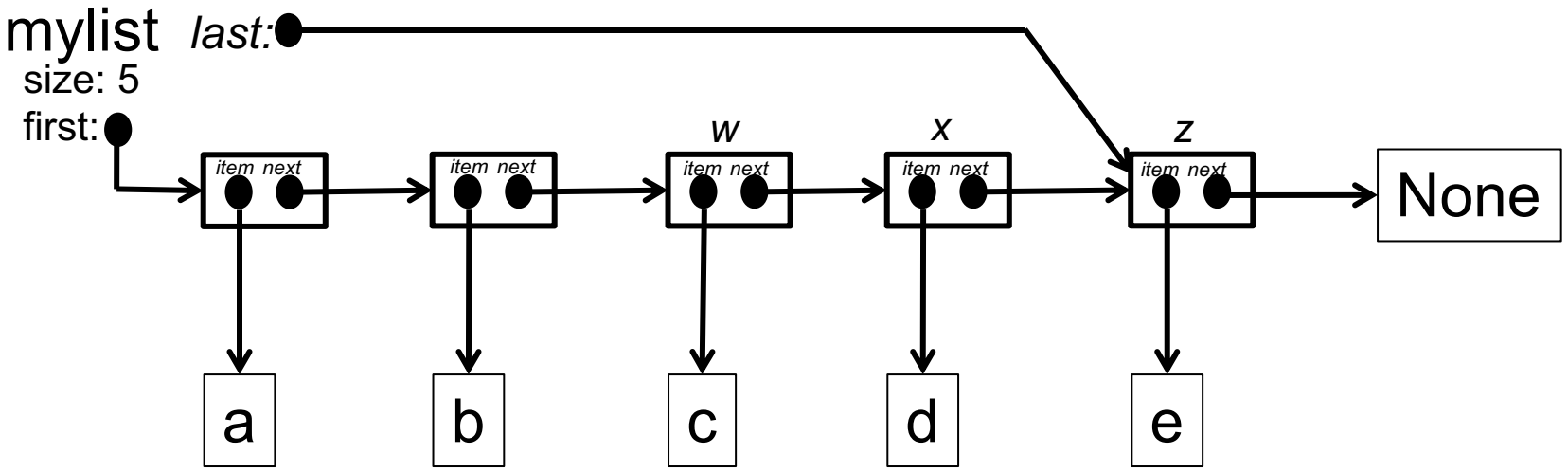
modified to include 'last' and method add_last()

LinkedLists are a general data structure – we would like to be able to use them for any sequential storage.

What happens if we want to remove an item from the middle of the list, or the end of the list? Or add a new item after one that is somewhere in the middle?

SLL
first # SLLNode
last # SLLNode
size # int

SLLNode
item # Object
next # SLLNode



To remove the node *x* containing *d*, we first need to find x, then we need to link *w* to *z*, by setting *w*.next = *z*.

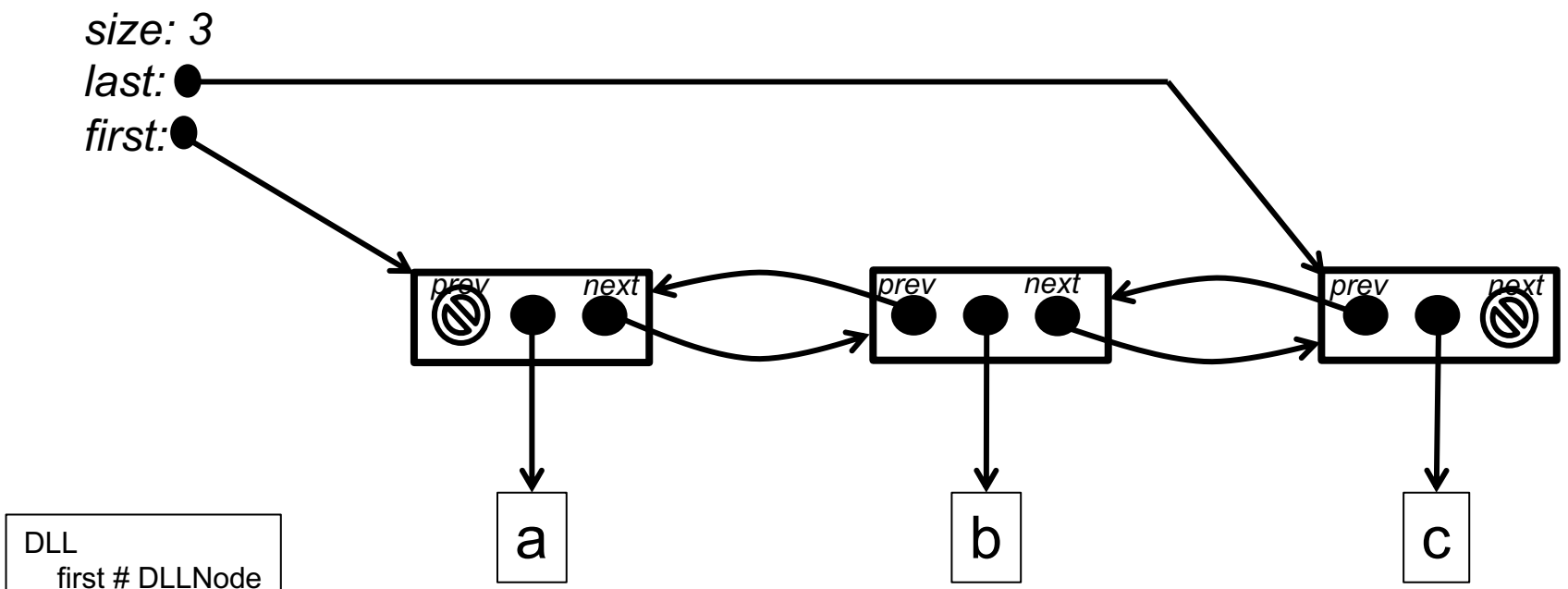
x
item: d
next: z

We can find *z* from *x*.next.
But how do we get hold of *w*?

Even if we are given a reference to the node we want to remove, we still must search through the list to find its predecessor.

first version ...

mylist

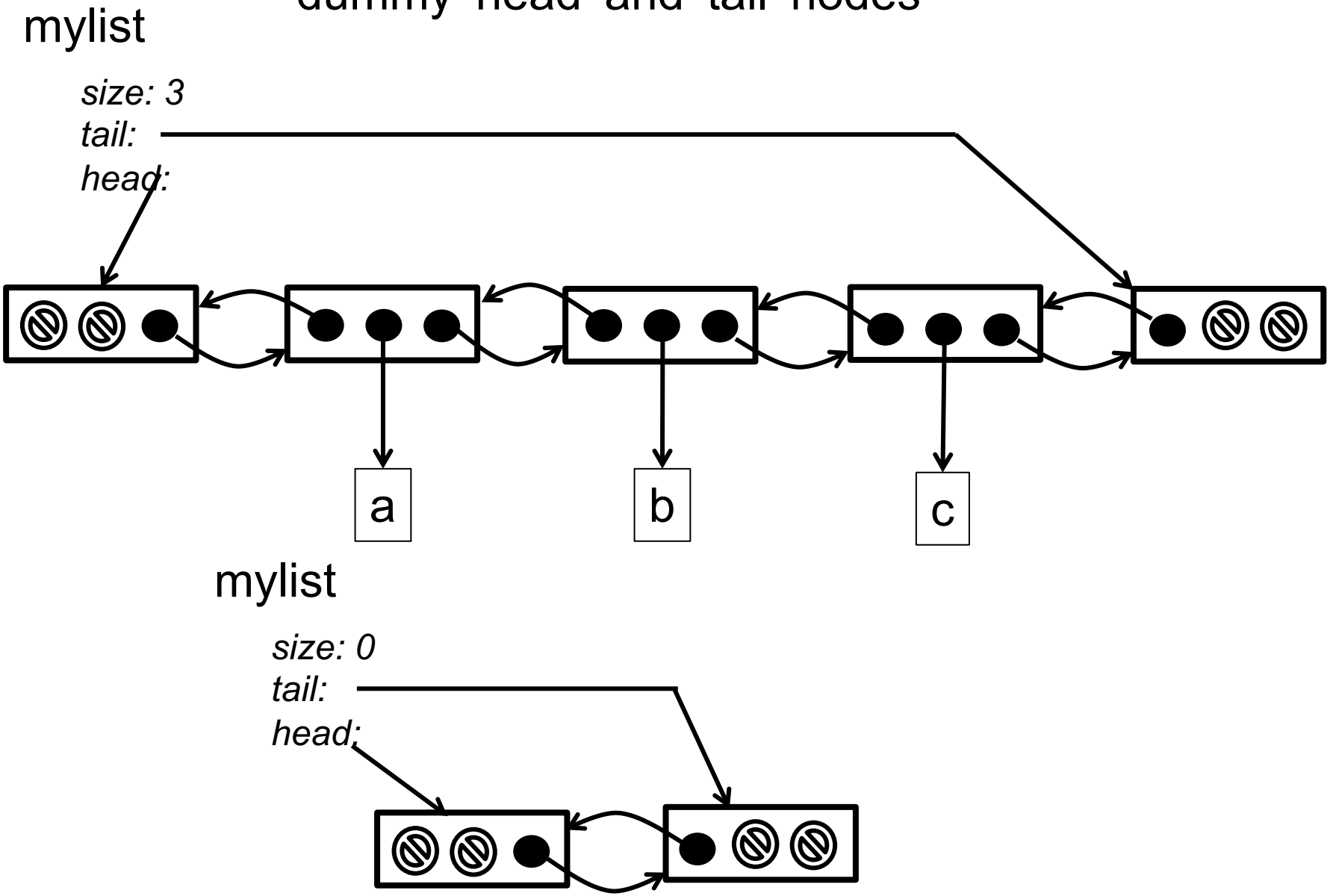


DLL
first # DLLNode
last # DLLNode
size # int

DLLNode
item # Object
prev # DLLNode
next # DLLNode

From now on, we will use ⊘ instead of a link to the None object

To make the operations easier, we will use dummy 'head' and 'tail' nodes



```
class DLLNode:
    def __init__(self, item, prevnode, nextnode):
        self.element = item
        self.next = nextnode
        self.prev = prevnode

class DLinkedList:
    def __init__(self):

    def add_after(self, item, before):

    def add_first(self, item):

    def add_last(self, item):

    def get_first(self):

    def get_last(self):

    def remove_node(self, node):

    def remove_first(self):

    def remove_last(self):
```

Note: implementing these is tricky – you need to be very careful that you don't create cycles, or dead-ends in the middle of the list.

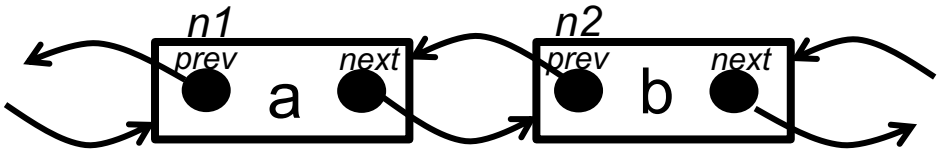
add_after(item, n1)

mylist

size: k
tail:
head:

n1
  next: n2

n2
  prev: n1



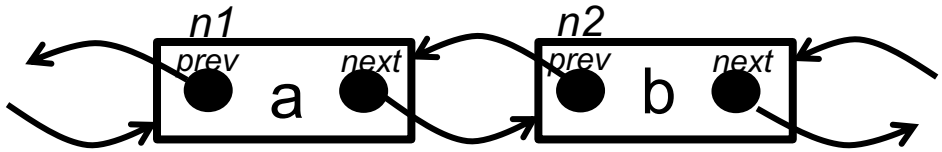
add_after(item, n1)

mylist

size: k
tail:
head:

n1
  next: n2

n2
  prev: n1



... becomes ...

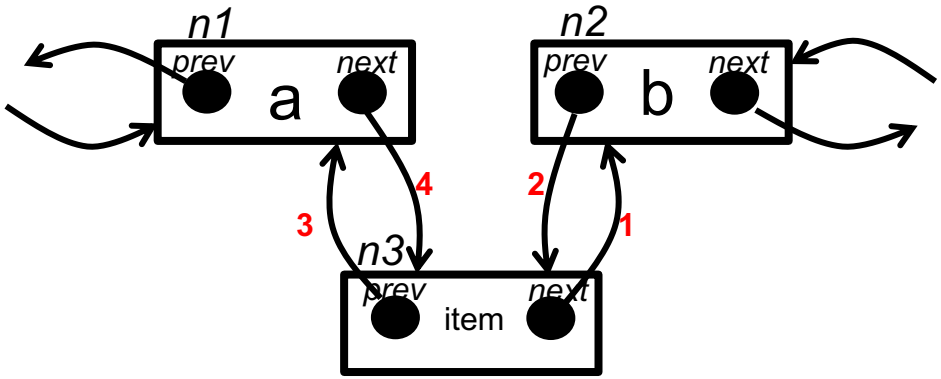
mylist

size: k+1 (5)
tail:
head:

n1
  next: n3

n2
  prev: n3

n3
  next: n2
  prev: n1
  elt: item



```
class DLLNode:
    def __init__(self, item, prevnode, nextnode):
        self.element = item
        self.next = nextnode
        self.prev = prevnode

class DLinkedList:
    def __init__(self):

    def add_after(self, item, before):

    def add_first(self, item):

    def add_last(self, item):

    def get_first(self):

    def get_last(self):

    def remove_node(self, node):

    def remove_first(self):

    def remove_last(self):
```

Exercise: implement a method to swap two *adjacent* nodes in the list.
Exercise: implement a method to swap two *arbitrary* nodes in the list.

Do this by

(i) Leaving the DLLNodes as they are, and just swapping the items

(ii) Leaving the items as they are, and swapping the next and prev pointers for DLLNodes as required.

Next lecture ...

A review of list implementations, and a problem with their complexity