

Extended Introduction to Computer Science

CS1001.py

Chapter G Data Structures 1: Linked Lists

Lecture 14b

Michal Kleinbort, Amir Rubinstein

School of Computer Science
Tel-Aviv University
Fall Semester 2023-24
<http://tau-cs1001-py.wikidot.com>

* Slides based on a course designed by Prof. Benny Chor

Overview

- ✓ Intro to Object Oriented Programming (OOP)
- Data Structures
 1. Linked Lists (← today)
 2. Binary Search Trees
 3. Hash tables
 4. Iterators and generators

This Lecture Plan

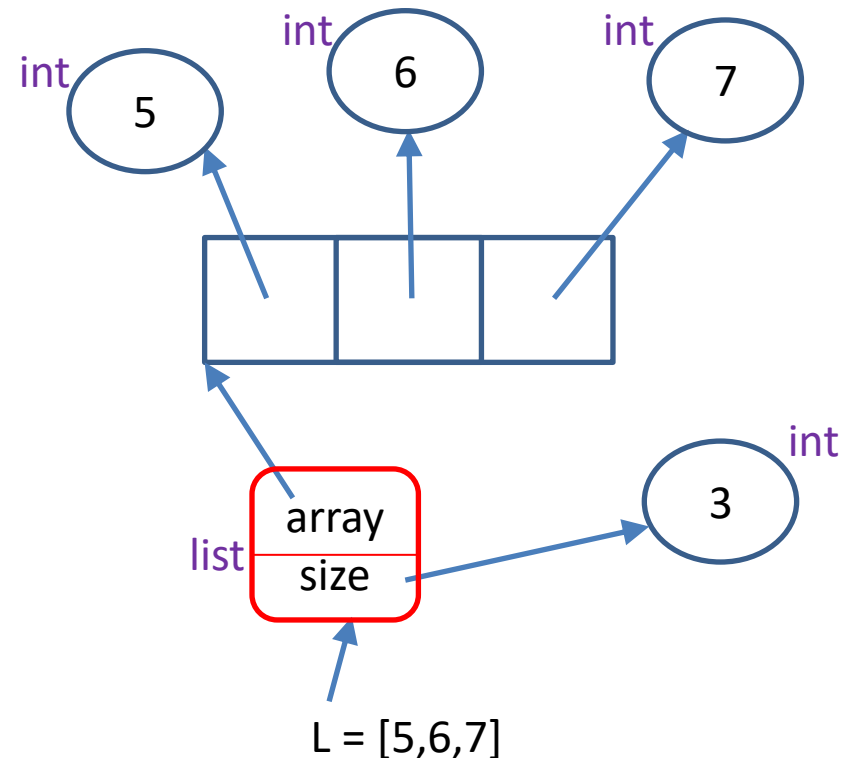
- Intro to Data Structures
- Python Lists vs. Linked Lists
- Implementation for `class Linked_list`

Data Structures

- A **data structure** is a way to **organize** data in memory as to support various operations of the data.
- The choice of data structures for a particular problem depends on the desired operations and complexity constraints (time and memory).
- We have seen some **built-in** Python data structures: **strings**, **tuples**, **lists**, **dictionaries**. In fact, "atomic" types, such as **int** or **float**, may also be considered structures, albeit **primitive** ones.
- To distinguish the functionality of a class from its concrete implementation, The term **Abstract Data Type (ADT)** is often used. It emphasizes the point that the user (client) needs to know **what operations** are allowed, but not **how** they are **implemented**.
- OOP supports this approach naturally, as we have seen

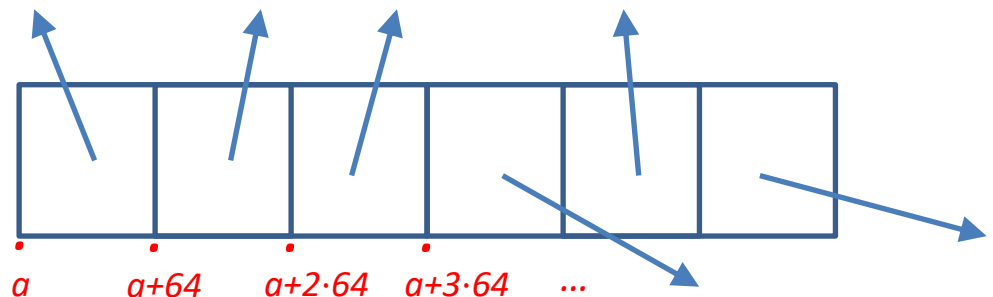
Representing Lists in Python

- We have extensively used Python's built-in **lists**.
- As we already know, “under the hood”, a Python **list** is stored as an **array**: a **contiguous** space of **pointers** used as references to (addresses of) other objects (each pointer normally takes 32/64 bits).
- A **list** basically keeps the address of the **beginning** of this **array** in memory, plus its **length**.



“Random Access”

- The fact that the list stores pointers, and **not** the elements themselves, enables Python's lists to contain objects of **heterogeneous types** (something not possible in some other programming languages).
- But most importantly, this makes **accessing/modifying** a list element, `lst[i]`, an operation whose cost is $O(1)$ - **independent** of the **size** of the list or the value of the **index**. This is termed **random access**.
- If the address in memory of `lst[0]` is **a** , and assuming each pointer takes 64 bits, then the address in memory of **`lst[i]`** is simply **$a+64i$** .

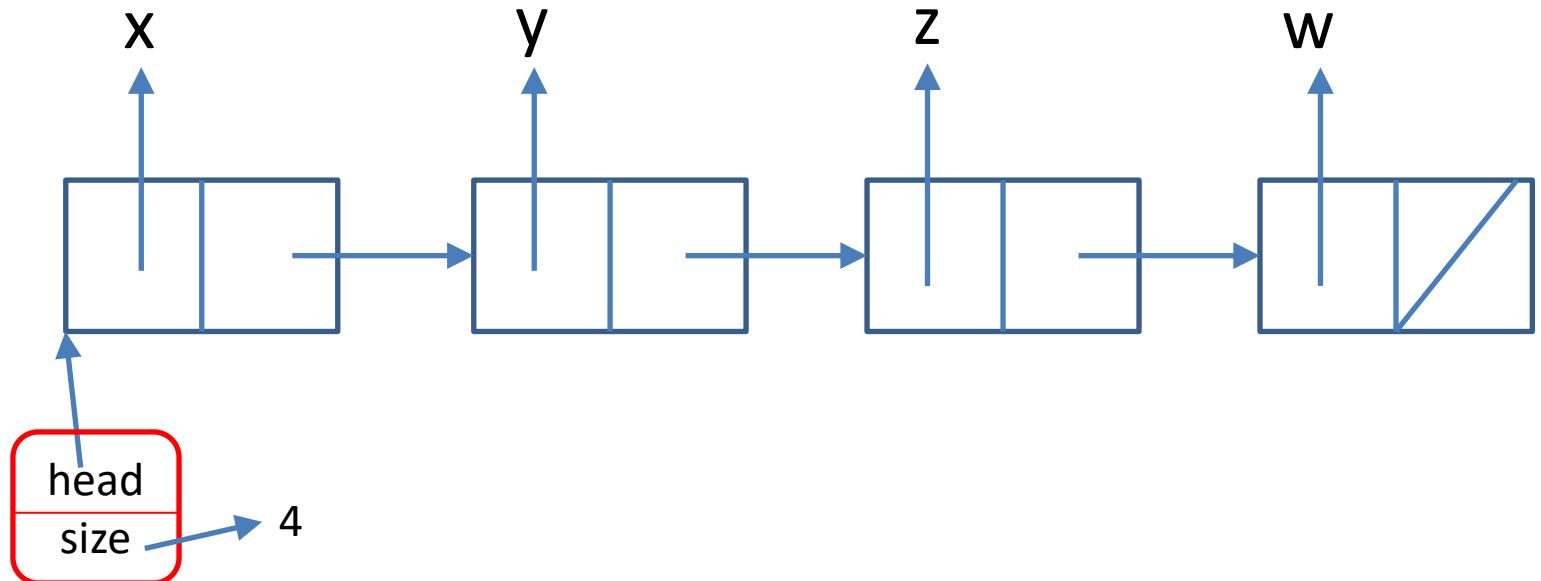


Disadvantages of Random Access

- However, the contiguous storage of addresses must be maintained when the list **evolves**.
- In particular if we want to **insert/delete** an item at location **i**, all items from **location i onwards** must be “**pushed**” forward/backward.
 - **$O(n)$** operations in the worst case for lists with **n** elements.
- Moreover, if we use up all of the memory block allocated for the list, we may need to move items to get a block of **larger size** (possibly starting in a different location).
 - Comment: some cleverness is applied to **improve the performance** of appending items repeatedly; when the array must be grown, extra space is allocated right away, so the next few times do not require an actual resizing (taken from this [source](#)).

The Alternative: **Linked Lists**

- An alternative to using a contiguous block of memory, is to specify, for each item, the memory location of the **next** item in the list.
- We can represent this graphically using a **boxes-and-pointers diagram**.



The Alternative: **Linked Lists**

- We will implement two **classes**. One for **nodes** in the list, and another one to represent the **list**.
- We will try to keep the **interface** (names of methods and how they are used) the same as for Python **lists**. For example:

```
lst = Linked_list() #empty linked list
lst.insert(0,3) #insert 3 at position 0
lst.insert(0,5)
lst.insert(1,4)
lst.insert(2,7)
print(len(lst))           4
print(lst)                [5, 4, 7, 3]
print(lst[2])             7
print(lst.index(7))       2
lst.pop(0) #remove element at position 0
lst[1] = 999
print(lst)                [4, 999, 3]
```

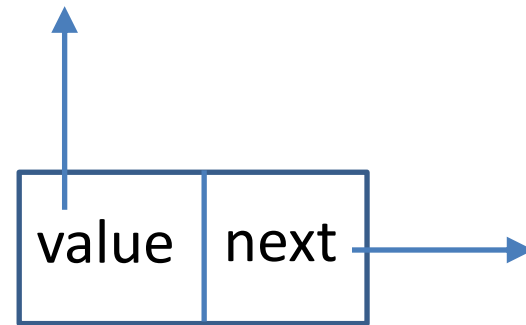
class Node

- Class Node is very simple, holding just two fields, as illustrated in the diagram.

```
class Node:
```

```
    def __init__(self, val):  
        self.value = val  
        self.next = None
```

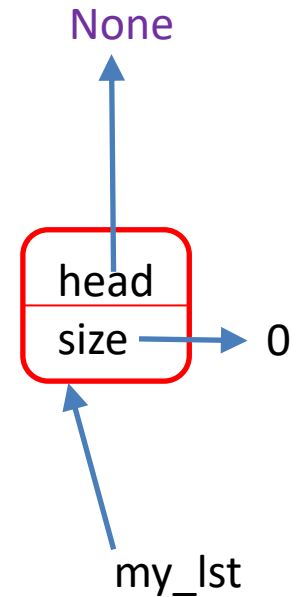
```
    def __repr__(self):  
        return str(self.value)
```



class Linked_list

```
class Linked_list:  
    def __init__(self):  
        self.head = None  
        self.size = 0
```

```
>>> my_lst = Linked_list()
```



Linked List Operations:

Insertion at the **Start**

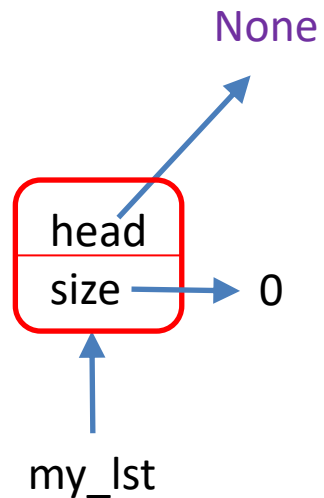
```
def add_at_start(self, val):  
    ''' add node with value val at the list head '''  
    tmp = self.head  
    self.head = Node(val)  
    self.head.next = tmp  
    self.size += 1
```

- Note: time complexity is **O(1)** in the worst case!

Memory View (1)

```
>>> my_lst = Linked_list()
```

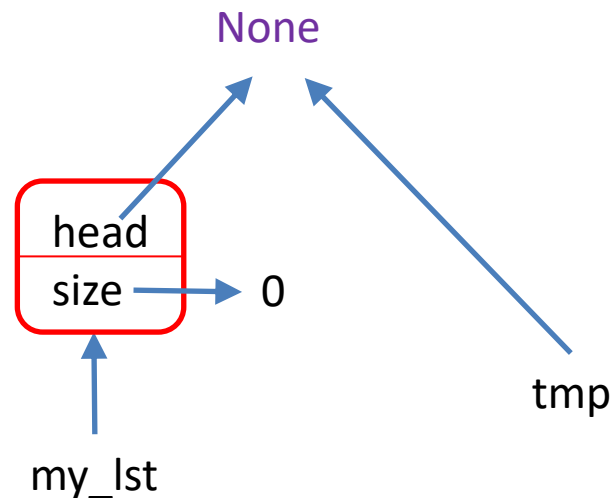
```
def __init__(self):  
    self.head = None  
    self.size = 0
```



Memory View (2)

```
def add_at_start(self, val):  
    tmp = self.head  
    self.head = Node(val)  
    self.head.next = tmp  
    self.size += 1
```

```
>>> my_lst.add_at_start("a")
```

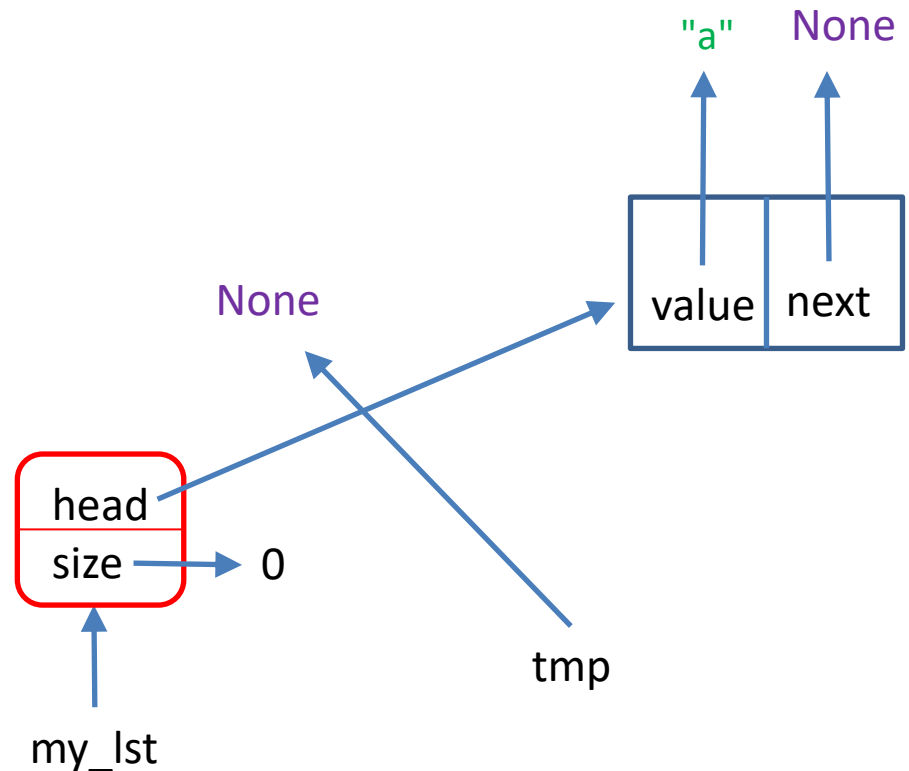


Memory View (2)

```
def add_at_start(self, val):  
    tmp = self.head  
    self.head = Node(val)  
    self.head.next = tmp  
    self.size += 1
```

```
class Node:  
    def __init__(self, val):  
        self.value = val  
        self.next = None
```

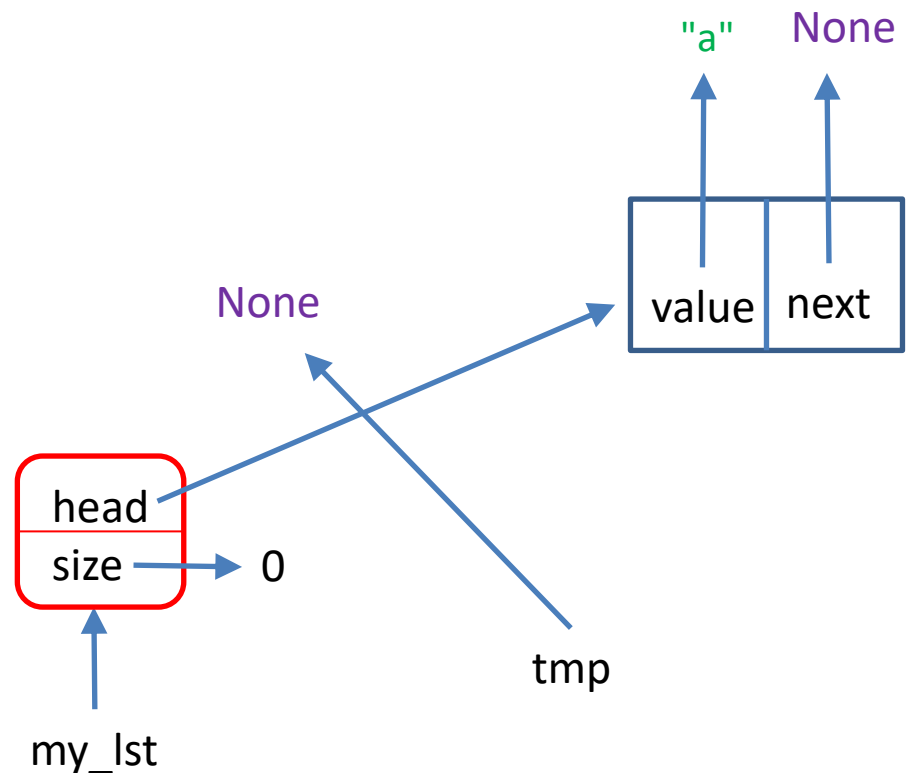
```
>>> my_lst.add_at_start("a")
```



Memory View (2)

```
def add_at_start(self, val):  
    tmp = self.head  
    self.head = Node(val)  
    self.head.next = tmp  
    self.size += 1
```

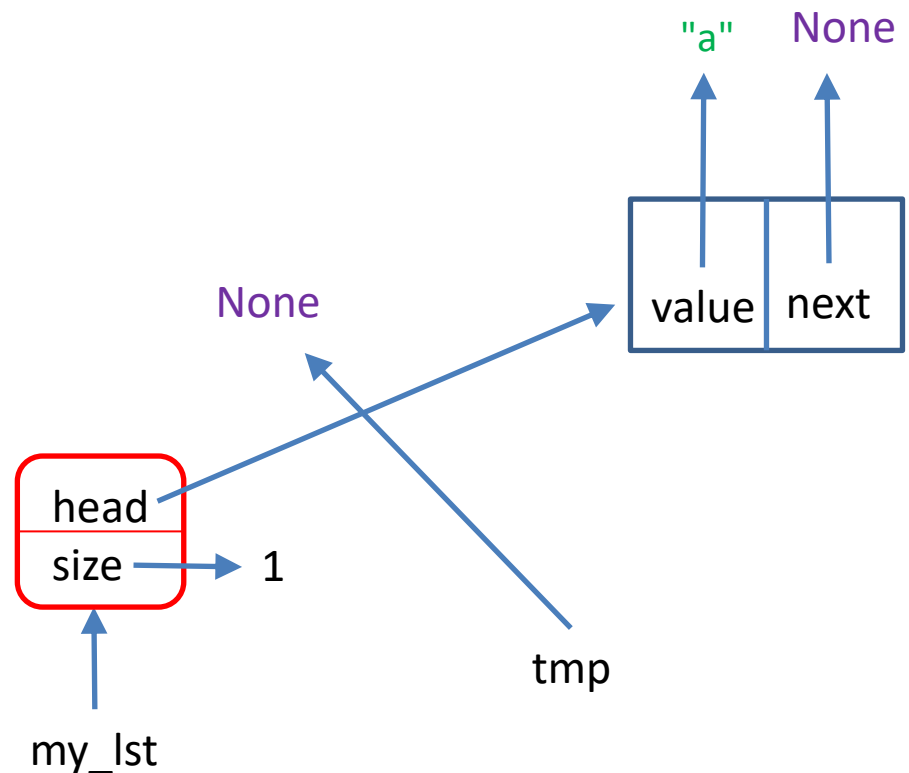
```
>>> my_lst.add_at_start("a")
```



Memory View (2)

```
def add_at_start(self, val):  
    tmp = self.head  
    self.head = Node(val)  
    self.head.next = tmp  
    self.size += 1
```

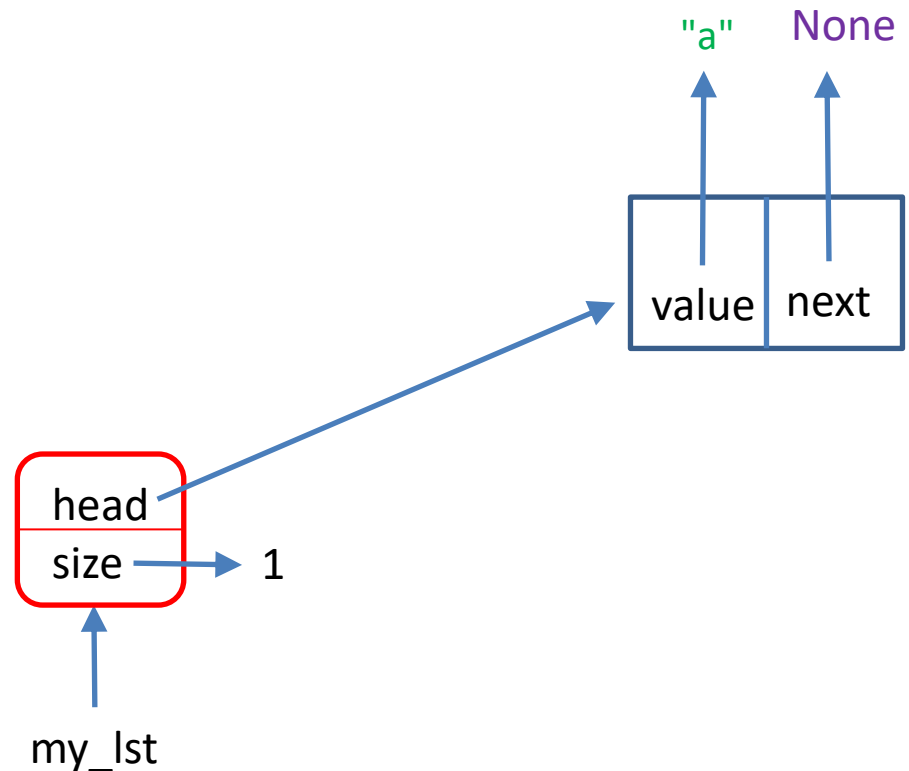
```
>>> my_lst.add_at_start("a")
```



Memory View (end of first insert)

```
def add_at_start(self, val):  
    tmp = self.head  
    self.head = Node(val)  
    self.head.next = tmp  
    self.size += 1
```

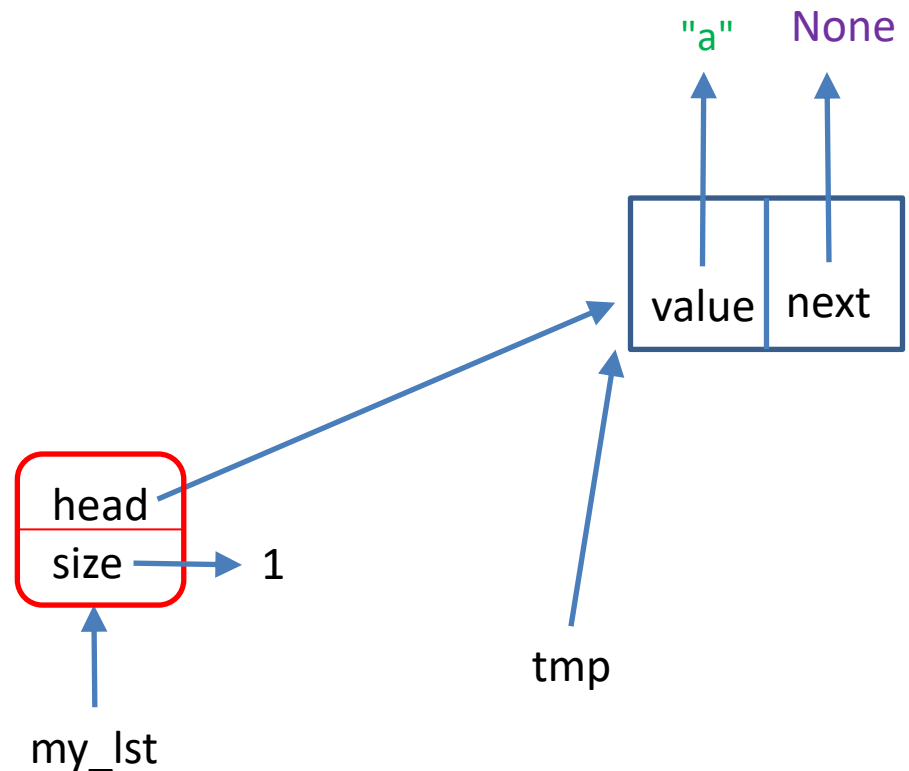
```
>>> my_lst.add_at_start("a")
```



Memory View (2)

```
def add_at_start(self, val):  
    tmp = self.head  
    self.head = Node(val)  
    self.head.next = tmp  
    self.size += 1
```

```
>>> my_lst.add_at_start("b")
```

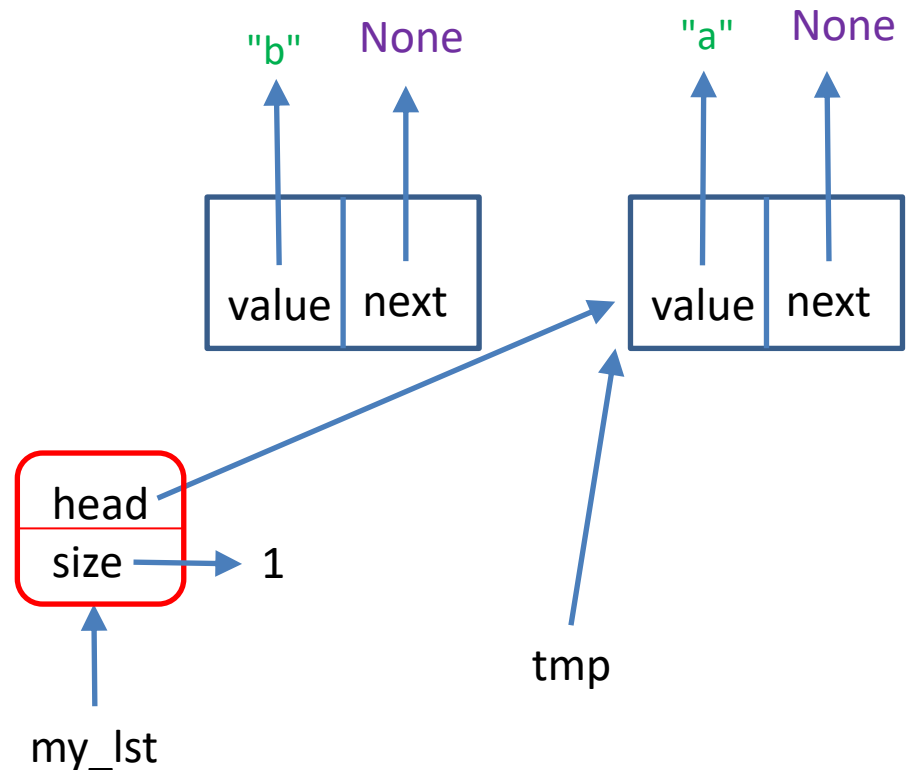


Memory View (2)

```
def add_at_start(self, val):  
    tmp = self.head  
    self.head = Node(val)  
    self.head.next = tmp  
    self.size += 1
```

```
class Node:  
    def __init__(self, val):  
        self.value = val  
        self.next = None
```

```
>>> my_lst.add_at_start("b")
```

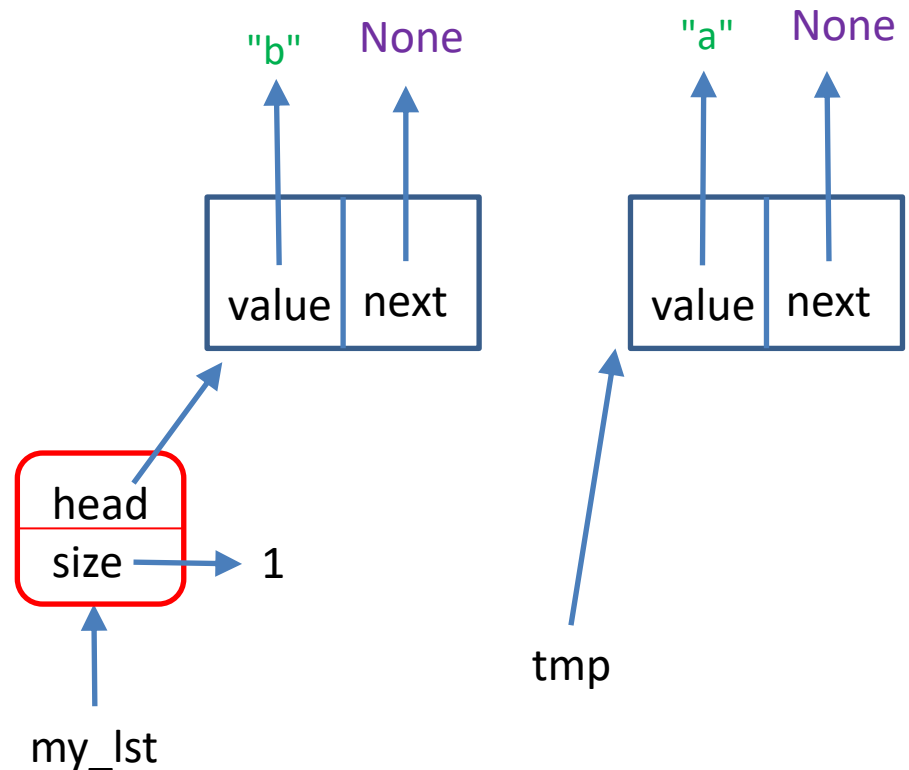


Memory View (2)

```
def add_at_start(self, val):  
    tmp = self.head  
    self.head = Node(val)  
    self.head.next = tmp  
    self.size += 1
```

```
class Node:  
    def __init__(self, val):  
        self.value = val  
        self.next = None
```

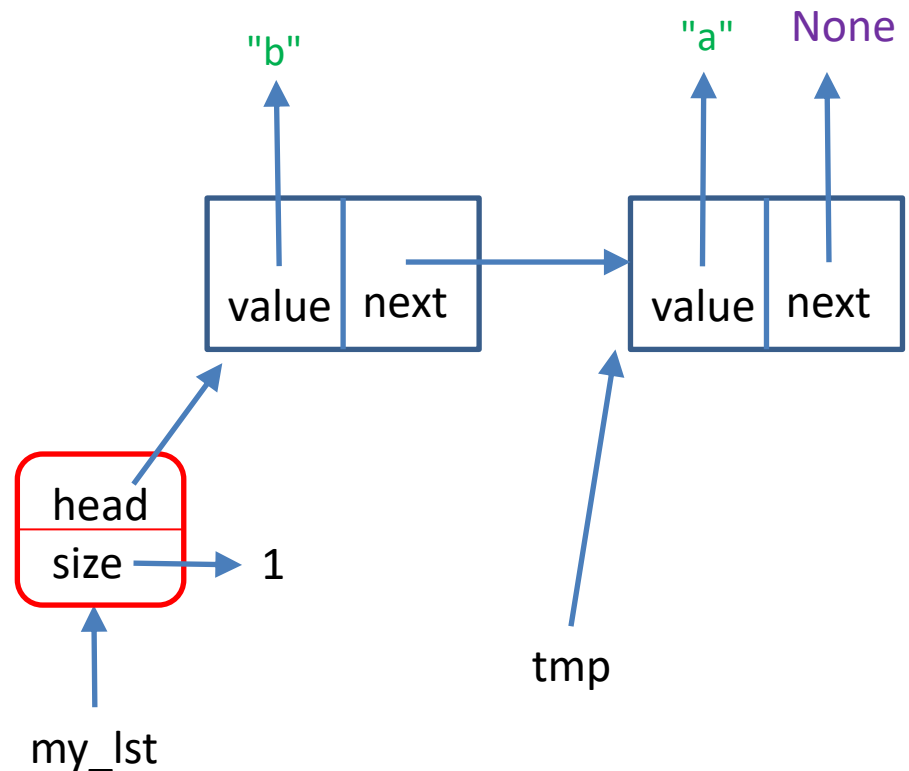
```
>>> my_lst.add_at_start("b")
```



Memory View (2)

```
def add_at_start(self, val):  
    tmp = self.head  
    self.head = Node(val)  
    self.head.next = tmp  
    self.size += 1
```

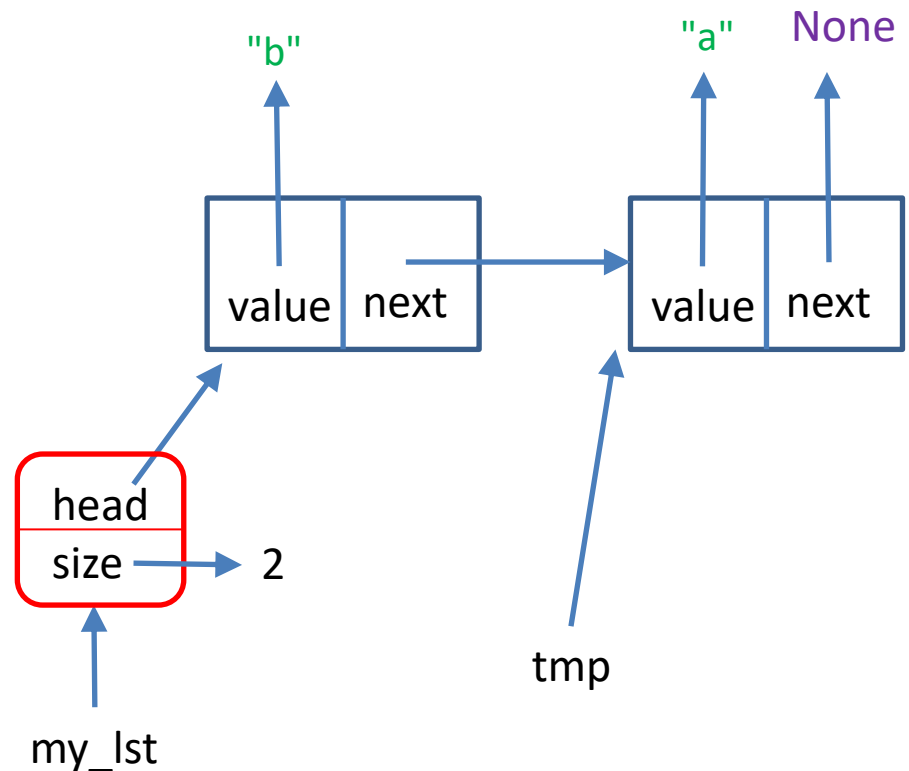
```
>>> my_lst.add_at_start("b")
```



Memory View (3)

```
def add_at_start(self, val):  
    tmp = self.head  
    self.head = Node(val)  
    self.head.next = tmp  
    self.size += 1
```

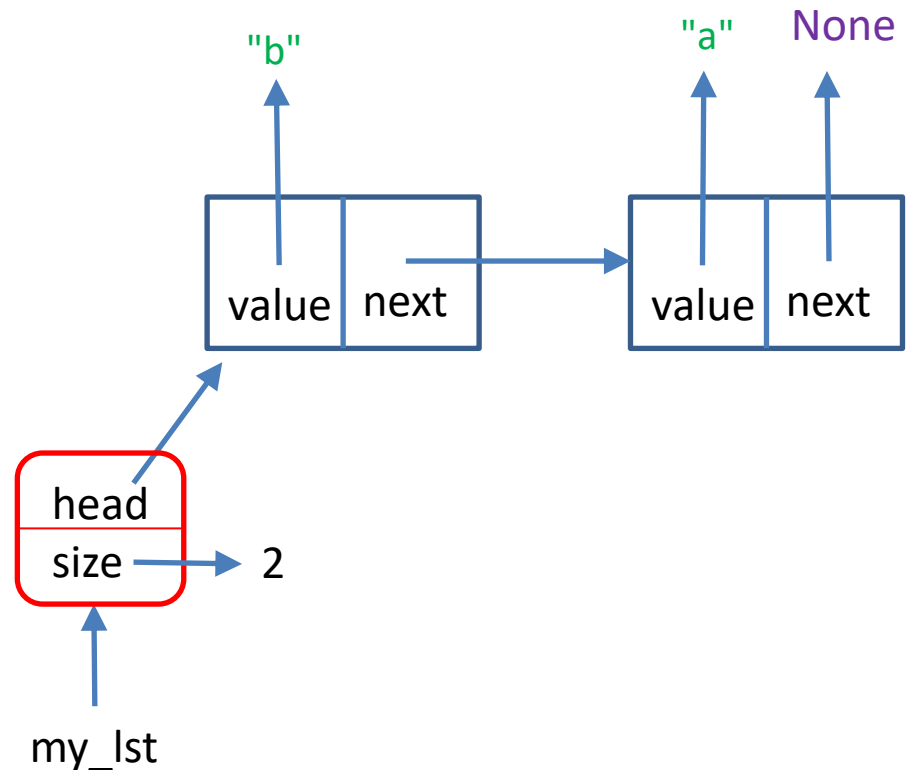
```
>>> my_lst.add_at_start("b")
```



Memory View (end of second iteration)

```
def add_at_start(self, val):  
    tmp = self.head  
    self.head = Node(val)  
    self.head.next = tmp  
    self.size += 1
```

```
>>> my_lst.add_at_start("b")
```

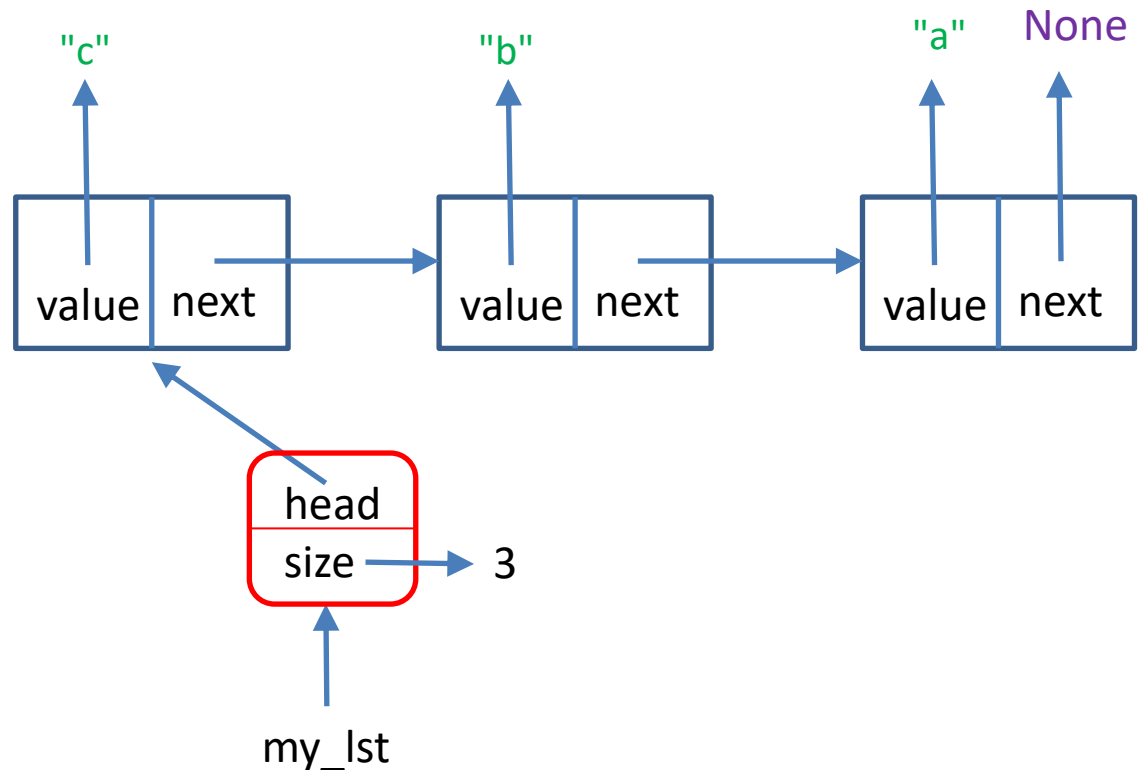


Memory View (4)

```
def add_at_start(self, val):  
    tmp = self.head  
    self.head = Node(val)  
    self.head.next = tmp  
    self.size += 1
```

```
class Node():  
    def __init__(self, val):  
        self.value = val  
        self.next = None
```

```
>>> my_lst.add_at_start("c")
```

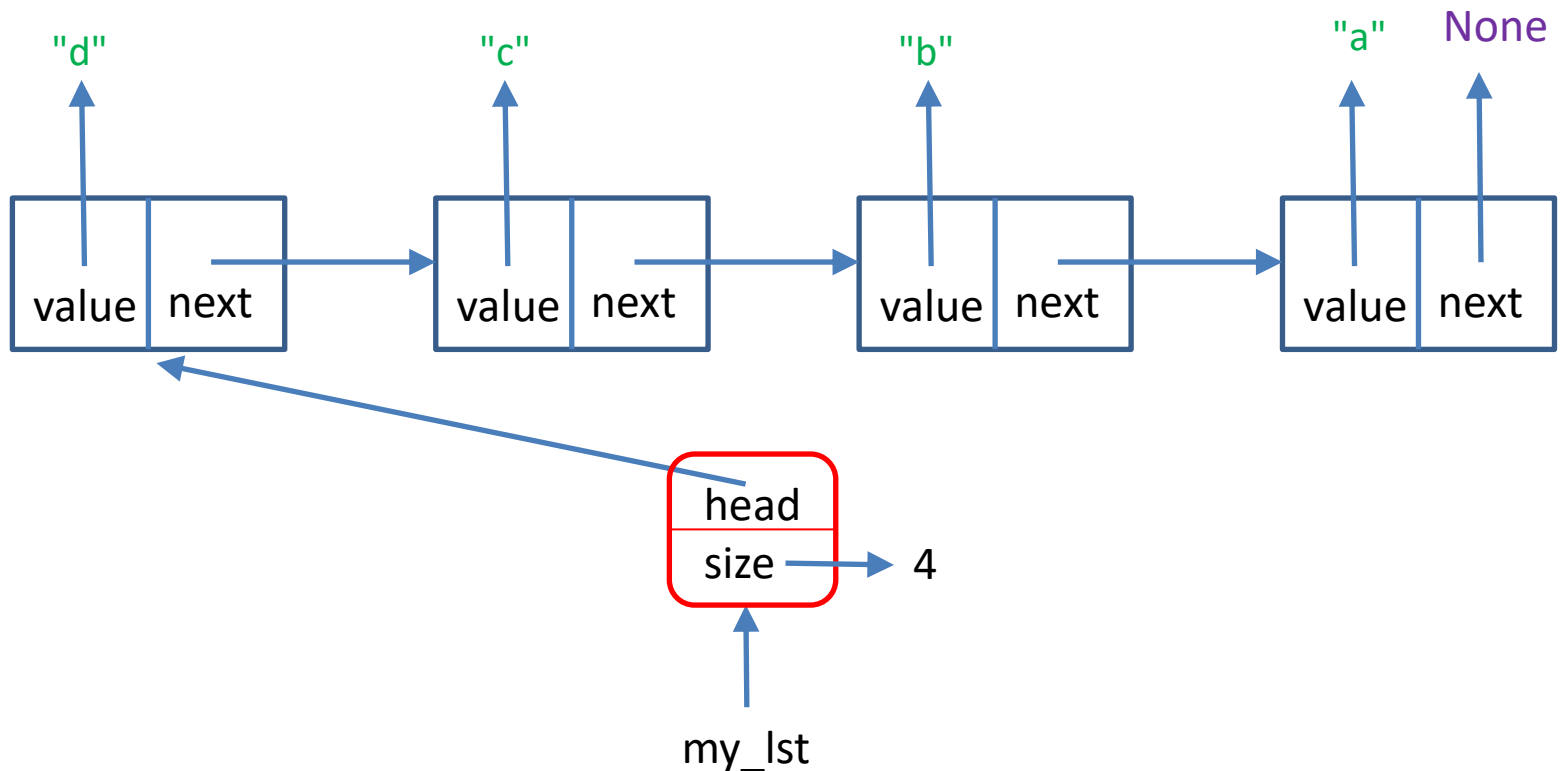


Memory View (5)

```
def add_at_start(self, val):  
    tmp = self.head  
    self.head = Node(val)  
    self.head.next = tmp  
    self.size += 1
```

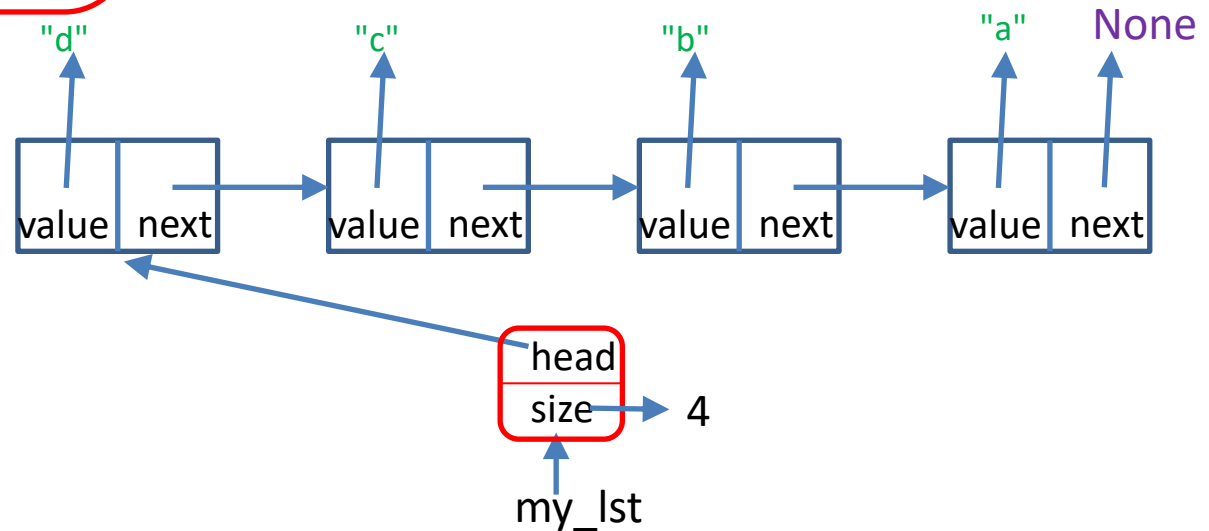
```
class Node():  
    def __init__(self, val):  
        self.value = val  
        self.next = None
```

```
>>> my_lst.add_at_start("d")
```



Linked List Operations : `__repr__`

```
def __repr__(self):  
    out = ""  
    p = self.head  
    while p != None :  
        out += p.__repr__() + ", "  
        p = p.next  
    return "[" + out[:-2] + "]"
```



```
>>> print(my_lst)    #calls __repr__ of class Linked_list  
[d, c, b, a]
```

Linked List Operations:

length

```
def __len__(self):  
    return self.size
```

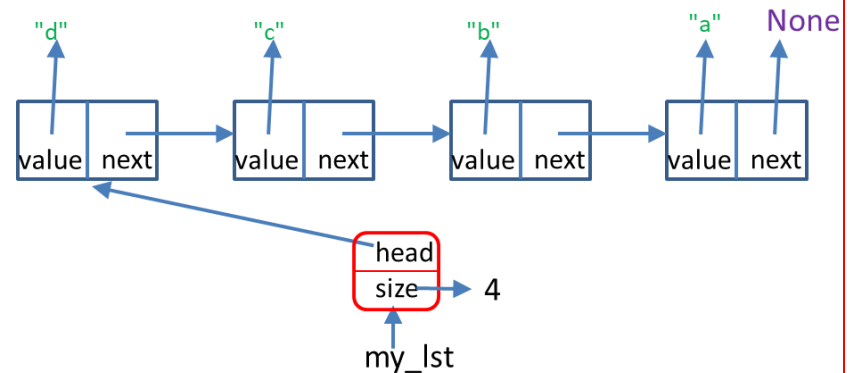
called when using Python's len()

```
>>> len(my_lst)  
4  
>>> my_lst.__len__() #same same  
4  
>>> my_lst.size      #same same, direct access to the data  
4
```

- The time complexity is $O(1)$
- But recall the field `size` must be updated when inserting / deleting elements

Linked List Operations: Index

```
def find(self, val):  
    ''' find index of (first) node with value val in list  
        return None if not found '''  
    p = self.head  
    i = 0      # we want to return the location  
    while p != None:  
        if p.value == val:  
            return i  
        else:  
            p = p.next  
            i += 1  
    return None # not found
```



- Time complexity: worst case $O(n)$, best case $O(1)$

Tip: Use PythonTutor

- [Link](#)
- For better visualization, choose the following parameters:

Visualize Execution

hide exited frames [default] ▾

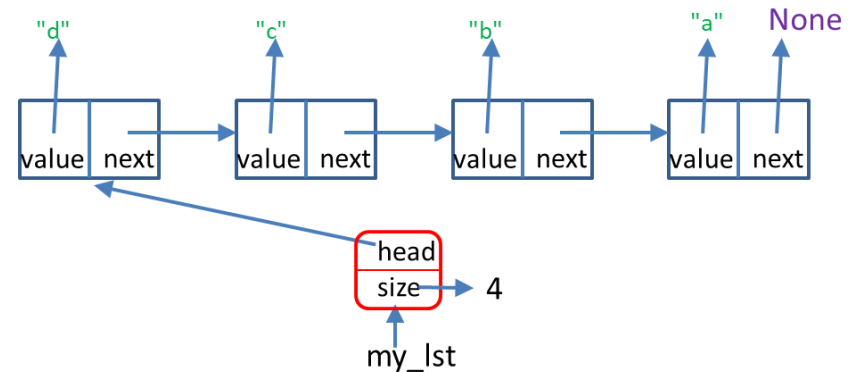
inline primitives, don't nest objects [default] ▾

draw pointers as arrows [default] ▾

Special Standard Method `__getitem__`

```
def __getitem__(self, i):  
    assert 0 <= i < len(self)  
    p = self.head  
    for j in range(0, i):  
        p = p.next  
    return p.value
```

called when using `L[i]` for **reading**



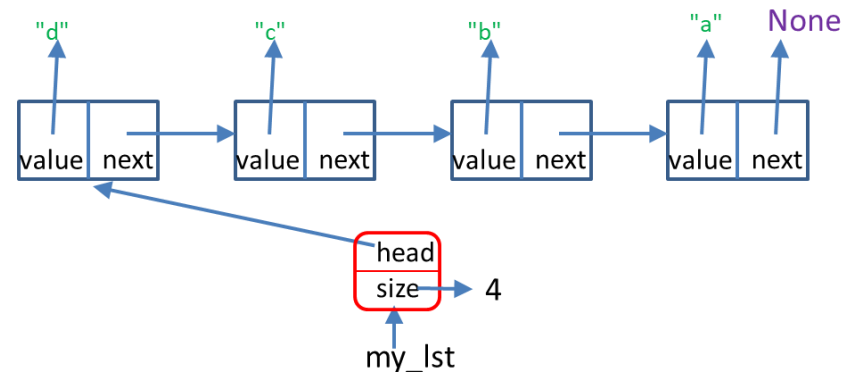
```
>>> my_lst[2]  
'b'  
>>> my_lst.__getitem__(2) #same same  
'b'  
>>> my_lst.head.next.next.value #same same but don't!  
'b'
```

- The argument `i` must be between 0 and the length of the list (otherwise `assert` will notify an error).
- Time complexity: $O(i+1)$. In the worst case ($i = n-1$) this is $O(n)$.

Special Standard Method `__setitem__`

```
def __setitem__(self, i, val):  
    assert 0 <= i < len(self)  
    p = self.head  
    for j in range(0, i):  
        p = p.next  
    p.value = val  
    return None
```

called when using `L[i]` for **writing**



```
>>> my_lst[1] = 999 #same as my_lst.__setitem__(1, 999)  
>>> print(my_lst)  
[d, 999, b, a]
```

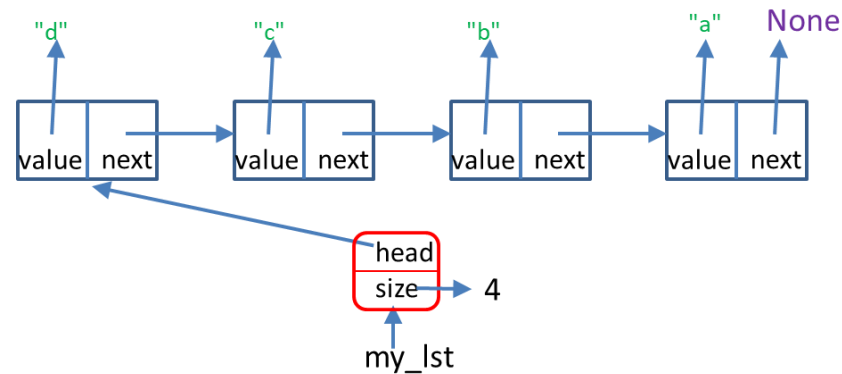
- The argument `i` must be between 0 and the length of the list (otherwise `assert` will notify an error).
- Time complexity: $O(i+1)$. In the worst case ($i = n-1$) this is $O(n)$.

Linked List Operations:

Insertion at a Given Location

```
def insert(self, i, val):  
    assert 0 <= i <= len(self)  
    if i == 0:  
        self.add_at_start(val)  
    else:  
        p = self.head  
        for j in range(0, i-1):  
            p = p.next  
        tmp = p.next  
        p.next = Node(val)  
        p.next.next = tmp  
        self.size += 1
```

```
def add_at_start(self, val):  
    tmp = self.head  
    self.head = Node(val)  
    self.head.next = tmp  
    self.size += 1
```



- Note: elements **after index i** implicitly move **one position forward**
- When **i=0** we get the same effect as **add_at_start**, which updated the list head. Note that **i=n** is allowed.
- Time complexity: **O(i+1)**. In the worst case (**i = n**) this is **O(n)**.

Linked List Operations:

Deletion at a Given Location

```
def pop(self, i):  
    ''' delete element at location i '''  
    assert 0 <= i < len(self)  
    if i == 0:  
        self.head = self.head.next  
    else:  
        p = self.head  
        for j in range(0, i-1):  
            p = p.next  
        # p is the element BEFORE i'th  
        p.next = p.next.next  
  
    self.size -= 1
```

- Python *Garbage collector* will “remove” the deleted item (assuming there is no other reference to it) from memory.
- Note: In some languages (e.g. C, C++) the programmer is responsible to explicitly free unused memory

- Note: elements after index i implicitly move one position backward
- When $i=0$, list head must be updated
- Time complexity: $O(i+1)$. In the worst case ($i = n-1$) this is $O(n)$.

Comment on Deletion by Value

- How would you delete an item with a given **value** (not location)?
- Searching and then deleting the found item presents a (small) **technical inconvenience**: in order to delete an item, we need access the item **before** it.
- A possible solution would be to keep a **2-directional linked list**, aka **doubly** linked list (each node points both to the **next** node and to the **previous** one).
 - This requires, however, $O(n)$ additional memory (compared to a 1-directional linked list).



An Extended `__init__`

- Suppose we wanted to allow the initialization of a `Linked_list` object that will not be initially empty. Instead, it will contain an existing Python's `sequence` (e.g. list, string, tuple) upon initialization.

```
class Linked_list:
    def __init__(self, seq=None):
        self.head = None
        self.len = 0
        if seq != None:
            for ch in seq[::-1]:
                self.add_at_start(ch)
```

```
>>> L = Linked_list("abc")
>>> print(my_lst)
[a, b, c]
```

- We employ `add_at_start(ch)` for efficiency reasons, as each such `insertion` takes only $O(1)$ operations, and overall $O(\text{len}(\text{seq}))$.
- Additionally, we could easily avoid `slicing` (used here to reverse)

Linked Lists vs. Python Lists:

Complexity Summary

Operation	Worst case time complexity – Linked lists	Worst case time complexity – Python lists
Insertion after a given element (or at start)	$O(1)$	$O(n)$
Insertion at position i	$O(n)$	$O(n)$
Get or modify the i 'th element	$O(n)$	$O(1)$
Delete, given previous element	$O(1)$	$O(n)$
Delete at position i	$O(n)$	$O(n)$