Extended Introduction to Computer Science CS1001.py

Lecture 16: Linked Lists

Instructors: Benny Chor, Amir Rubinstein

Teaching Assistants: Michal Kleinbort, Amir Gilad

School of Computer Science
Tel-Aviv University
Spring Semester 2017
http://tau-cs1001-py.wikidot.com

Lecture 15 Highlights

- GCD (integer greatest common divisor)
- OOP (object oriented programming)

Lecture 16 and 17 (!) - Plan

- Data Structures
- Lecture 16 (i.e., today)
  - Linked Lists
- Lecture 17 (i.e., Wednesday)
  - Binary Search Trees
  - Hash Tables

Data Structures

- A data structure is a way to organize data in memory (or other storage media), as to support various operations.

- Operations are divided into two types:
  - Queries, like search, finding minimum, etc.
  - Mutations, like insertion, deletion, modification.

- 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.

- The choice of data structures for a particular problem depends on desired operations and complexity constraints (time and memory).

- The term Abstract Data Type (ADT) emphasizes the point that the user (client) needs to know what operations may be used, but not how they are implemented.

Data Structures (cont.)

- Next, we will implement a new data structure, called Linked List, and compare it to Python's built-in list structure.
- Next class, we will discuss another linked structure, Binary Search Trees.
- Later in the course we will see additional non-built-in data structures implemented by us (and you), *e.g.* hash tables and matrices.

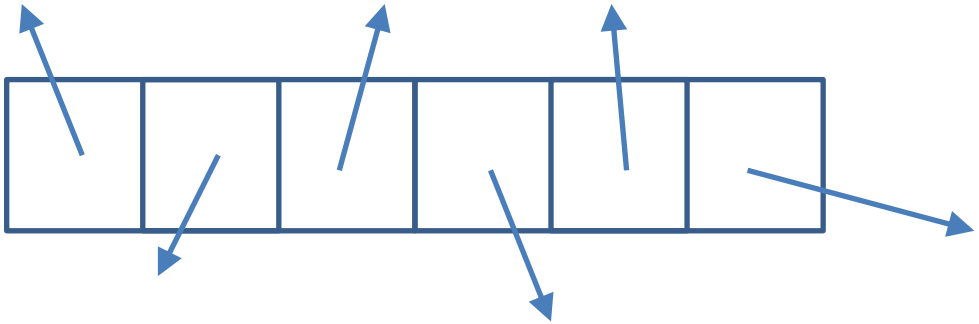
Reminder: Representing Lists in Python

- We have extensively used Python's built-in list object.

- “Under the hood", Python lists employ C's array. This means it uses a contiguous array of pointers: references to (addresses of) other objects.

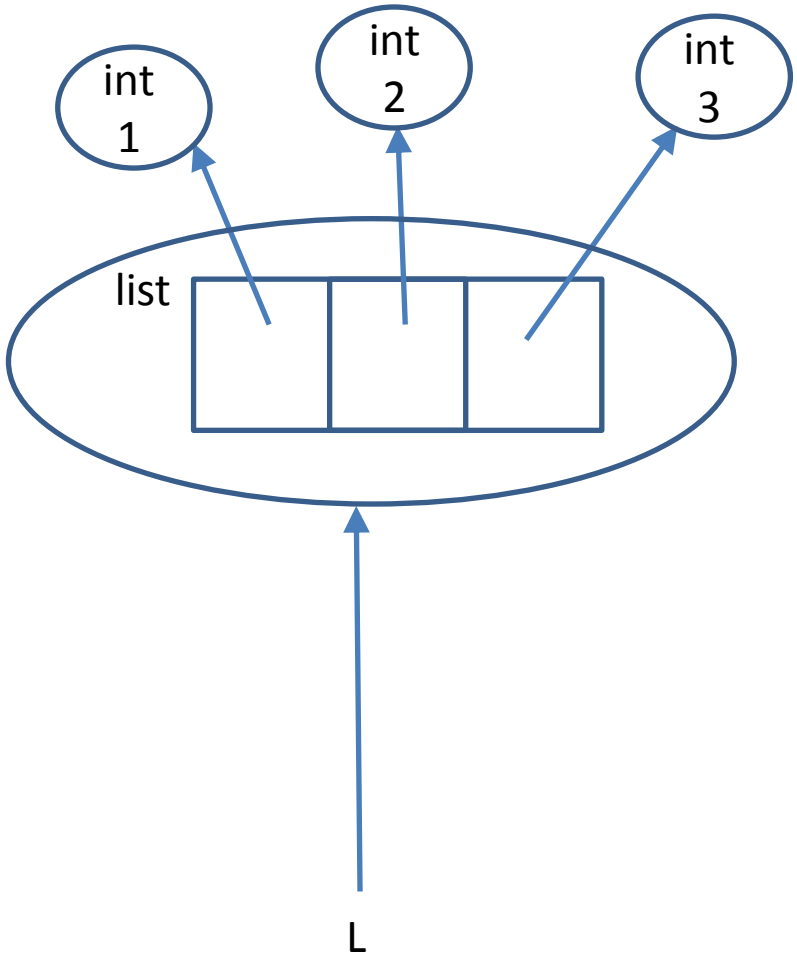
- Python keeps the address of this array in memory, and its length in a list head structure.
- This makes accessing/modifying a list element, a[i], an operation whose cost is O(1) - independent of the size of the list or the value of the index:

  if the address in memory of lst[0] is *a*, then the address in memory of lst[i] is simply *a*+*i*. 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 other programming languages “naked arrays”).



Reminder: Representing Lists in Python

```
>>> L = [1,2,3]
```



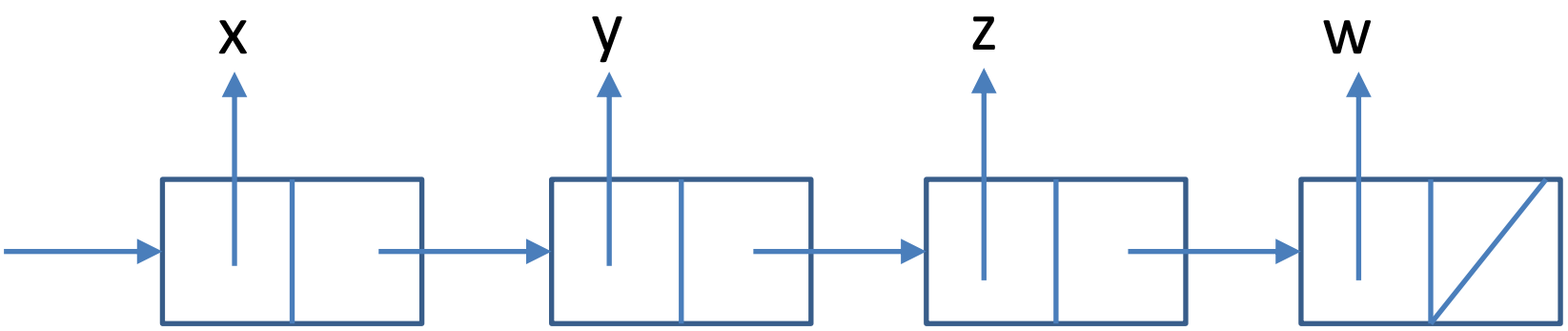
Reminder: Representing Lists in Python, cont.

- However, the contiguous storage of addresses must be maintained when the list evolves.
- In particular if we want to insert an item at location i, all items from location i onwards must be "pushed" forward, leadsing to O(n) operations many 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).
- 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.
- Official source: How are lists implemented?

Linked Lists

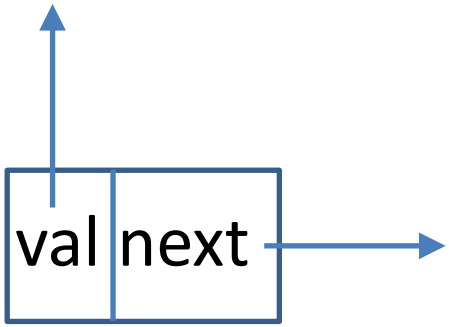
- An alternative to using a contiguous block of memory, is to specify, for each item, the memory location of the next item in the list.

- We can represent this graphically using a boxes-and-pointers diagram:



Linked Lists Representation

- We introduce two classes. One for nodes in the list, and another one to represent a list.
- 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) + "," + str(id(self.next))+ "]"
        #  showing pointers as well (for educational purposes)
```

Linked List class

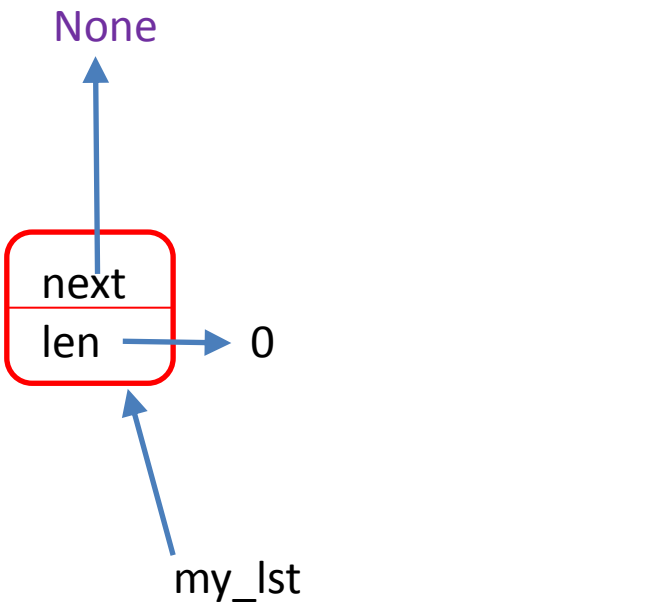
```
class Linked_list():
    def __init__ (self):
        self.next = None   # using same field name as in Node
        self.len = 0

    def __repr__(self):
        out = ""
        p = self.next
        while  p != None :
              out += str(p) + "  " # str(p) envokes __repr__ of class Node
              p = p.next
        return out
```

More methods will be presented in the next slides.

Memory View

```
class Linked_list():
    def __init__(self):
        self.next = None
        self.len = 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 '''
    p = self
    tmp = p.next
    p.next = Node(val)
    p.next.next = tmp
    self.len += 1
```

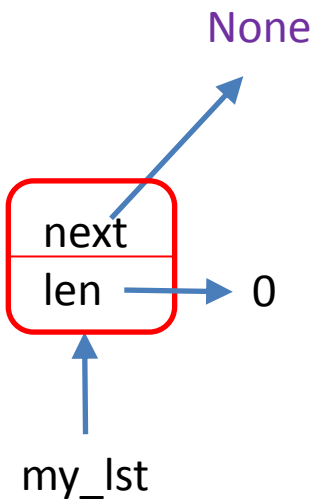
- Note: time complexity is O(1) in the worst case!

Memory View (1)

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

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

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

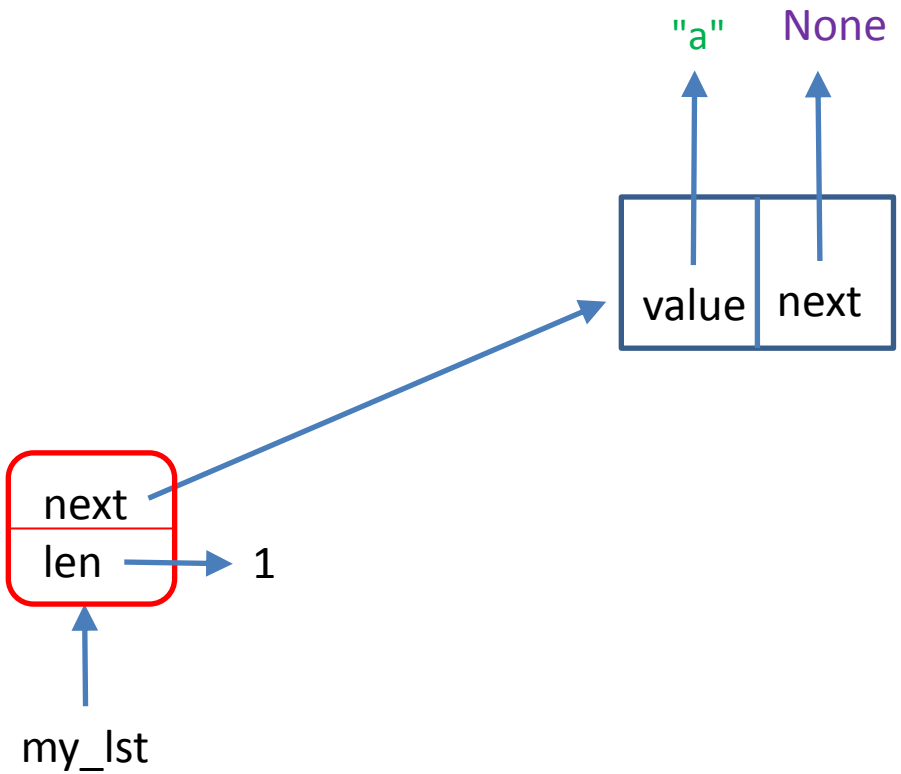


Memory View (2)

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

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

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

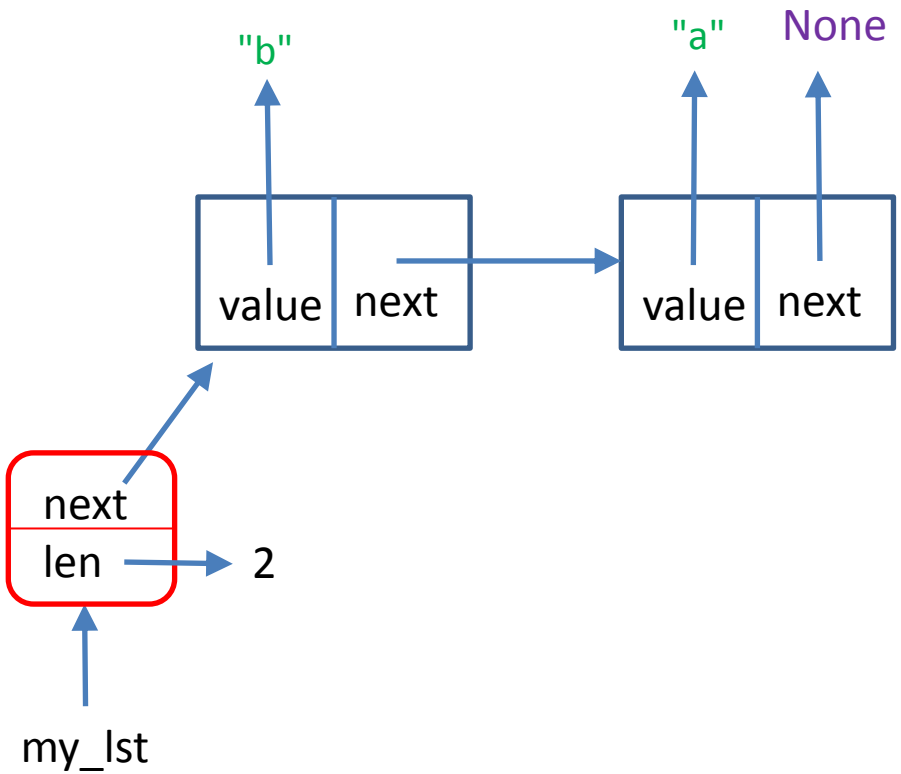


Memory View (3)

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

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

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

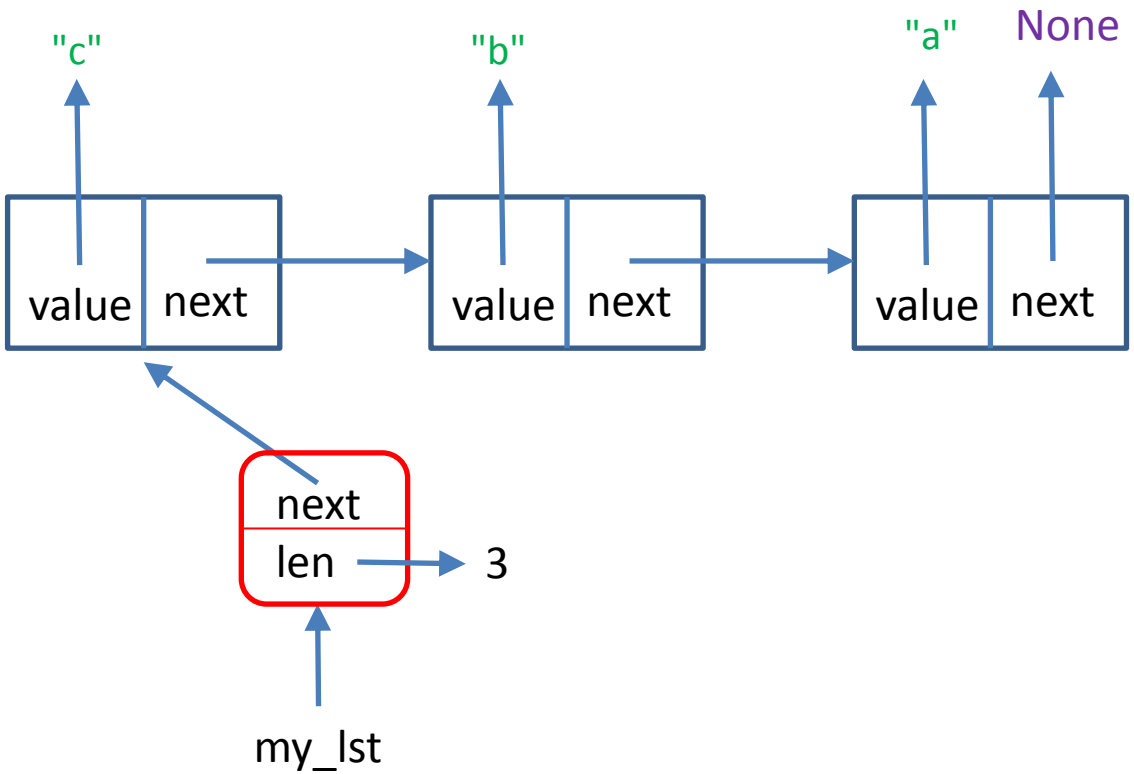


Memory View (4)

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

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

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

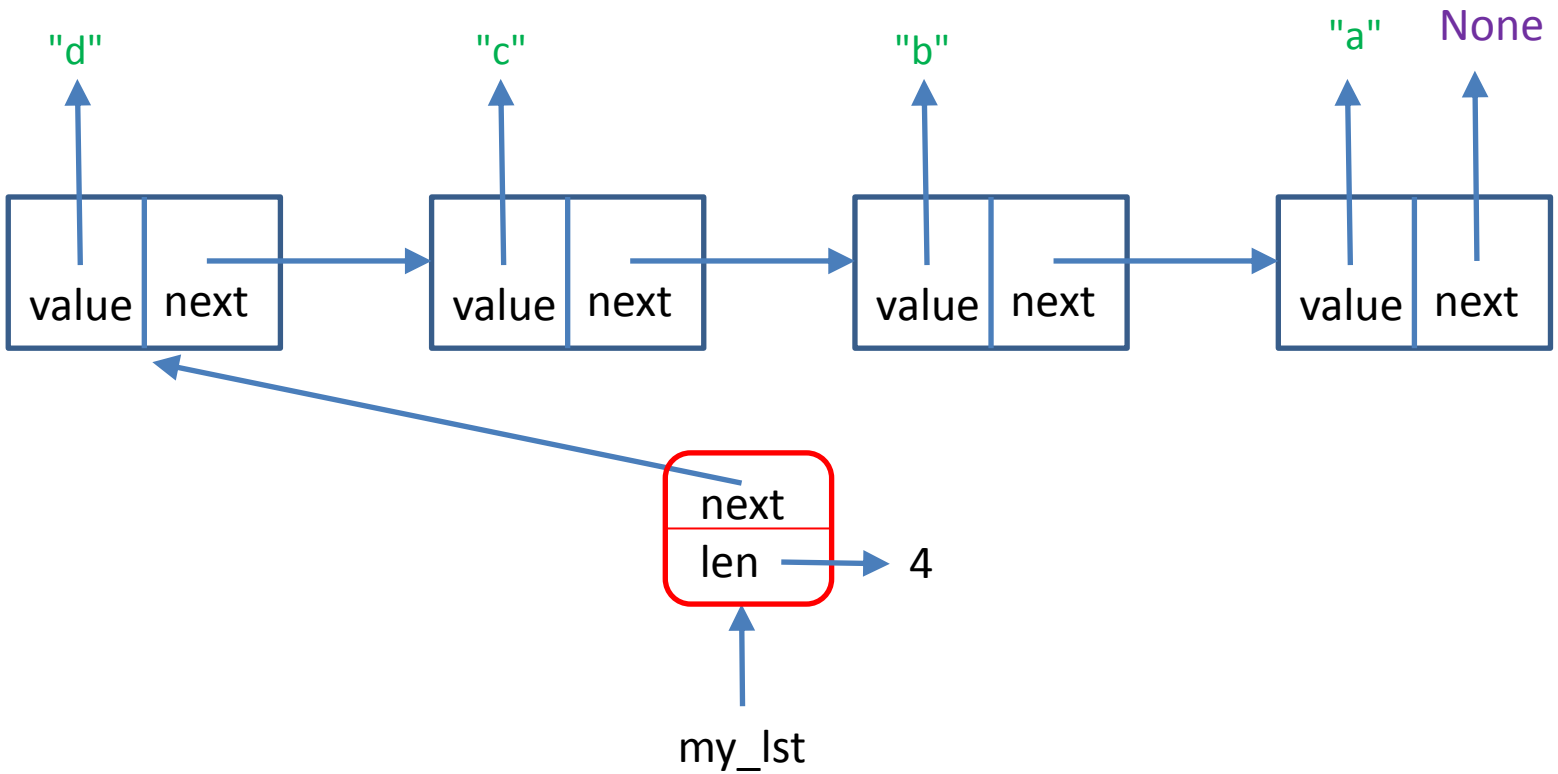


Memory View (5)

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

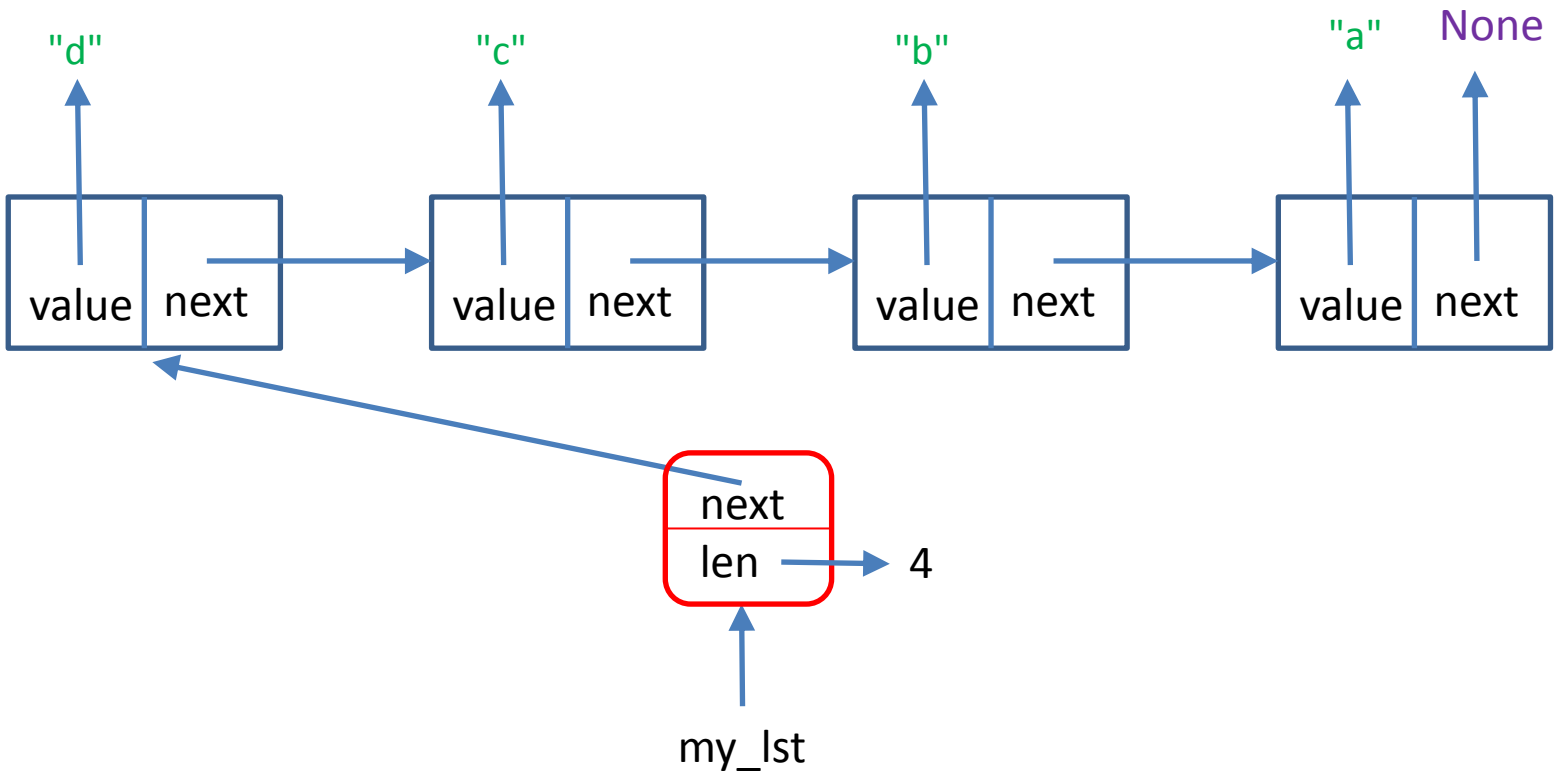
```
class Node():
    def __init__(self, val):
        self.value = val
        self.next = None
```

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



Memory View (6)

```
>>> print(my_lst)  #calls __repr__ of class Linked_list

[d,44602768] [c,44602736] [b,44602096] [a,1749231768]

>>> id(None)
1749231768
```



Linked List Operations: length

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

called when using Python's len()

```
>>> len(my_lst)
4
>>> my_lst.__len___() #same
4
```

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

Linked List Operations: Indexing

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

called when using L[i] for reading

```
>>> my_lst[1]
'c'
>>> my_lst.__getitem(1) #same
'c'
```

- The argument loc must be between 0 and the length of the list (otherwise assert will cause an exception).

- Time complexity: O(loc). In the worst case loc = n.

Linked List Operations: Indexing (2)

```
def __setitem__(self, loc, val):
    assert 0 <= loc < len(self)
    p = self.next
    for i in range(0, loc):
        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,44602768] [999,44602736] [b,44602096] [a,1749231768]
```

- The argument loc must be between 0 and the length of the list (otherwise assert will cause an exception).

- Time complexity: O(loc). In the worst case loc = n.

Linked List Operations: Insertion at a Given Location

```
def insert(self, val, loc):
    assert 0 <= loc < len(self)
    p = self
    for i in range (0, loc):
        p = p.next
    tmp = p.next
    p.next = Node(val)
    p.next.next = tmp
    self.len += 1
```

Compare to:

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

- The argument loc must be between 0 and the length of the list (otherwise assert will cause an exception)
- When loc is 0, we get the same effect as add_at_start
- Time complexity: O(loc). In the worst case loc = n.

Linked List Operations: Find

```
def find(self, val):
    p = self.next
    #loc = 0        #   in case we want to return the location
    while  p != None:
        if  p. value == val:
            return p
        else :
            p = p.next
            #loc = loc +1 # in case we want to return the location
    return None
```

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

Linked List Operations: Delete

```
def delete(self, loc):
    ''' delete element at location loc '''
    assert 0 <= loc < len(self)
    p = self
    for i in range(0, loc):
        p = p.next
    if p!= None and p.next != None:
        p.next = p.next.next
    self.len -= 1
```

- The argument loc must be between 0 and the length.
- Time complexity: O(loc). In the worst case loc = n.

- 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 ask for the memory to be freed.

Linked List Operations: Delete

- 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 to 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).
- You may encounter this issue again in the next HW assignment.

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.next = None
        self.len = 0
        if seq != None:
            for ch in seq[::-1]:
                self.add_at_start(ch)
```

- We employ add_at_start(ch) for efficiency reasons, as each such insertion takes only O(1) operations.

```
>>> L = Linked_list("abc")
>>> print(my_lst)
[a,42430064] [b,42430032] [c,1749231768]
```

Accessing Fields Directly

Goal in example:

- Search for a certain item, and if found, increment it:

```
x = lst.find(3)
if x != None:
    x.value += 1
```

- Actually, this is a bad practice, which we strongly discourage.
- It is way better to use a designated method, like __setitem__.

Linked Lists vs. Python Lists: Complexity of Operations

- When we have a pointer to an element, inserting an element after it requires just O(1) operations (in particular when inserting at the start). Compare to O(n) for python lists.

- Deletion of a given item requires O(1) time, assuming we have access to the previous item. Compare to O(n) for Python lists.

- Accessing the i-th item requires O(i) time. Compare to O(1) for Python lists.

Sorted, Linked Lists

- So far we treated unordered lists. We will now consider sorted linked lists. What would be improved this way? What would not?
- We can maintain an ordered linked list, by always inserting an item in its correct location. The version below allows duplicates.

```
def insert_ordered(self, val):
    p = self
    q = self.next
    while q != None and q.value < val:
        p = q
        q = q.next
    newNode = Node(val)
    p.next = newNode
    newNode.next = q
    self.len += 1
```

Searching in an Ordered linked list

- We cannot use binary search to look for an element in an ordered linked list.
- This is because random access to the i'th element is not possible in constant time in linked lists (as opposed to Python's lists).

```
def find_ordered(self, val):
    p = self.next
    while p != None and p.value < val:
        p = p.next
    if p != None and p.value == val:
        return p
    else:
        return None
```

- We leave it to you to write a delete method for ordered lists.

Perils of Linked Lists

- With linked lists, we are in charge of memory management, and if not careful, we may introduce cycles:

```
>>> L = Linked_list()
>>> L.next = L
>>> L  #What do you expect to happen?
```

- Can we check if a given list includes a cycle?
- Here we assume a cycle may only occur due to the "next" pointer pointing to an element that appears closer to the head of the structure.

- But cycles may also occur due to the "content" field.

Detecting Cycles: First Variant

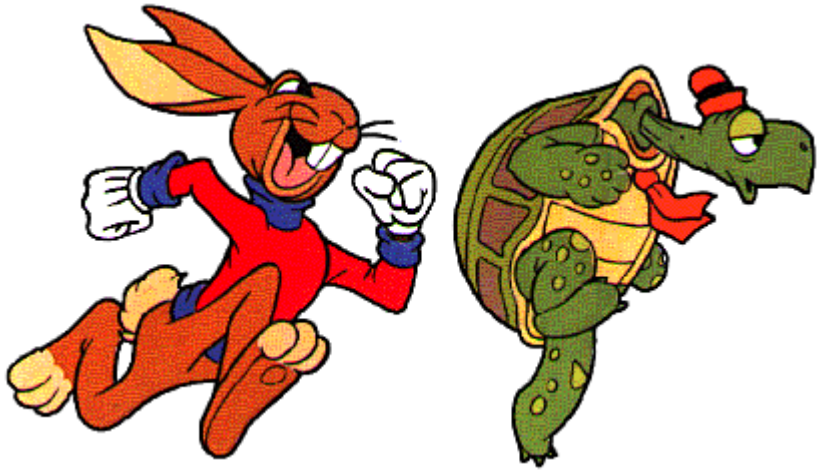
```
def detect_cycle1(self):
    s= set() #like dict, but only keys
    p = self
    while True :
        if p == None:
            return False
        if p in s:
            return True
        s.add(p)
        p = p. next
```

- Note that we are adding the whole list element (“box") to the dictionary, and not just its contents.
- Can we do it more efficiently?
- In the worst case, we may have to traverse the whole list to detect a cycle, so O(n) time in the worst case is inherent.
- But can we detect cycles using just O(1) additional memory?

Detecting cycles: Bob Floyd's Tortoise and the Hare Algorithm (1967)

The hare moves twice as quickly as the tortoise. Eventually they will both be inside the cycle. The distance between them will then decrease by 1 at each additional step.
When this distance becomes 0, they are on the same point on the cycle.

See demo on board.



Detecting cycles: The Tortoise and the Hare Algorithm

```
def detect_cycle2(self):
    # The hare moves twice as quickly as the tortoise
    # Eventually they will both be inside the cycle
    # and the distance between them will increase by 1 until
    # it is divisible by the length of the cycle .
    slow = fast = self
    while True :
        if slow == None or fast == None:
            return False
        if fast.next == None:
            return False
        slow = slow.next
        fast = fast.next.next
        if slow is fast:
            return True
```

The same idea is used in Pollard's rho algorithm for factoring integers in $O(2^{n/4})$ expected time.

Testing the cycle algorithms

The Python file contains a function introduces a cycle in a list.

```
>>> lst = Linked_list("abcde")
>>> lst
[a,42422928] [b,34885200] [c,34881648] [d,42430032]
[e,1749231768]
>>> detect_cycle1(lst)
False

>>> create_cycle(lst,2,4)
>>> detect_cycle1(lst)
True
>>> detect_cycle2(lst)
True
```

Cycles in “Regular" Python Lists

Mutations may introduce cycles in Pythons lists as well. In this example, either append or assign do the trick.

```
>>> lst =["a","b","c","d","e"]
>>> lst.append(lst)
>>> lst
['a', 'b', 'c', 'd', 'e', [...]]
>>> lst =["a","b","c","d","e"]
>>> lst [3]= lst
>>> lst
['a', 'b', 'c', [...] , 'e']
>>> lst [1]= lst
>>> lst
['a', [...] , 'c', [...] , 'e']
```

We see that Python recognizes such cyclical lists and [...] is printed to indicate the fact.

Linked lists: Additional Issues

- Note that items of multiple types can appear in the same list.

- Some programming languages require homogenous lists (namely all elements should be of the same type).

Linked data structures

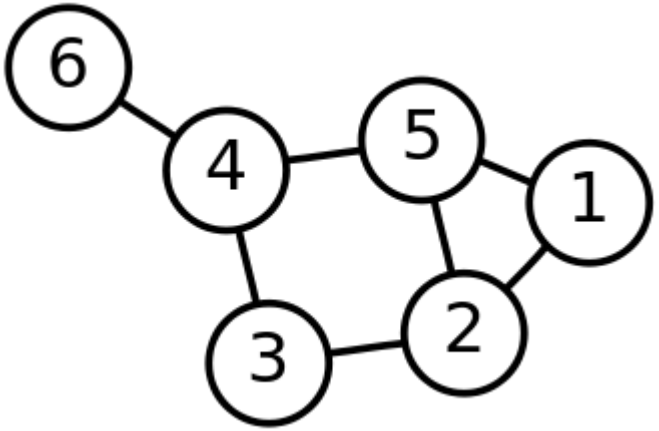
- Linked lists are just the simplest form of linked data structures, in which pointers are used to link objects.
- We can create structures of other forms.
- For example doubly-linked lists, whose nodes include a pointer from each item to the preceding one, in addition to the pointer to the next item.
- Another linked structure is binary trees, where each node points to its left and right child. We will see it now and also how it may be used to store and search data.
- Another linked structure is graphs (probably not in this course).

Graphs

- A graph is a structure with nodes (or vertices) and edges. An edge connects two nodes.
- In directed graphs, edges have a direction (go from one node to another).
- In undirected graphs, the edges have no direction.

Example: undirected graph.

Drawing from wikipedia



Trees

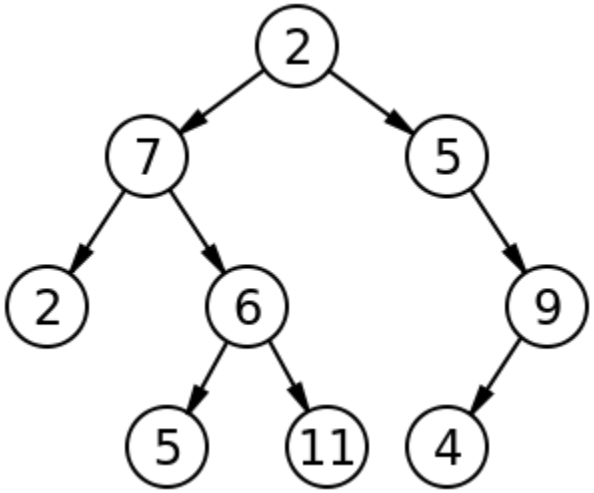
- Trees are useful models for representing various physical, biological, and other abstract structures.

- Trees may be defined as a special case of graphs. Basic properties of graphs and trees are discussed in the Discrete Mathematics course (and used in many courses).

- We will only discuss a specific form, the so called rooted trees.

Rooted Binary Trees

- Definition: a rooted binary tree
  - contains no nodes (empty tree), or
  - comprised of three disjoint sets of nodes:
    - a root node,
    - a binary tree called its left subtree, and
    - a binary tree called its right subtree



Adapted from wikipedia

- Note that this is a recursive definition.

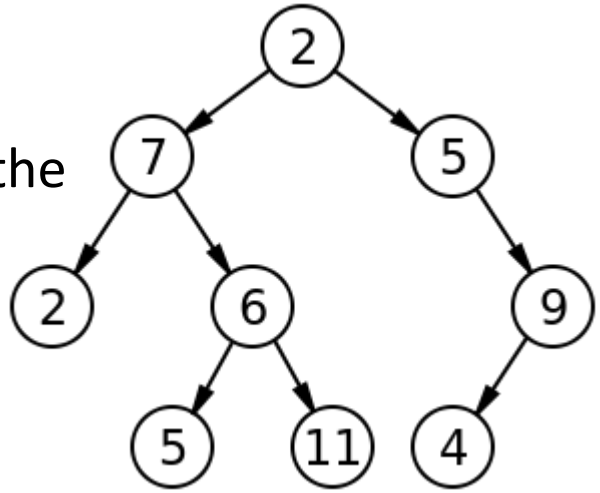
- Rooted binary trees are a special case of rooted trees, in which each node has at most 2 children.

Rooted Trees – Basic Notions

- A **directed edge** refers to the edge from the parent to the child (the arrows in the picture of the tree)
- The root node of a tree is the (unique) node with no parents (usually drawn on top).
- A leaf node has no children.
- Non leaf nodes are called internal nodes.

- The depth (or height) of a tree is the length of the longest path from the root to a node in the tree. A (rooted) tree with only one node (the root) has a depth of zero.

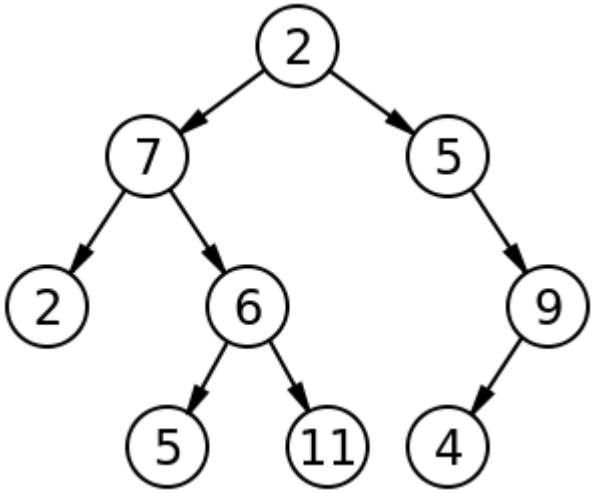
- A node *p* is an ancestor of a node *q* if *p* exists on the path from the root node to node *q*.
- The node *q* is then termed as a descendant of *p*.
- The out-degree of a node is the number of edges
- leaving that node.
- Leaf nodes have an out-degree 0.



Adapted from wikipedia

Example: Rooted Binary Tree

- Here the root is labeled 2. the depth of the tree is 3. Node 11 is a descendent of 7, but not of (either of the two nodes labeled) 5.



Drawing from wikipedia

- Note that this tree is not a complete binary tree. Not all leaves are at the same height.

And Now for Something Completely Different

אדון צלחת וגברת מי לימון

מילים: נתן אלתרמן
לחן: משה וילנסקי
כתיבה: 1946
הלחנה: לכל היאוחר 1932

סְלְחִי לִי מֵי, לִבִּי הִנֵּה!
אֲנִי לוֹקַחַת אוֹתוֹ צַלַּחַת.
סְלְחִי לִי מֵי, יָפָה כָּל כָּךְ אַתְּ!
אֲנִי סוֹלַחַת לָךְ צַלַּחַת.

אֶפְשָׁר לִחְיוֹת שָׁנִים בְּאֹפֶן אִידִיוֹטִי
בְּלִי אַוִּנְטוּרָה וְרוֹמַנְטִיקָה כָּזֹאתִ'י.
לִימוֹן, לִימוֹן – מַה יֵּשׁ צַלַּחַת?
אוּלַי נֵלֵכָה לְטַיֵּל מְעַט בְּיַחַד.

ביצוע: יונה עטרי ואילי גורליצקי

https://www.youtube.com/watch?v=JXm2sNThoCA