CSC148 winter 2015

linked lists, iteration, looping

mutation — week 8 → update objects.

Danny Heap

heap@cs.toronto.edu

BA4270 (behind elevators)

http://www.cdf.toronto.edu/~heap/148/W14/

416-978-5899

March 3, 2015

Outline

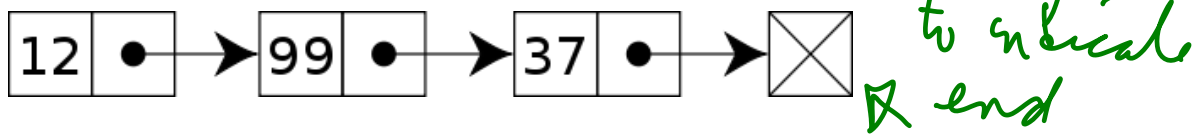
mutation

linked lists, two concepts

There are two useful, but different, ways of thinking of linked list structures

1. as lists made up of an item (value) and a sub-list (rest)

2. as objects (nodes) with a value and a reference to other similar objects



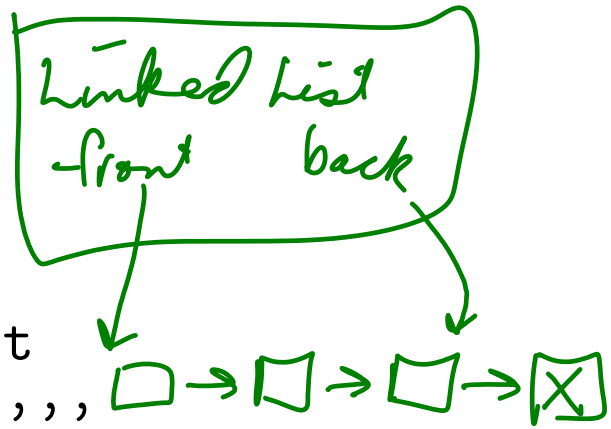
For now, will take the second point-of-view, and design a separate "wrapper" to represent a linked list as a whole.

a node class

```
class LLNode:
    '''Node to be used in linked list

    nxt: LLNode -- next node
                   None iff we're at end of list
    value: object --- data for current node
    '''

    def __init__(self, value, nxt=None):
        ''' (LLNode, object, LLNode) -> NoneType

        Create LLNode (self) with data value and successor nxt.
        '''
        self.value, self.nxt = value, nxt
```

a wrapper class for list

The list class keeps track of information about the entire list — such as its front, back, and size.




```
class LinkedList:
    '''Collection of LLNodes

    front: LLNode -- front of list
    back:  LLNode -- back of list'''
```

size : int ~ size of this

```
    def __init__(self):
        ''' (LinkedList) -> NoneType

        Create an empty linked list.
        '''
        self.front, self.back = None, None
        self.size = 0
```

division of labour

Most of the work of special methods is done by the nodes:

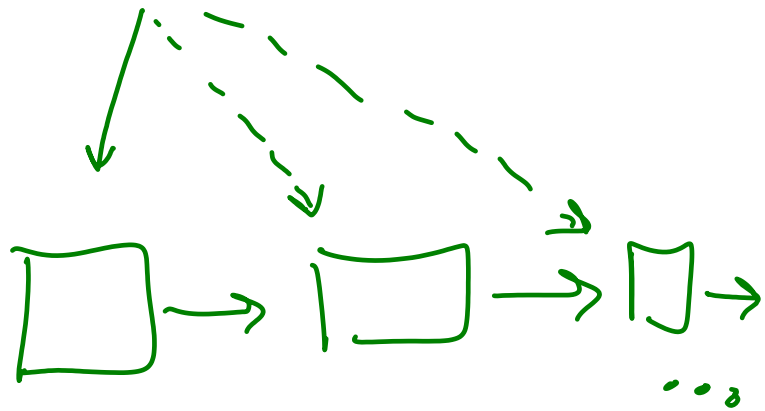
- `__repr__`
- `__str__`
- `__eq__`

Once these are done for nodes, it's easy to do them for the entire list.

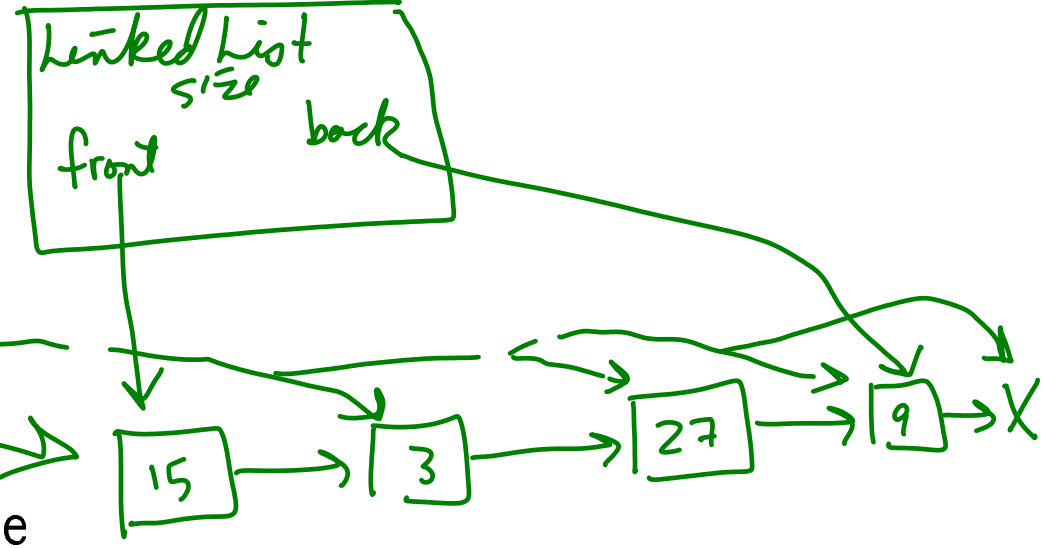
walking a list

Make a reference to (at least one) node, and move it along the list:

```
cur_node = self.front
while <some condition here...>:
    # do something here...
    cur_node = cur_node.nxt
```



__contains__

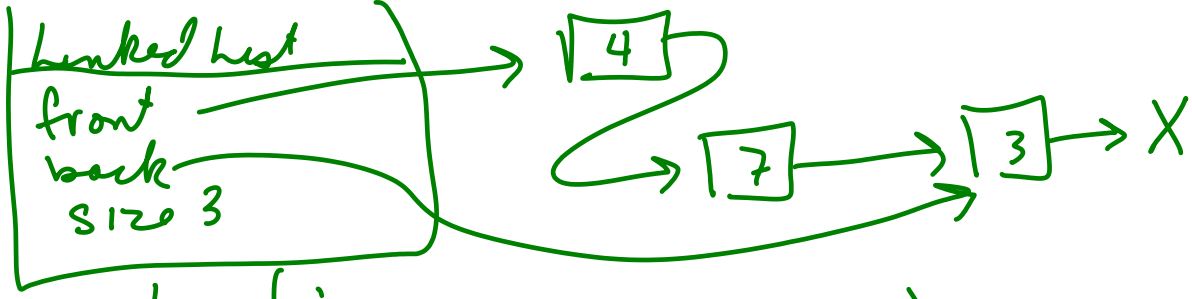


Check (possibly) every node

```
cur_node = self.front
while <some condition here...>:
    # do something here...
    cur_node = cur_node.nxt
```

- if cur_node is None: return False.
- if cur_node.value is what we're looking for return True
- keep looking
- walk through
  - report if find it
  - False if you reach end of list

__getitem__



Index outside → too big
neg: get index len(list) + neg amount.

Should enable things like

```
>>> print(lnk[0])
5
```

alias lnk.__getitem__(0)

for Wednesday

... or even

```
>>> print(lnk[0:3])
5 -> 4 -> 3 ->|
```

won't implement slices — if you want, look at help(slice)

append

We'll need to change...

- last node
- former last node
- **back**
- **size**
- possibly **front**

draw pictures!

delete_back

We need to find the **second last** node. Walk **two** references along the list.

```
prev_node, cur_node = None, lnk.front
# walk along until cur_node is lnk.back
while <some condition>:
    prev_node = cur_node
    cur_node = cur_node.nxt
```