CSC148 winter 2018

linked lists, iteration, mutation   week 4

Danny Heap

heap@cs.toronto.edu / BA4270 (behind elevators)

http://www.teach.cs.toronto.edu/~csc148h/winter/

416-978-5899

January 26, 2018

Computer Science
UNIVERSITY OF TORONTO

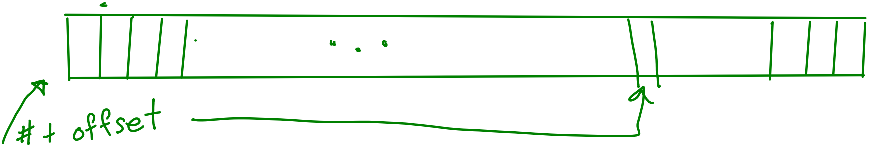
Outline

linked lists

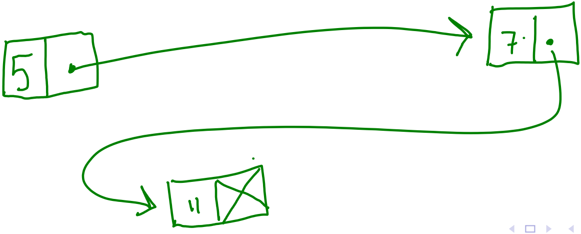
mutation

linked list queues

why linked lists?



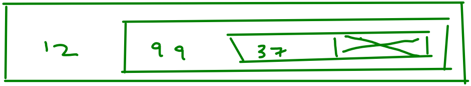
- regular Python lists are flexible and useful, but overkill in some situations   they allocate large blocks of contiguous memory, which becomes increasingly difficult as memory is in use.
- linked list nodes reserve just enough memory for the object value they want to refer to, a reference to it, and a reference to the next node in the list.



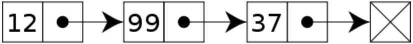
linked lists, two concepts

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

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.

LL class

a node class

to distinguish from built-in list

```
class LinkedListNode:
    """
    Node to be used in linked list

    === Attributes ===
    next_ - successor to this LinkedListNode
    value: data this LinkedListNode represents
    """
    next_: ~~LinkedListNode~~
    value: object

    def __init__(self, value: object,
                 next_: Union["LinkedListNode", None]=None) -> None:
        """
        Create LinkedListNode self with data value and successor next_.
        """
        self.value, self.next_ = value, next_
```

Union["LinkedListNode", None]

default value

a wrapper class for list

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

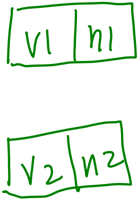
```
class LinkedList:
    """
    Collection of LinkedListNodes

    === Attributes ==
    front - first node of this LinkedList
    back - last node of this LinkedList
    size - number of nodes in this LinkedList, >= 0
    """
    front: Union[LinkedListNode, None]
    back: Union[LinkedListNode, None]
    size: int

    def __init__(self):
        """
        Create an empty linked list.
        """
        self.front, self.back, self.size = None, None, 0
```

division of labour

Three reasonable ways to define __eq__

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

- `__str__` ✓
- `__eq__`



id == id

value == value

value == value & next_ == next_

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:

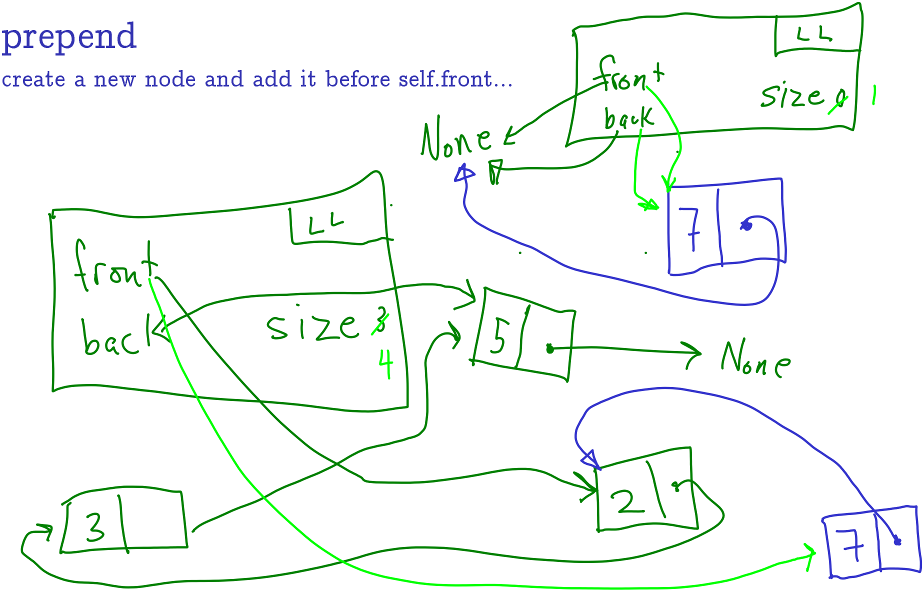
this name "moves" along the list...

check for None!

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

prepend

create a new node and add it before self.front...



__contains__

Special - alias in

Check (possibly) every node

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

not None

value? return True

__getitem__

alias lnk[...]

Should enable things like

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

help(list)

... or even

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

lnk.size == 3

lnk[3] → IndexError

lnk[2] → ✓

lnk[1] ✓

lnk[0] →

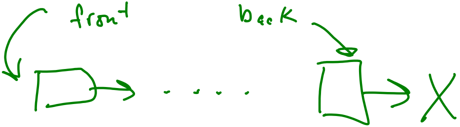
lnk[-1] ✓

lnk[-2]

lnk[-3]

lnk[-4]

append



We'll need to change...

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

draw pictures!!

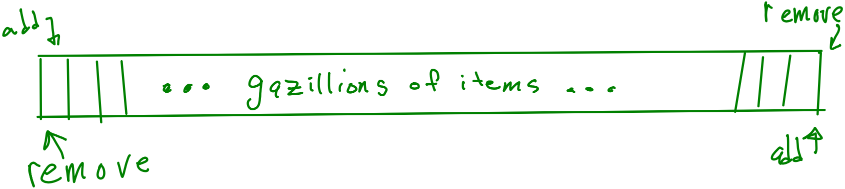
delete_back

left as an exercise...

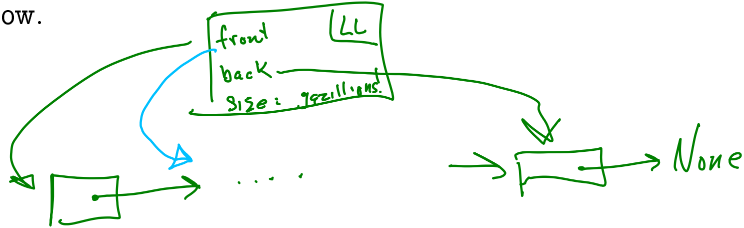
We need to find the **second last** node. Walk **two** references along the list.

```
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
```

something linked lists do better than lists?



list-based Queue has a problem: adding or removing will be slow.



symmetry with linked list

← see previous!

which end of a linked list would be best to add, which to remove? why?? think about front, back ...

build pop_front

- delete_front
- then pop_front

... already have append

revisit Queue API

these are now easy

use an underlying LinkedList

revisit Stack API while we're at it

also use an underlying LinkedList

they're all Containers

see our code

stress drive them through **container_cycle**, in container_timer.py:

- list-based Queue
- linked-list-based Queue
- list-based Stack
- linked-list-based Stack

what matters is the growth rate

when the list is 10x as big
what happens to run-time?

as Queue grows in size, list-based-Queue bogs down, becomes impossibly slow

notes...