CSC 148 Winter 2017

Week 6

Linked lists,

iteration, mutation

Bogdan Simion

bogdan@cs.toronto.edu

http://www.cs.toronto.edu/~bogdan

University of Toronto, Department of Computer Science

Outline

- Linked lists
- Linked list operations
- Mutating linked lists

Linked Lists

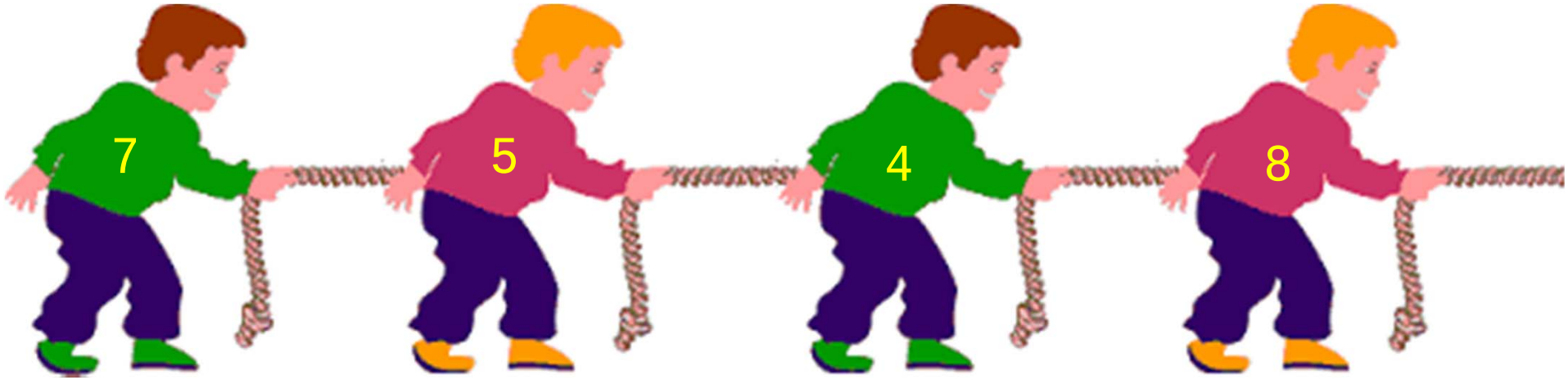
University of Toronto, Department of Computer Science

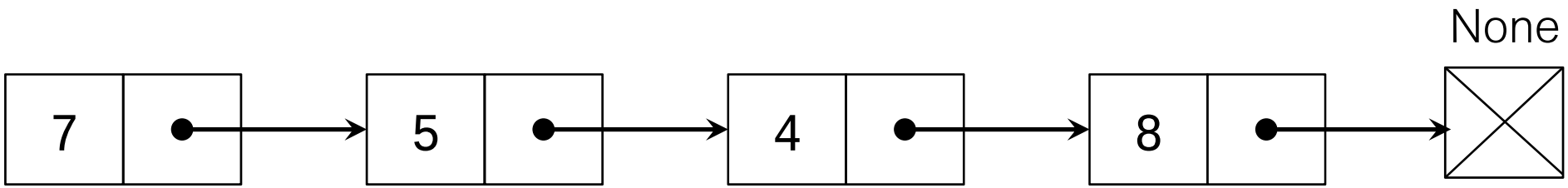
Motivation

- 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
  - e.g., Stack that uses list - add/remove items at the end vs. the front...

- Linked list nodes reserve just enough memory for the object value they refer to, a reference to it, and a reference to the next node in the list

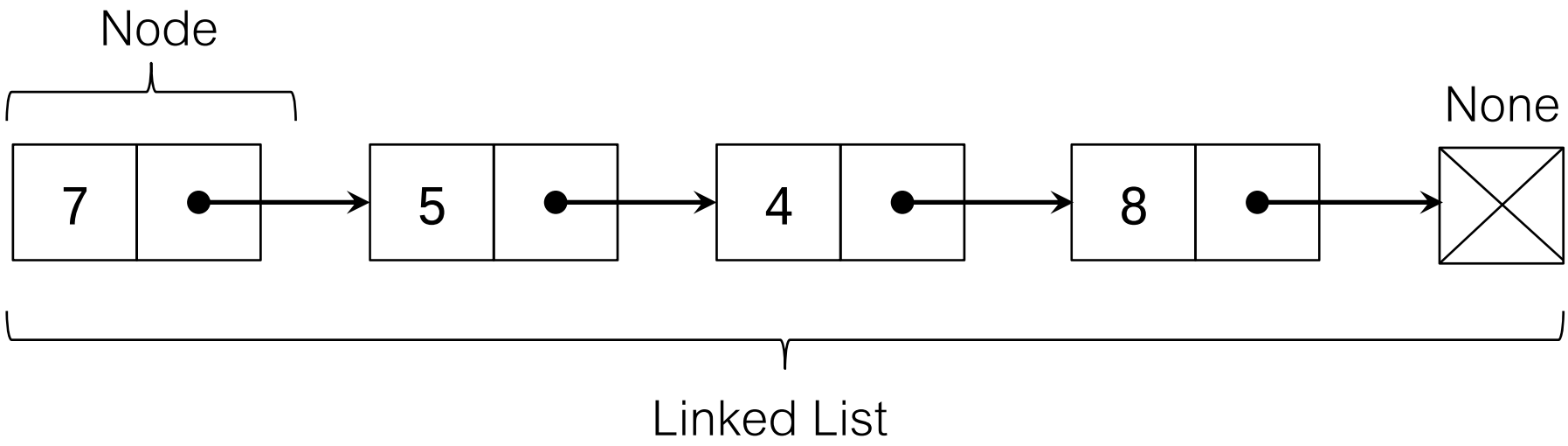
How to look at a linked list ...





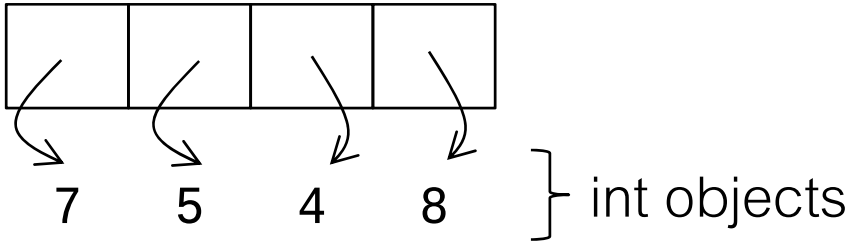
Linked List

- There are two useful, but different, ways of thinking of linked list nodes:
  - 1. As a list made up of an item (value) and a sub-list (rest)
  - 2. As objects (nodes), each containing a value and a reference to another similar node object (the “next link in the chain”)

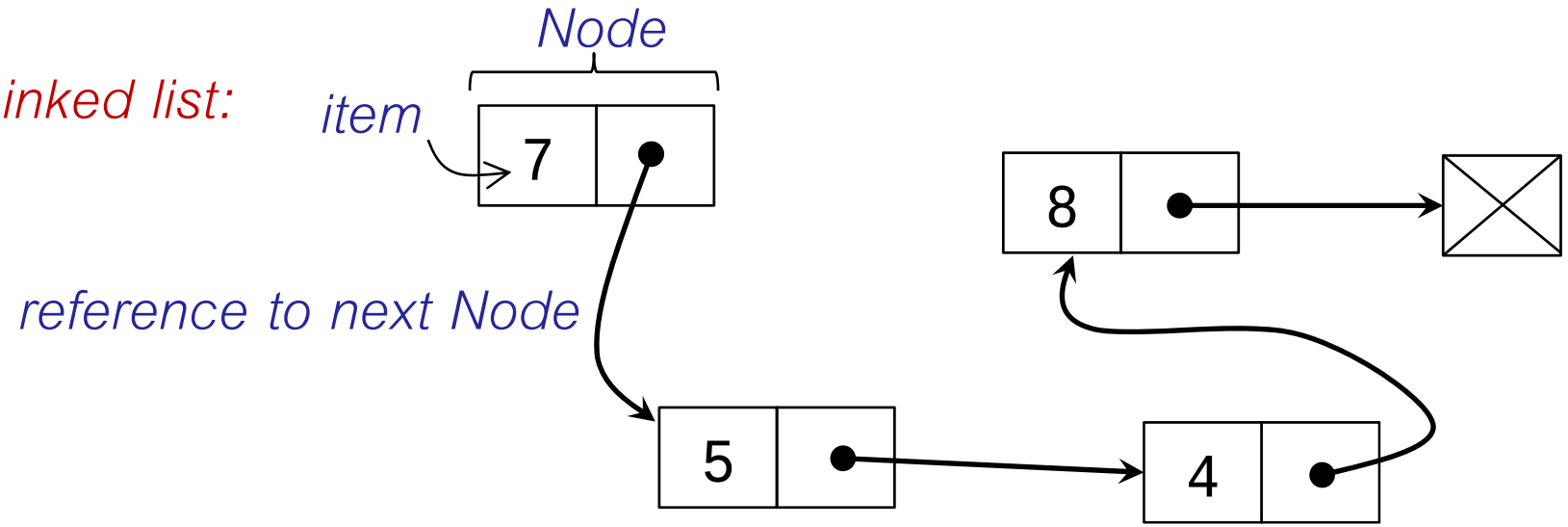


Linked List nodes

Python list = [7, 5, 4, 8]



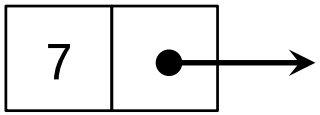
Linked list:



- Get in the habit of drawing diagrams to visualize things better ...
- Let's design a linked list node, then a separate "wrapper" to represent the linked list as a whole ...

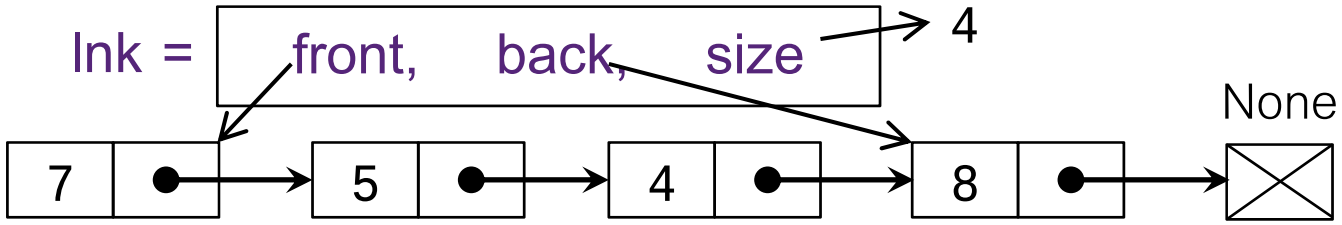
Node class

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

    === Attributes ===
    @param LinkedListNode next_: successor to this LinkedListNode
    @param object value: data this LinkedListNode represents
    """

    def __init__(self, value, next_=None):
        """ Create LinkedListNode self with data value
            and successor next_

        @param LinkedListNode self: this LinkedListNode
        @param object value: data of this linked list node
        @param LinkedListNode|None next_: successor to self
        @rtype: None
        """

        self.value, self.next_ = value, next_
```



A wrapper for class List

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

    === Attributes ===
    @param LinkedListNode front: first node of this LinkedList
    @param LinkedListNode back: last node of this LinkedList
    @param int size: number of nodes in this LinkedList
                     (a non-negative integer)
    """

    def __init__(self):
        """ Create an empty LinkedList

        @param LinkedList self: this LinkedList
        @rtype: None
        """

        self.front, self.back, self.size = None, None, 0
```

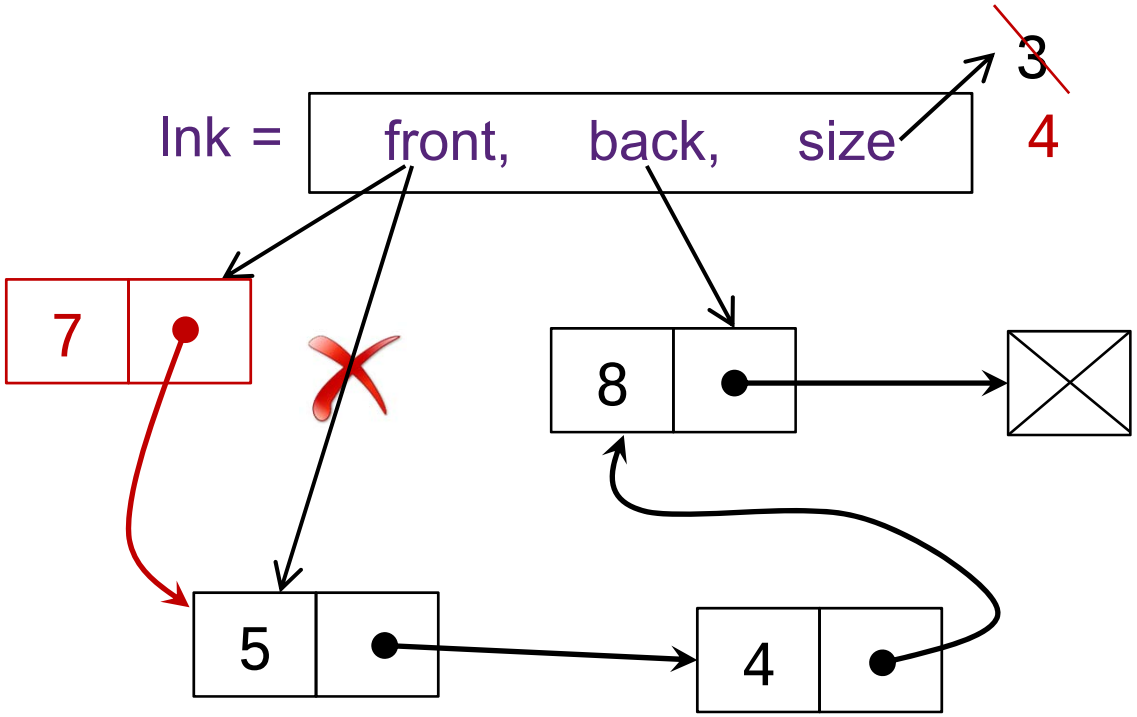


Division of labour

- Some of the work of special methods is done by the nodes:
  - __str__ **Something intuitive, say: "7 -> 5 -> 4 -> 8 -> |"**
  - __eq__ **More on this later ...**
- Once these are done for nodes, it's easy to do them for the entire list.
- How do we divide the work? What belongs in the node and what belongs in the linked list? Think what makes sense ...
  - May take different approaches ...

prepend (append to front)

- Easy: simply adjust the front reference
  - Nothing to do for "back" (or do we?)

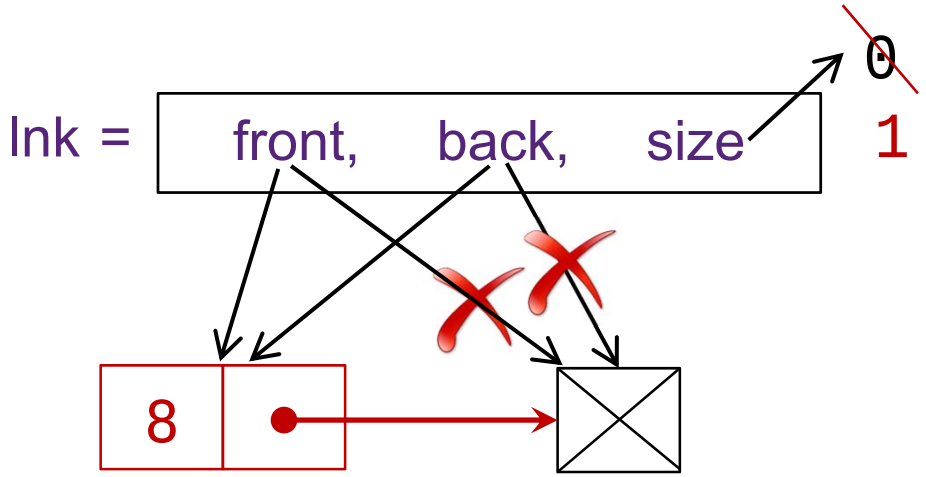


prepend (append to front)

- Easy: simply adjust the front reference
  - Make sure to account for what happens to "back" in corner cases ...

What if

list is empty?



prepend (append to the front)

- Easy: simply adjust the front reference
  - Make sure to account for what happens to "back" in corner cases ...

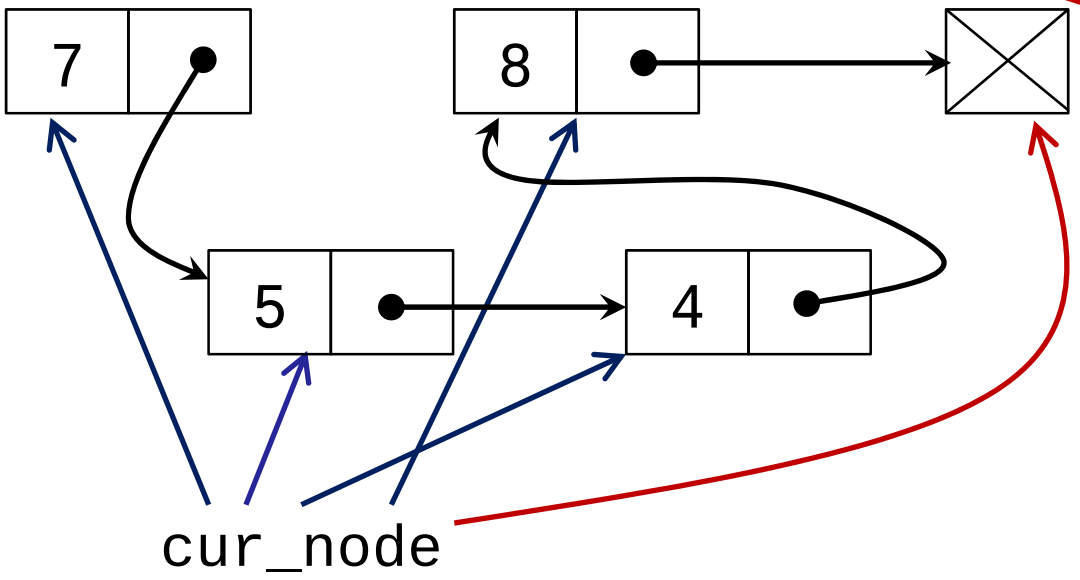
```
# create a new LinkedListNode and point "front" to it
lnk.front = LinkedListNode(value, lnk.front)

# list has no items yet?
# "back" should reference this new node too
if lnk.back is None:
    lnk.back = lnk.front

lnk.size += 1
```

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.next_
```

very common pattern

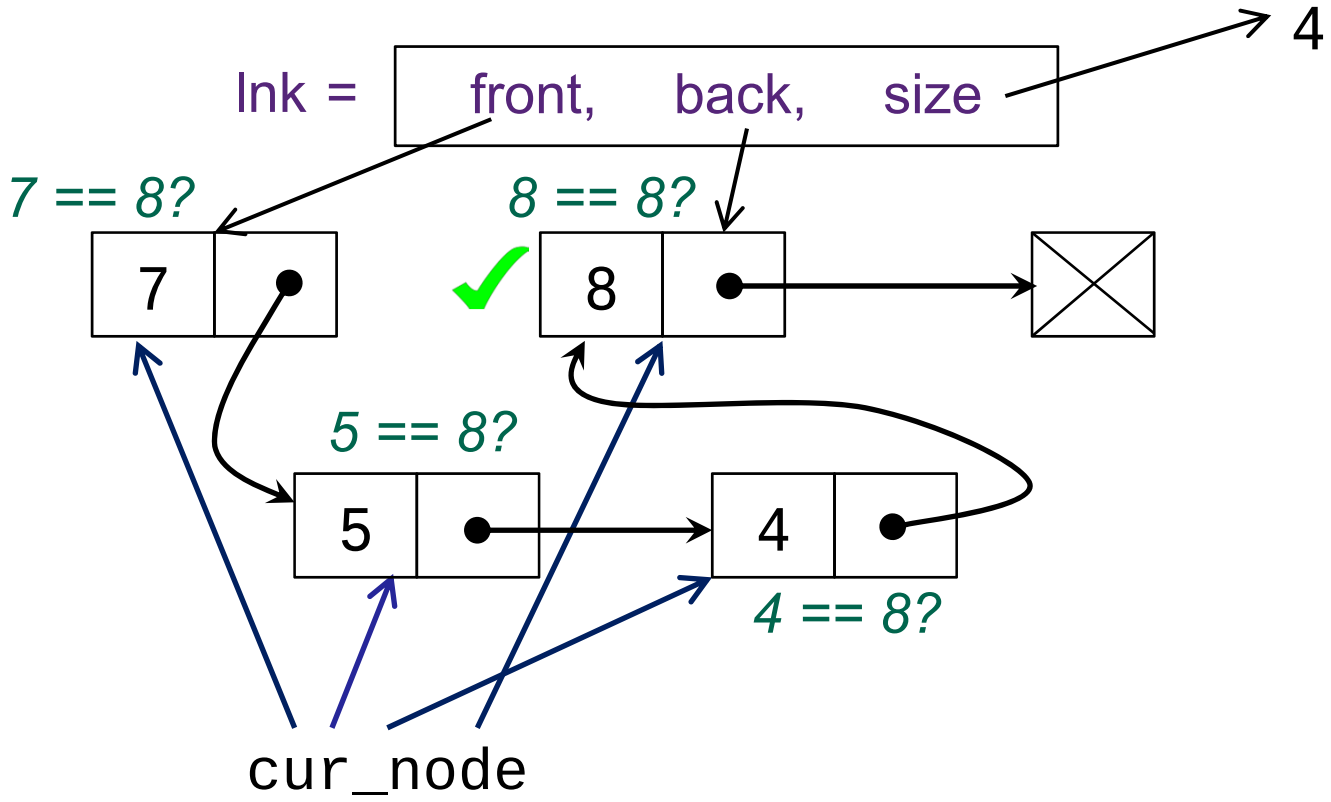


What if cur_node is None?

__contains__

- Check (possibly) every node

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



Question1:

Does the linked

list contain 8?

Question2:

Does the linked

list contain 3?

__getitem__

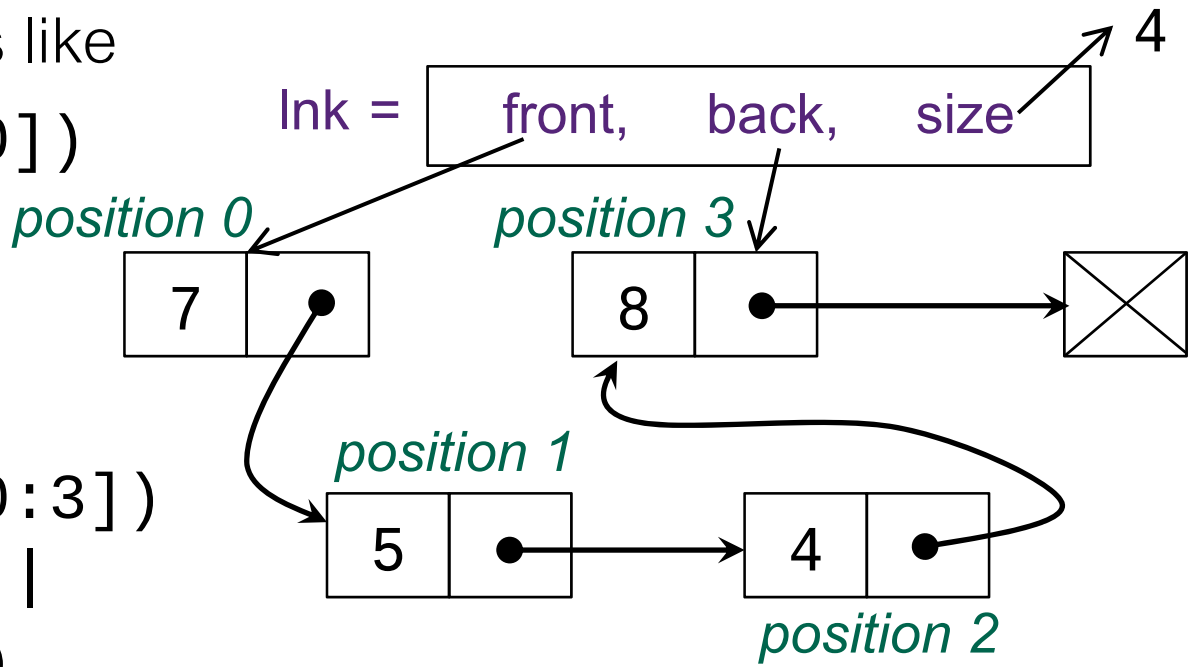
- Should enable things like

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

... or even this:

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

(More on slices later...)



- What corner cases do we have to be careful about?
- How do we handle them?

append

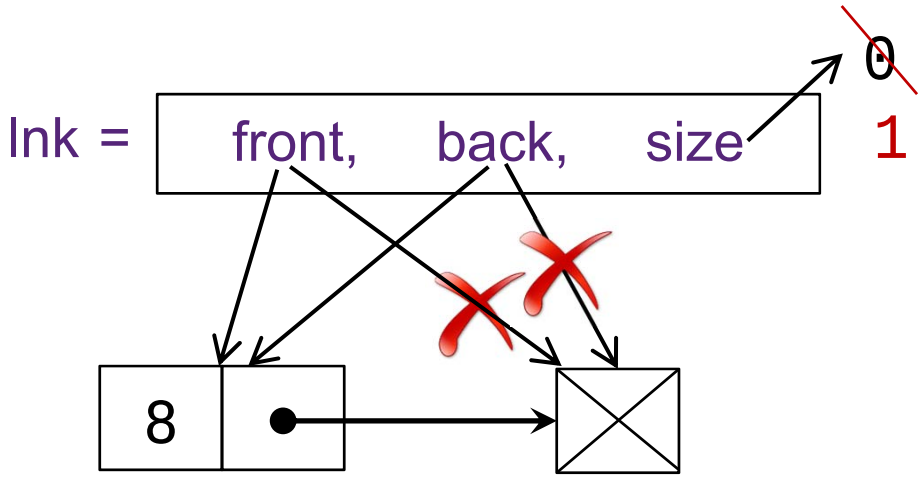
- We'll need to change...
  - last node
  - former last node
  - back
  - size
  - possibly front .. why?

- Always draw pictures!

append

- First node being appended (draw on worksheet!)

List is initially empty. Appending a new node.

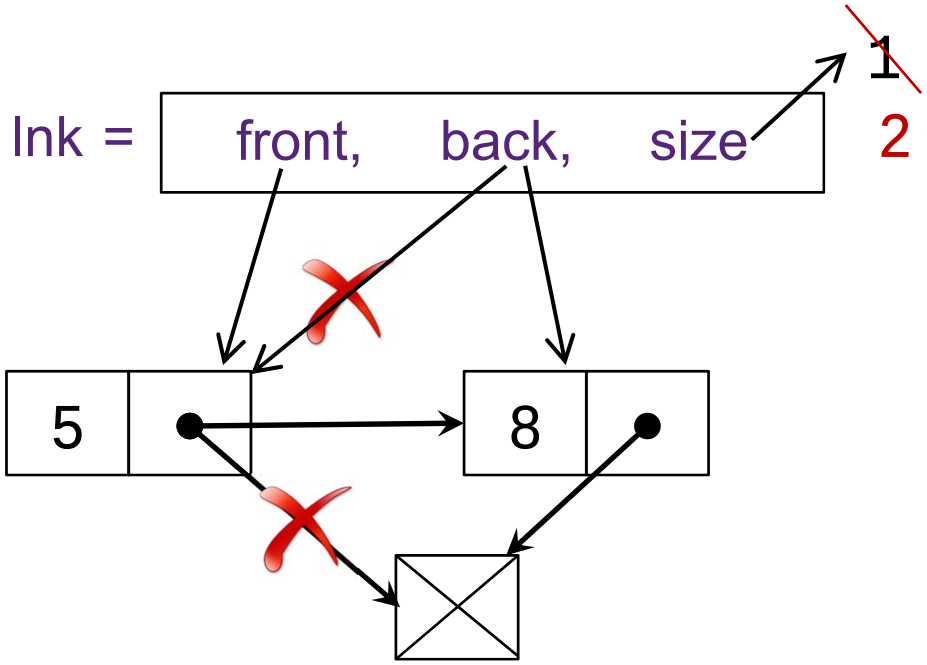


- Always draw pictures!

append

- First node being appended (draw on worksheet!)

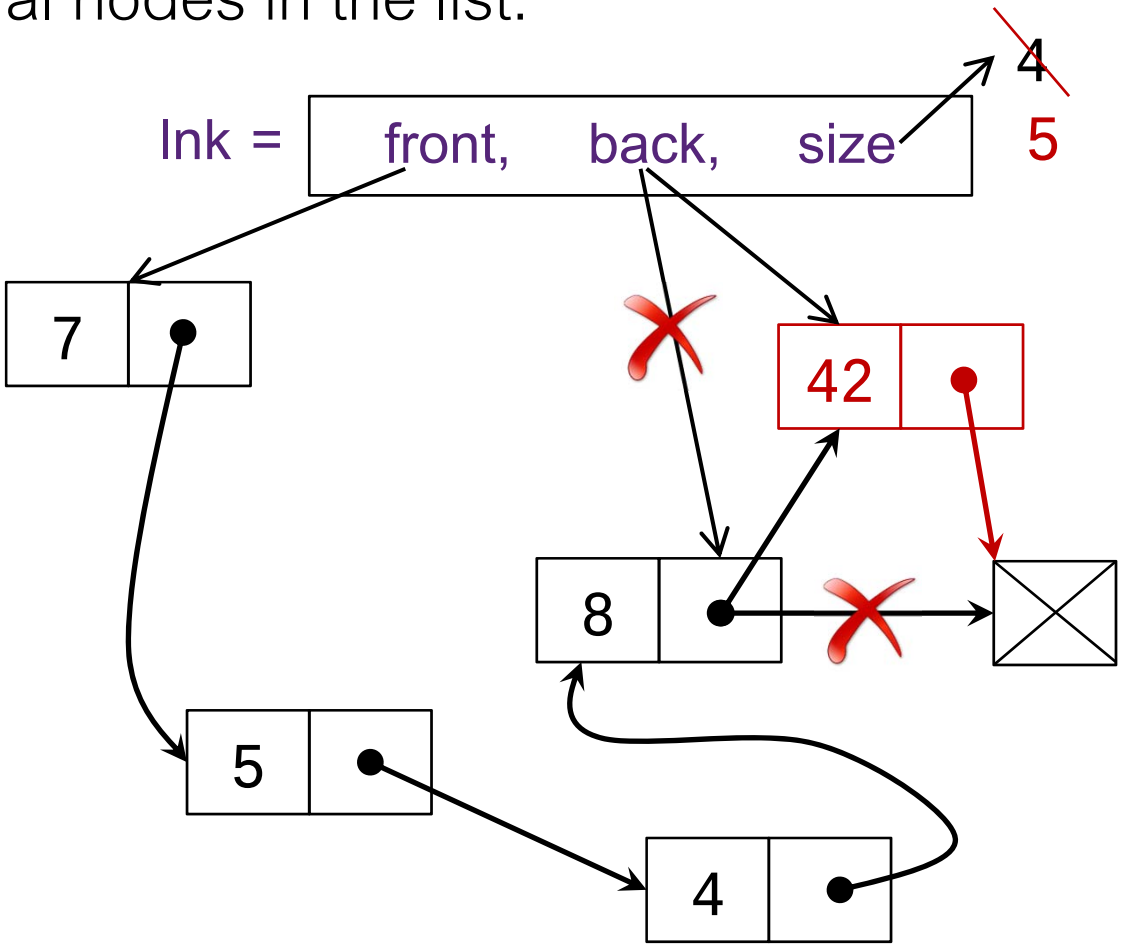
List has one element (8). Appending a new node (5).



- Always draw pictures!

append

- Several nodes in the list:



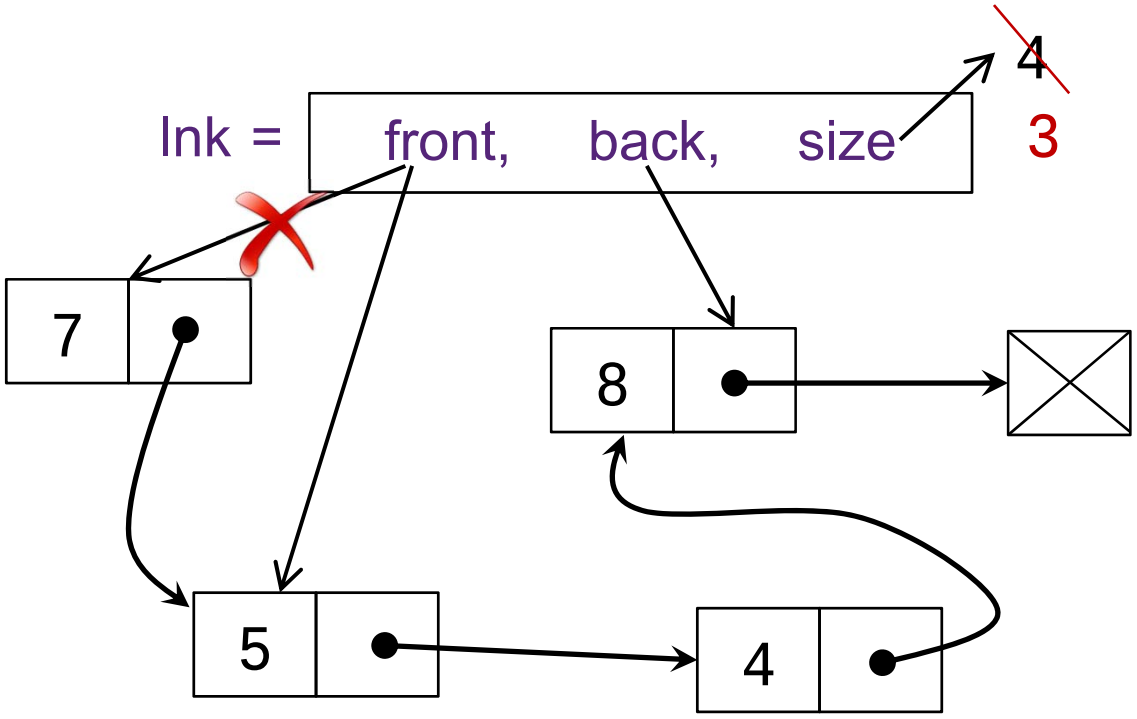
- Always draw pictures!

Inserting a node

- Practice problem for home ...
- Assume we want to keep the list sorted
- Implement a function insert_sorted, which inserts a value in the right place in the linked list, to keep the list's sorted property
- Hints:
  - You've seen how to walk a linked list, and how to "link in" a new node
  - Draw diagrams!!

delete_front

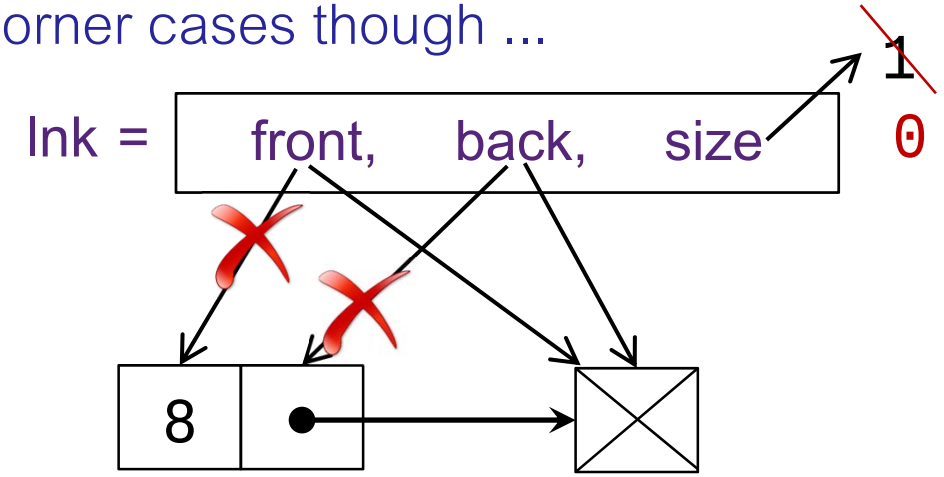
- Easy: make front reference the second node (garbage collection takes care of former first node automatically)
  - No need to walk the list ...



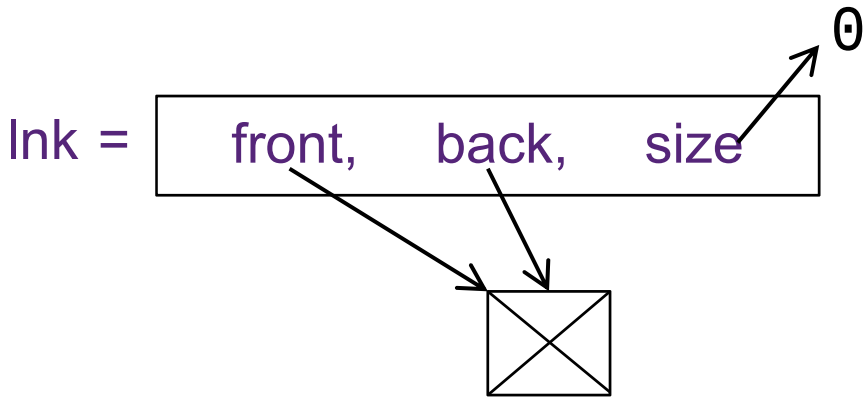
delete_front

- Easy: make front reference the second node (garbage collection takes care of former first node automatically)
  - Consider corner cases though ...

What if only 1 node?



What if list is empty?



delete_front

- Easy: make front reference the second node (garbage collection takes care of former first node automatically)
  - No need to walk the list for this one ...
  - Consider corner cases though ...

```
# cannot delete from empty list
assert lnk.front is not None, "Delete from empty list!"

# list has only one item?
if lnk.back is lnk.front:
    lnk.back = None

# "reposition" the front on the second node
lnk.front = lnk.front.next_
lnk.size -= 1
```

delete_back

- On your own .. practice!
- Hint: we need to find the second last node => must walk two references along the list

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

- Draw diagrams to picture this better!

LinkedList-based queues

LinkedList-based stacks

What do linked lists do better than lists?

- Imagine how a queue works:

9	5	27	14	3	31	8

2	0	10	23

- Using a regular Python list to implement a Queue
  - Decision: which end of the list is the front of the queue?
  - Problem: adding or removing will be slow. Why?

9	5	27	14	3	31	8

What do linked lists do better than lists?

- Imagine how a queue works:

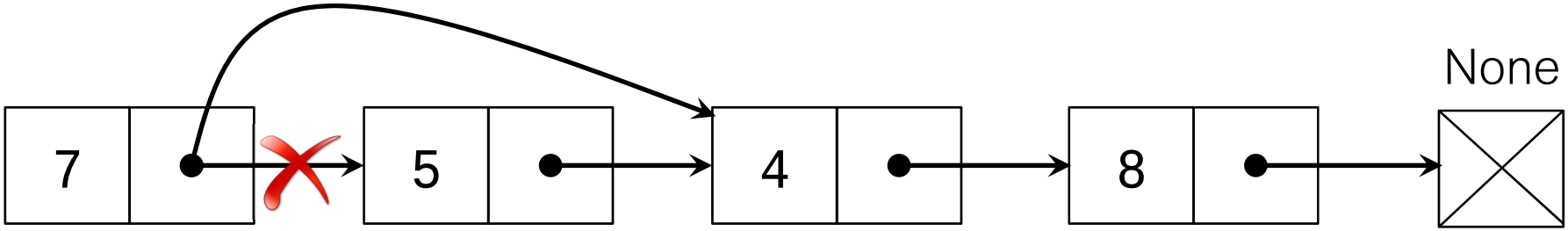
27	14	3	31	8	2	0	10	23

- Using a regular Python list to implement a Queue
  - Decision: which end of the list is the front of the queue?
  - Problem: adding or removing will be slow. Why?

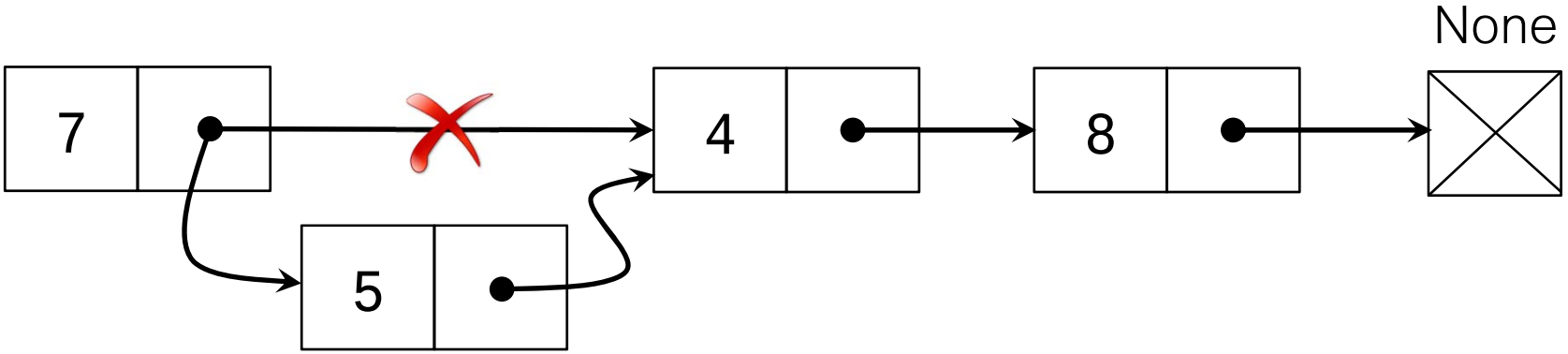
5	27	14	3	31	8	

What about linked lists?

- Remove an element – much faster (no need to "shift" anything)



- Insert an element – no need to shift subsequent elements



- Just adjust a couple of references, no moving memory!

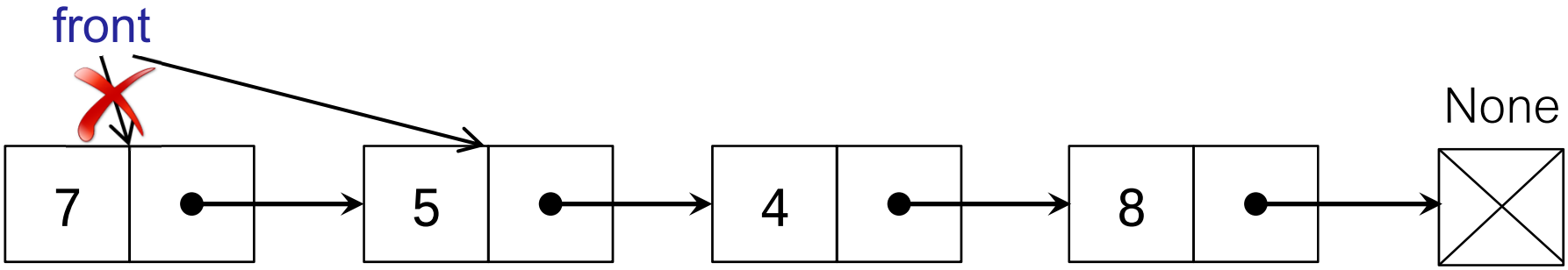
Symmetry to linked list

- Which end of a linked list would be best to add, which to remove? Why?

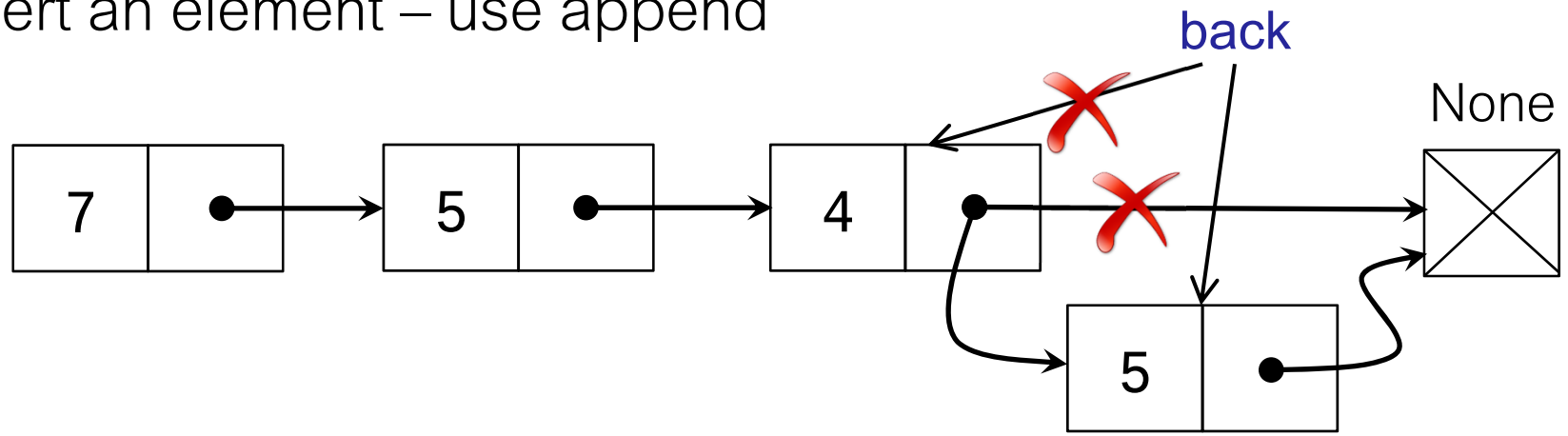
- Already have append: adds element to the back of the list
- Already have delete_front: removes the ‘front’ element from the list and returns the respective Node

Queue implementation ...

- Draw diagrams!!
- Remove an element – use delete_front



- Insert an element – use append



- Just adjust a couple of references, no moving memory!

Revisit Queue API

- Add <=> Append
- Remove <=> Delete_front
- Is_empty <=> size == 0

- Use an underlying LinkedList

Revisit Stack API too

- Add <=> Prepend
- Remove <=> Delete_front
- Is_empty <=> size == 0

- Use an underlying LinkedList

All are Containers

- Use different subclasses of Container to compare performance

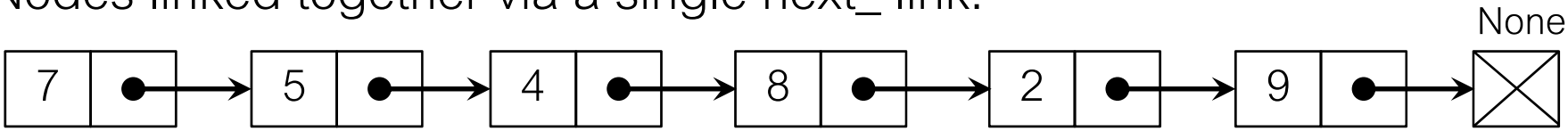
- Stress drive them through *container_cycle*, and compare times for:
  - List-based Queue (Python built-in list)
  - LinkedList-based Queue
  - List-based Stack (Python built-in list)
  - LinkedList-based Stack

What matters is growth rate!

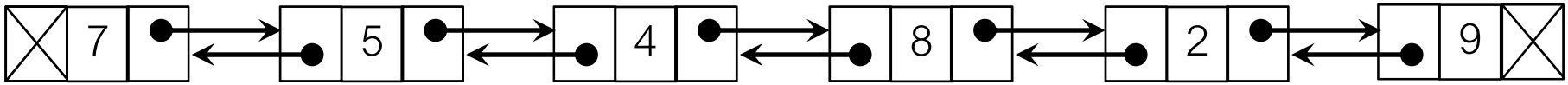
- Notice anything?
  - As the queue grows in size, list-based Queue (implemented using native Python list) bogs down impossibly!

Organizing nodes differently ...

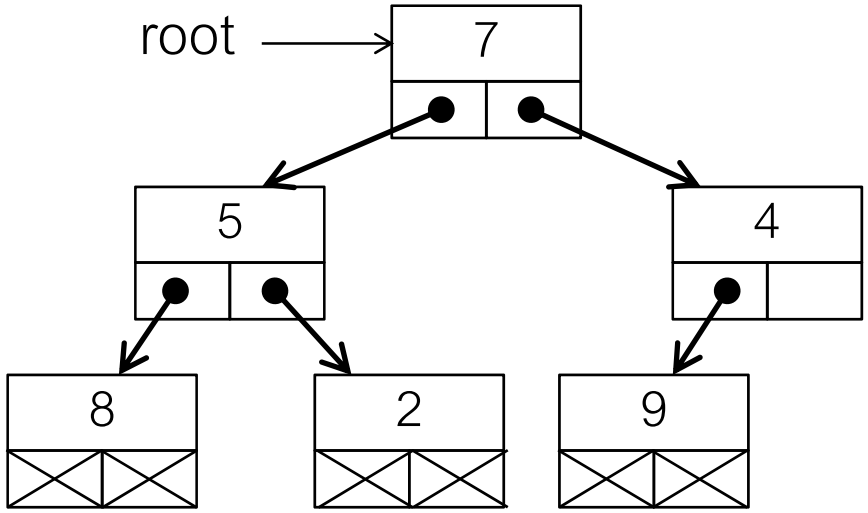
- Nodes linked together via a single next_ link:



- How about a doubly linked list (next + prev links)? Circular, optionally?



- What about a "hierarchical" structure of Nodes? .. Aka Tree



- Advantage: search path to each item in a tree is much shorter than in a linked list! (if the tree is reasonably balanced..) *To be continued ...*