CSC 148

Week 5

Sadia ‘Rain’ Sharmin

Classes begin 10 minutes after the hour

This Week's Cool Programmer(s) Feature: *Margaret Hamilton*

Week 5

Linked Lists

It's all about the links

- Traversals
- Insertion
- Deletion
- Efficiency

Announcements

Midterm

Our midterm is coming up next week!

More info here

Linked List Traversals

List Implementations

Two major ways

Array-based lists store references to elements in contiguous blocks of memory.

Linked lists can store elements anywhere, but each element must store a reference to the next element in the list.

List Implementations

So what's the difference, really?

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...

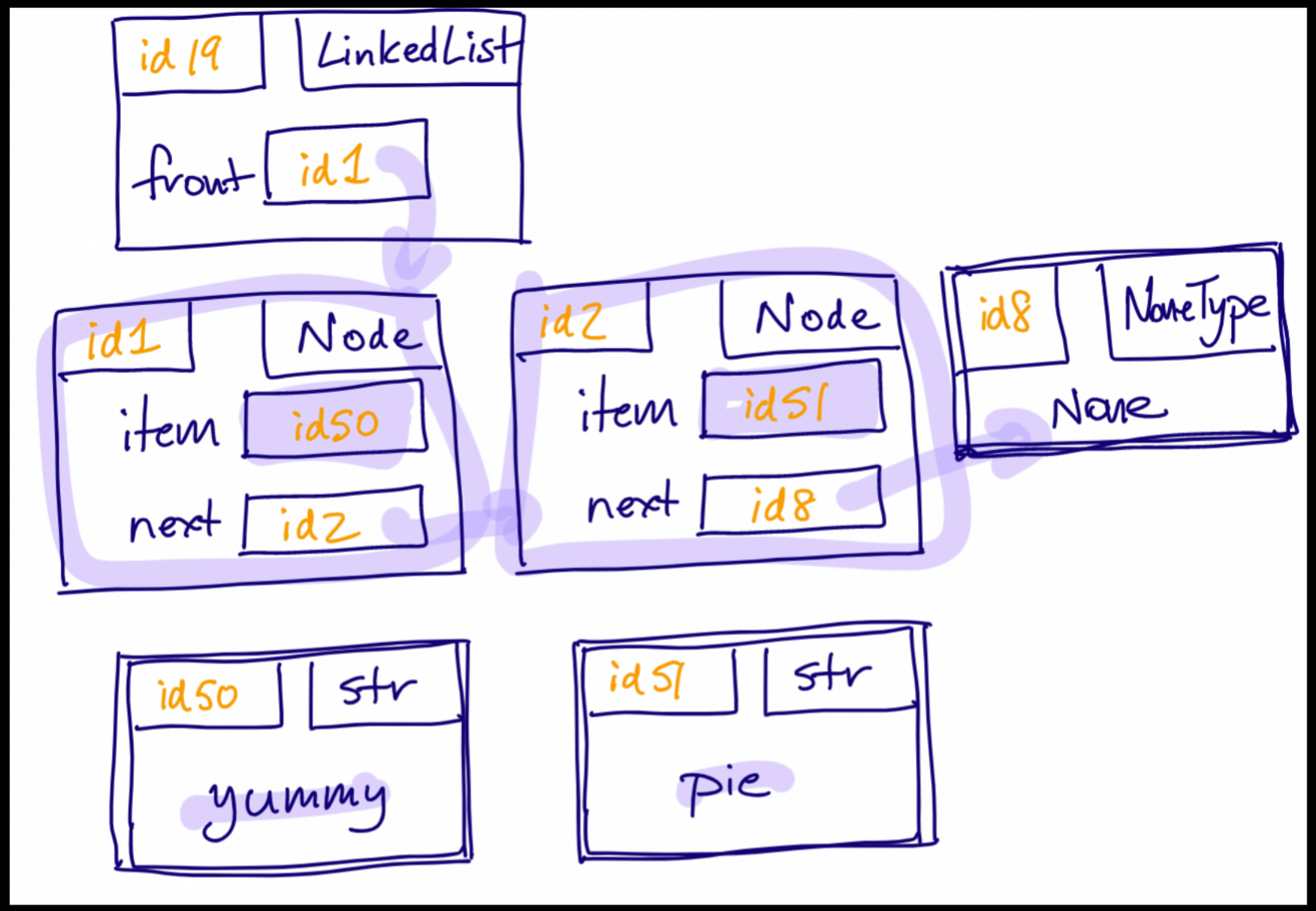
Linked list nodes reserve just enough memory for the object value they refer to, a reference to it, and a reference to the next node in the list

Our goals this week

Work with linked lists by implementing the same operations as Python's built-in list.

Analyze the running time of our linked list methods and compare them to the array-based list.

Visualizing Linked Lists

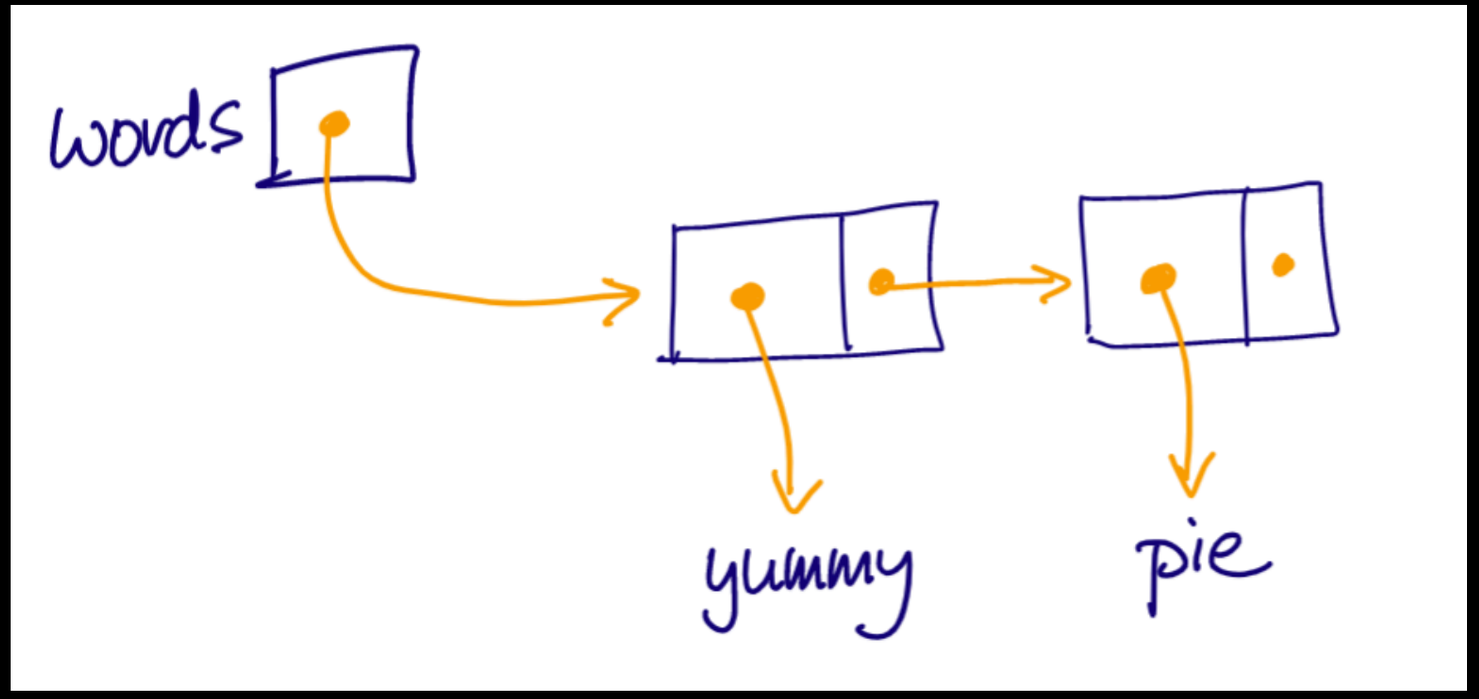


Here is a memory model diagram of a LinkedList with two elements.

We are going to switch to a more abstract shorthand for this.

Visualizing Linked Lists

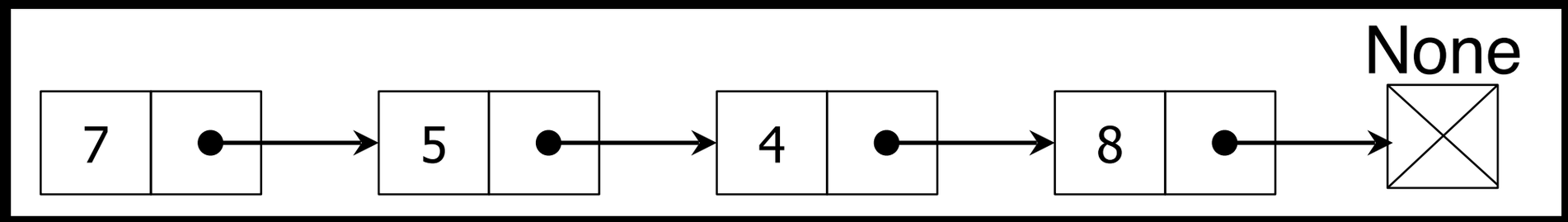
Here is a more abstract shorthand for the same linked list represented in the memory model diagram we just saw:



Visualizing Linked Lists

It's all about the links; that's what we focus on today.

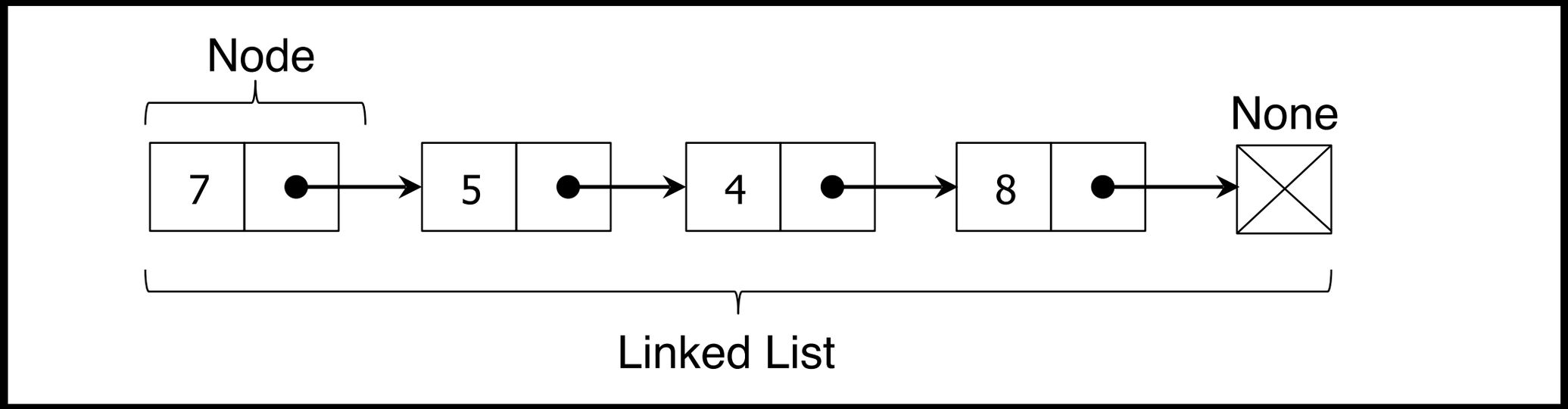
Visualizing Linked Lists



Thinking about linked lists

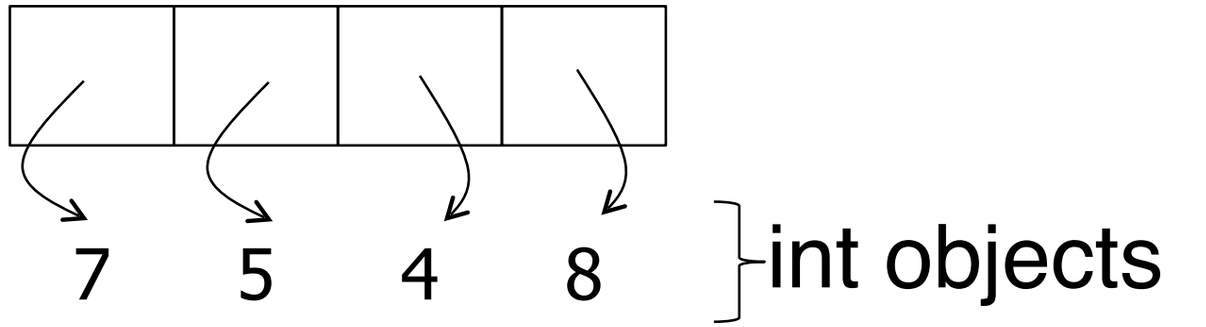
There are two useful, but different, ways of thinking of linked list nodes:

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”)

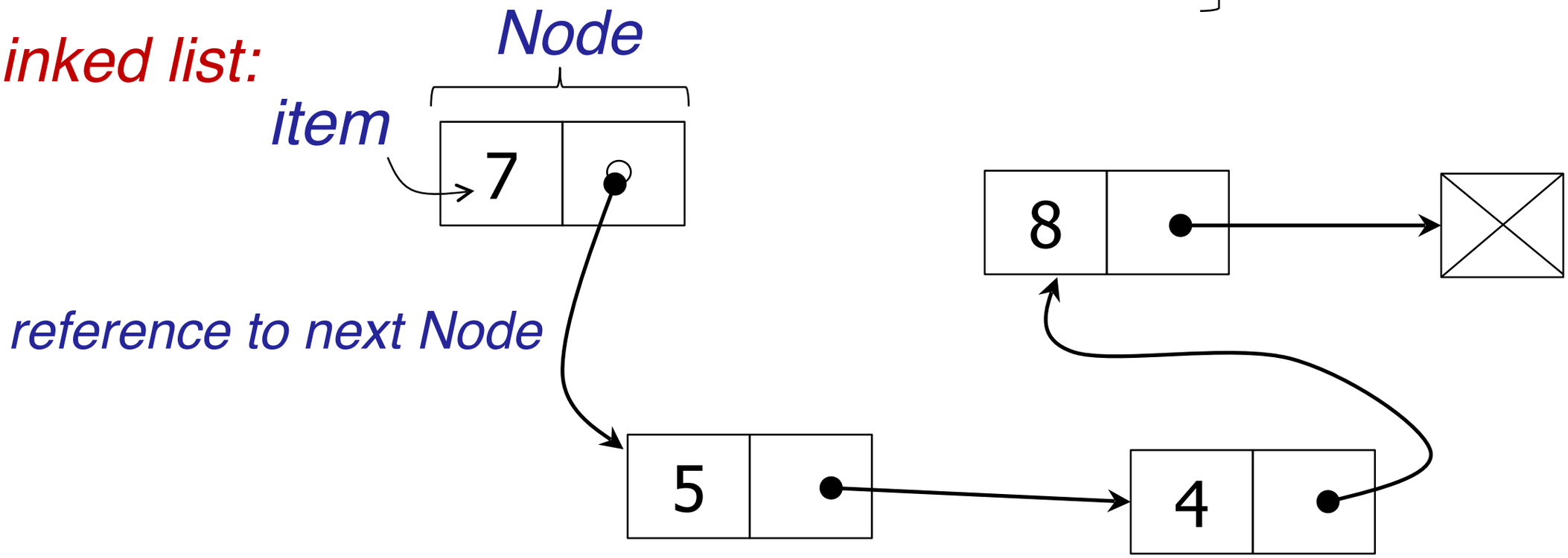


Drawing Linked Lists

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



Linked list:



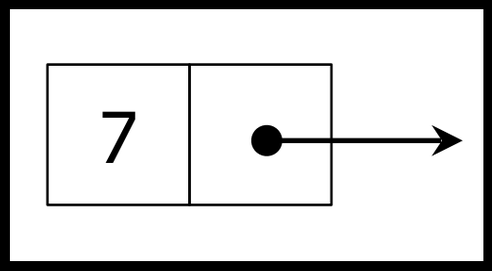
Get in the habit of drawing diagrams to visualize things better!

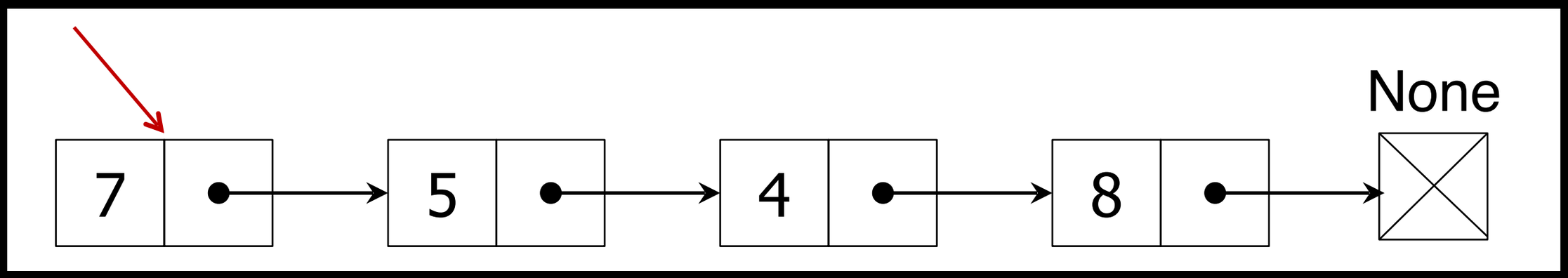
Code summary

We have a class for a linked list node, then a separate "wrapper" to represent the linked list as a whole. These are our data structures:

```
class _Node:
    item: Any
    next: Optional[_Node]

class LinkedList:
    _first: Optional[_Node]
```



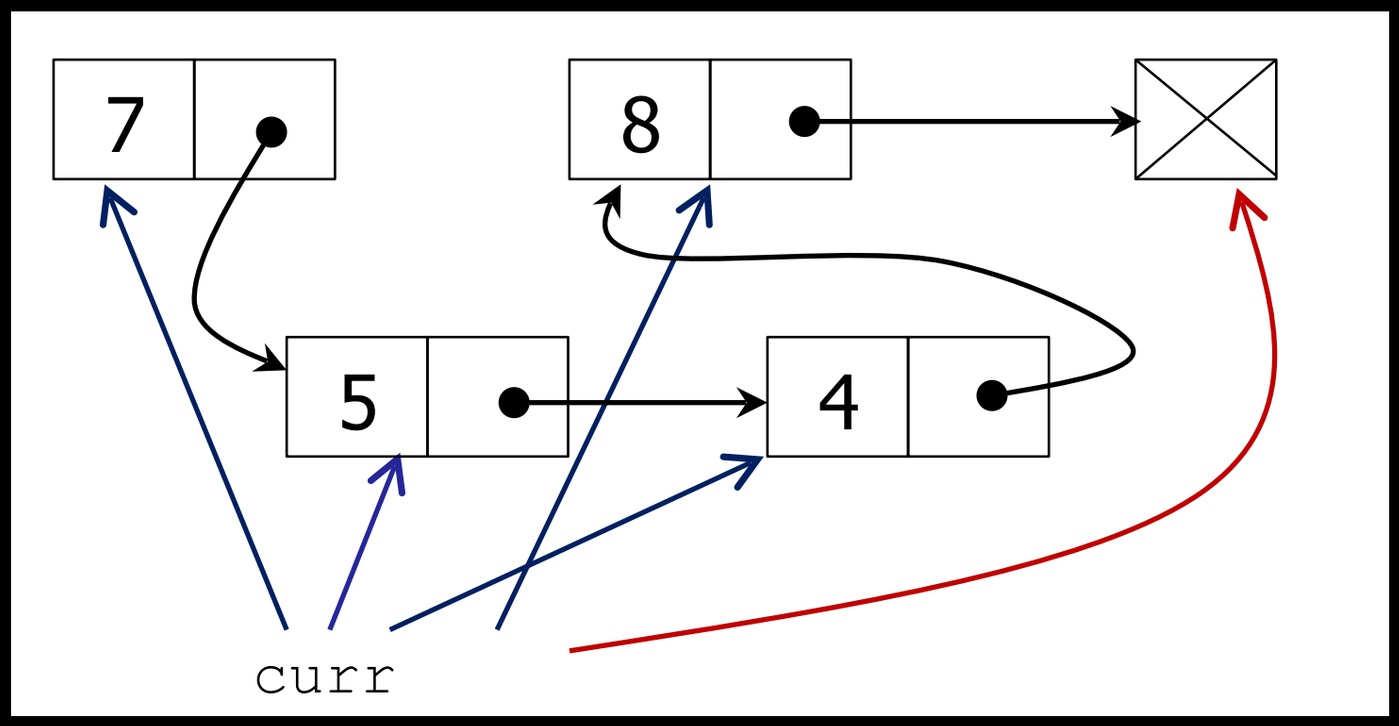


Walking a list

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

```
curr = self._first
while <some condition here...>:
    # do something here ...
    curr = curr.next
```

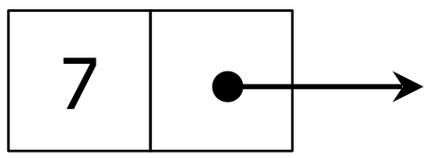
What if curr is None?



Linked lists code summary

Data structures:

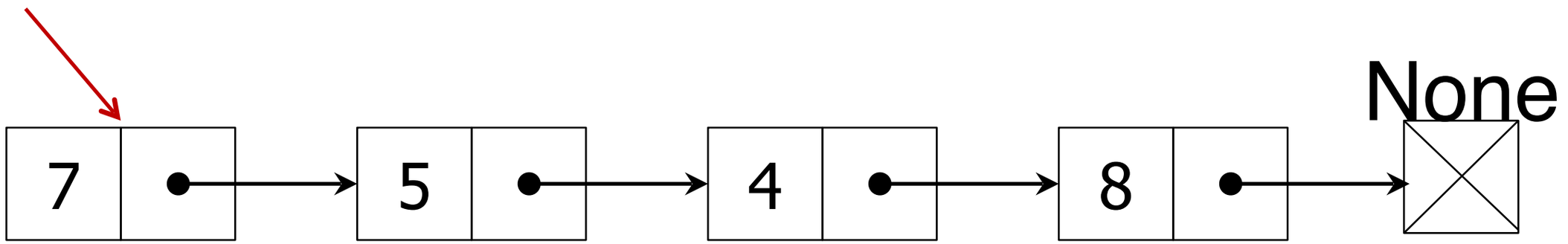
```
class _Node:
    item: Any
    next: Optional[_Node]
```




```
class LinkedList:
    _first: Optional[_Node]
```

Traversal idea:

```
curr = self._first
while curr is not None:
    ... curr.item ...
    curr = curr.next
```



Side Note

Special Methods

In the next worksheet, we will work on writing code for an __eq__ and __getitem__

DEMO

Special Methods

See snowball.py

WORKSHEET

Linked List Traversals

Linked Lists

Insertion

Linked Lists

Insert

We might want to implement all sorts of insert variations depending on what operations we want the linked list to support, e.g.

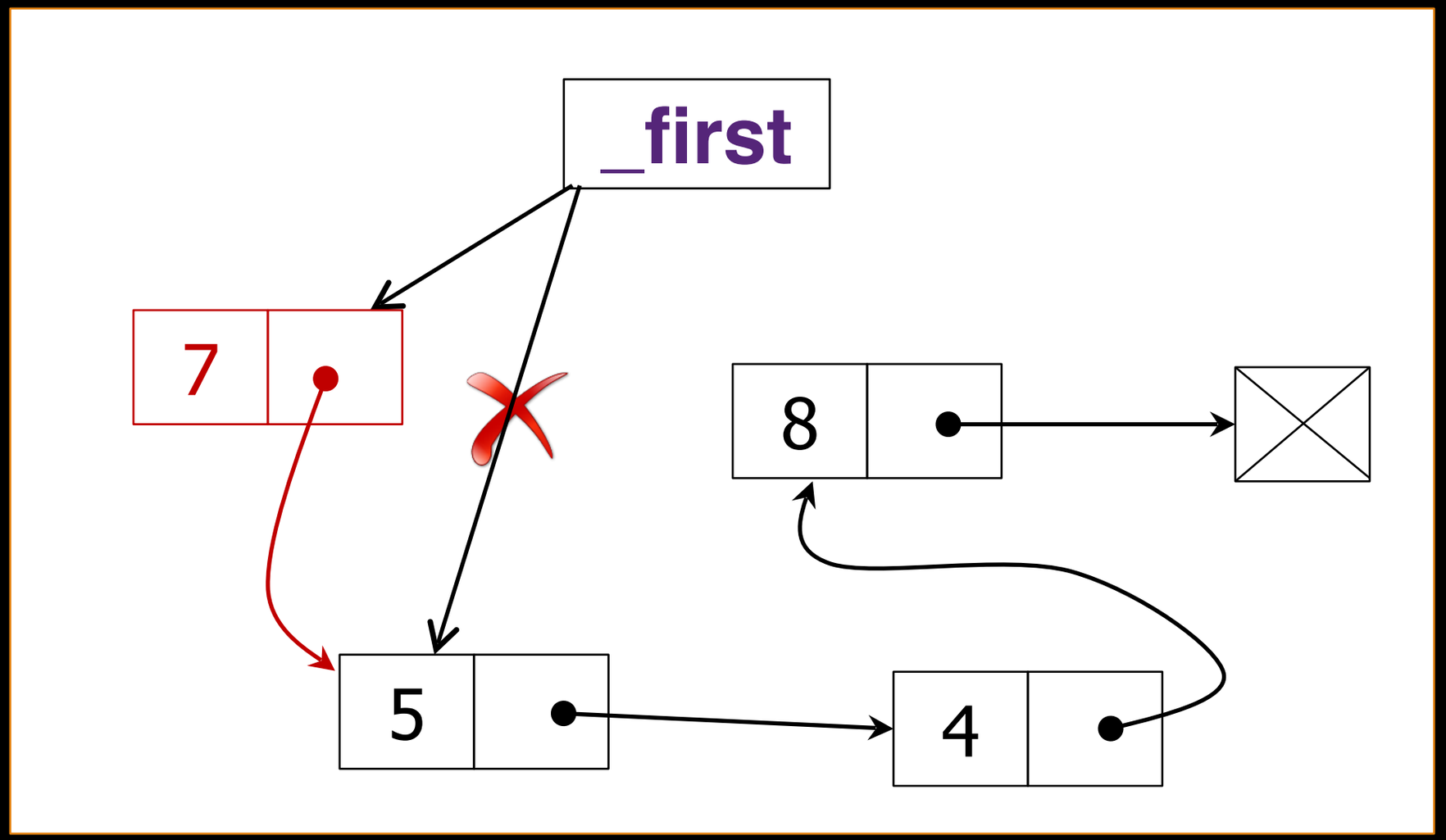
- Prepend
- Append
- Insert at a given index

Use diagrams to visualize such operations!

Insertion

prepend (insert at the front)

Easy: simply adjust the _first reference



Insertion

append

We'll need to change...

some node inside the list

possibly the last node

possibly _first .. why?

Always draw diagrams!

Insertion

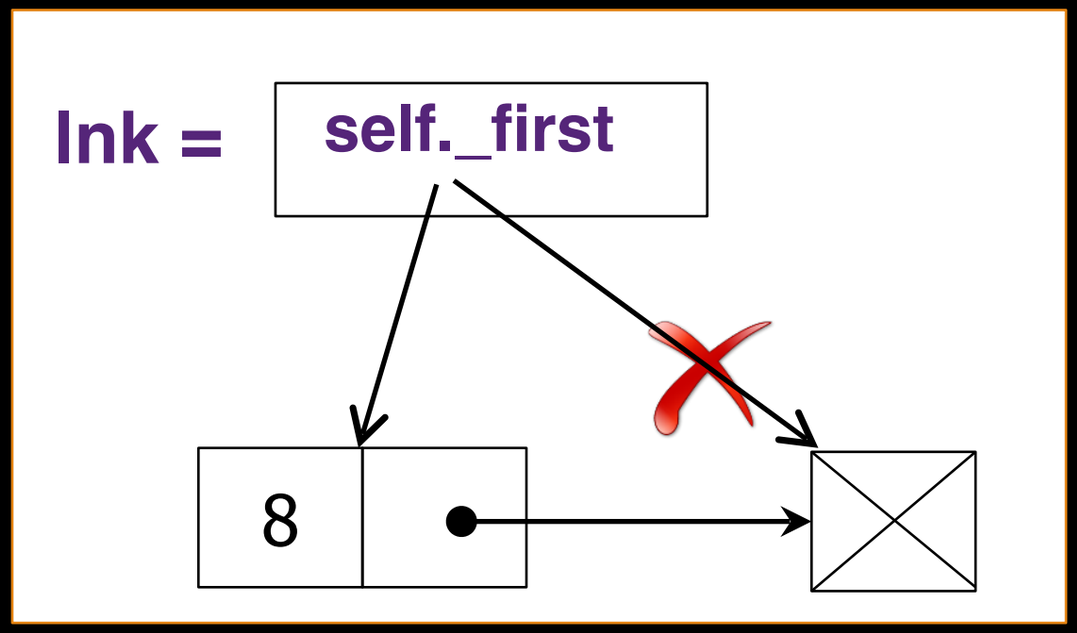
append to empty list

First node being appended

Sort of similar to prepend in this particular corner case...

List is initially empty.

Appending a new node.



Always draw diagrams!

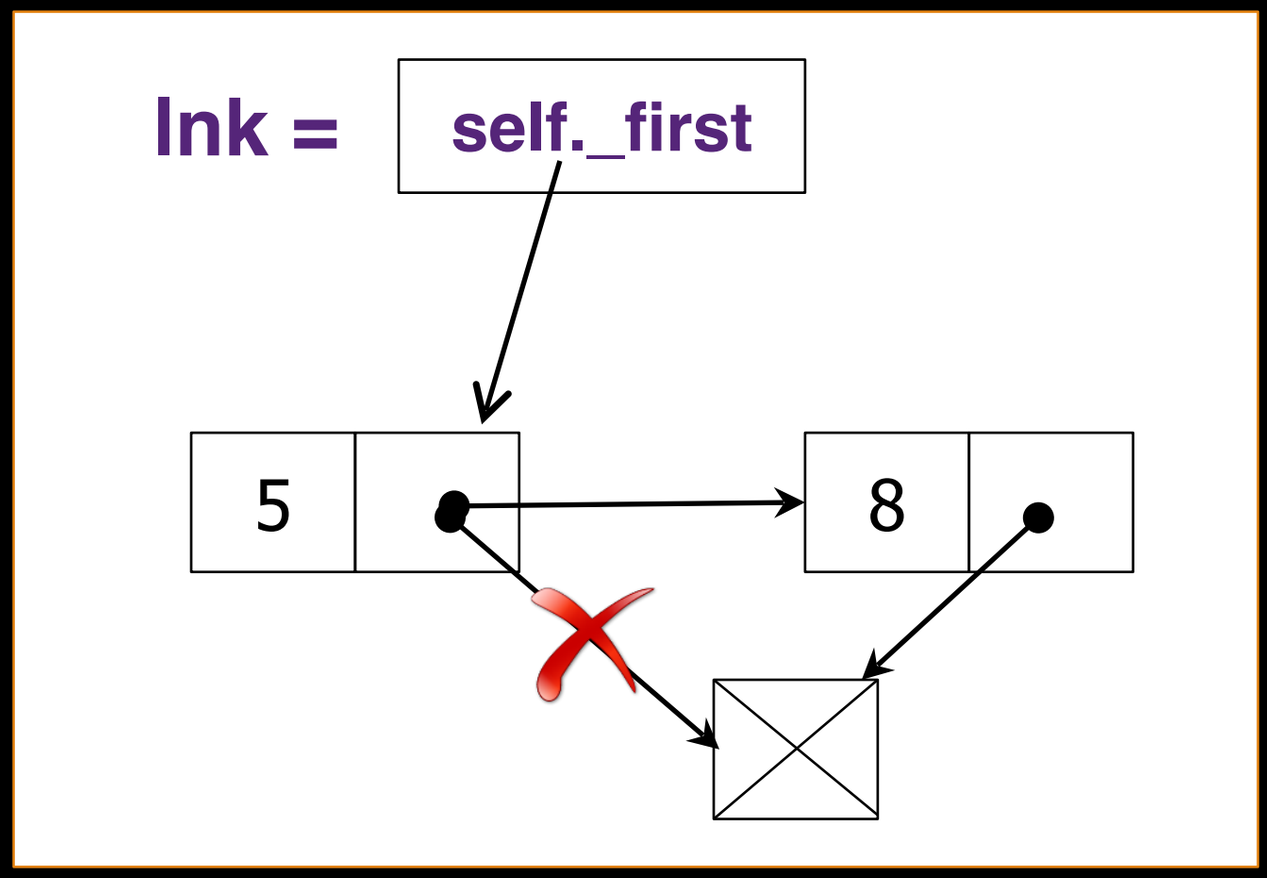
Insertion

append to 1 item list

Second node being appended

List has one element (5).

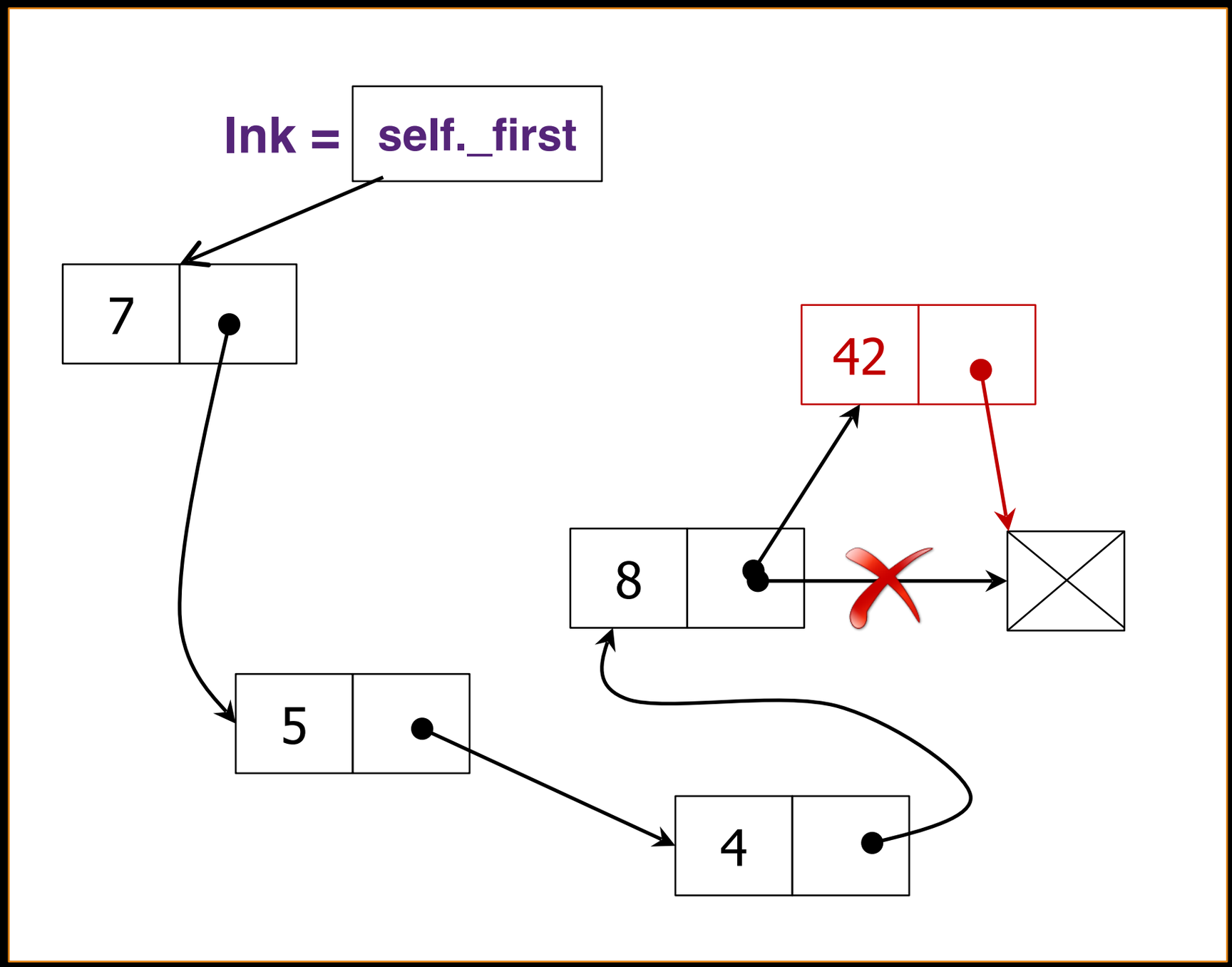
Appending a new node (8).



Always draw diagrams!

append to many item list

Insertion



Always draw diagrams!

Insertion

Insert at given index

Now let's work on writing the function to insert at a given index.

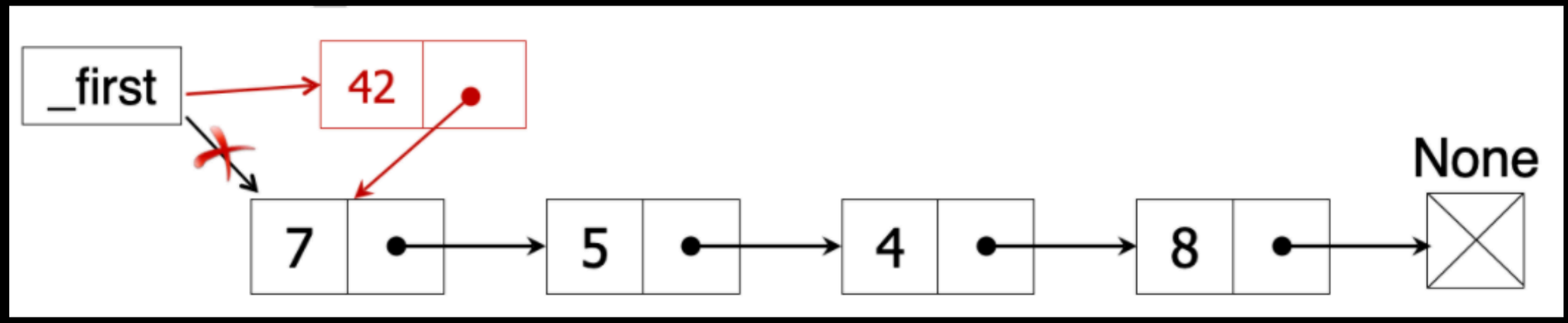
WORKSHEET

Linked List Insertion

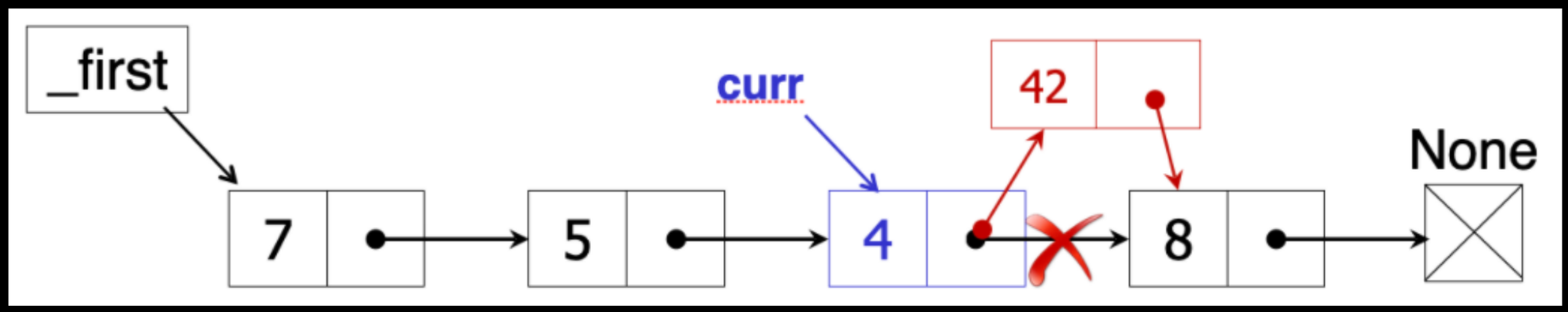
Insertion

Recapping the key ideas

1. Figure out when we need to modify `self._first` vs. a `_Node` in the list.



2. When `index` > 0, iterate to the (`index`-1)th node and update links.



Linked Lists

Deletion

Delete / pop

We might want to implement all sorts of delete variations depending on what operations we want the linked list to support, e.g.

- Delete from the front
- Delete from the back
- Delete (pop) from any index in the list

Ultimately, the "pop from an index" version covers all cases

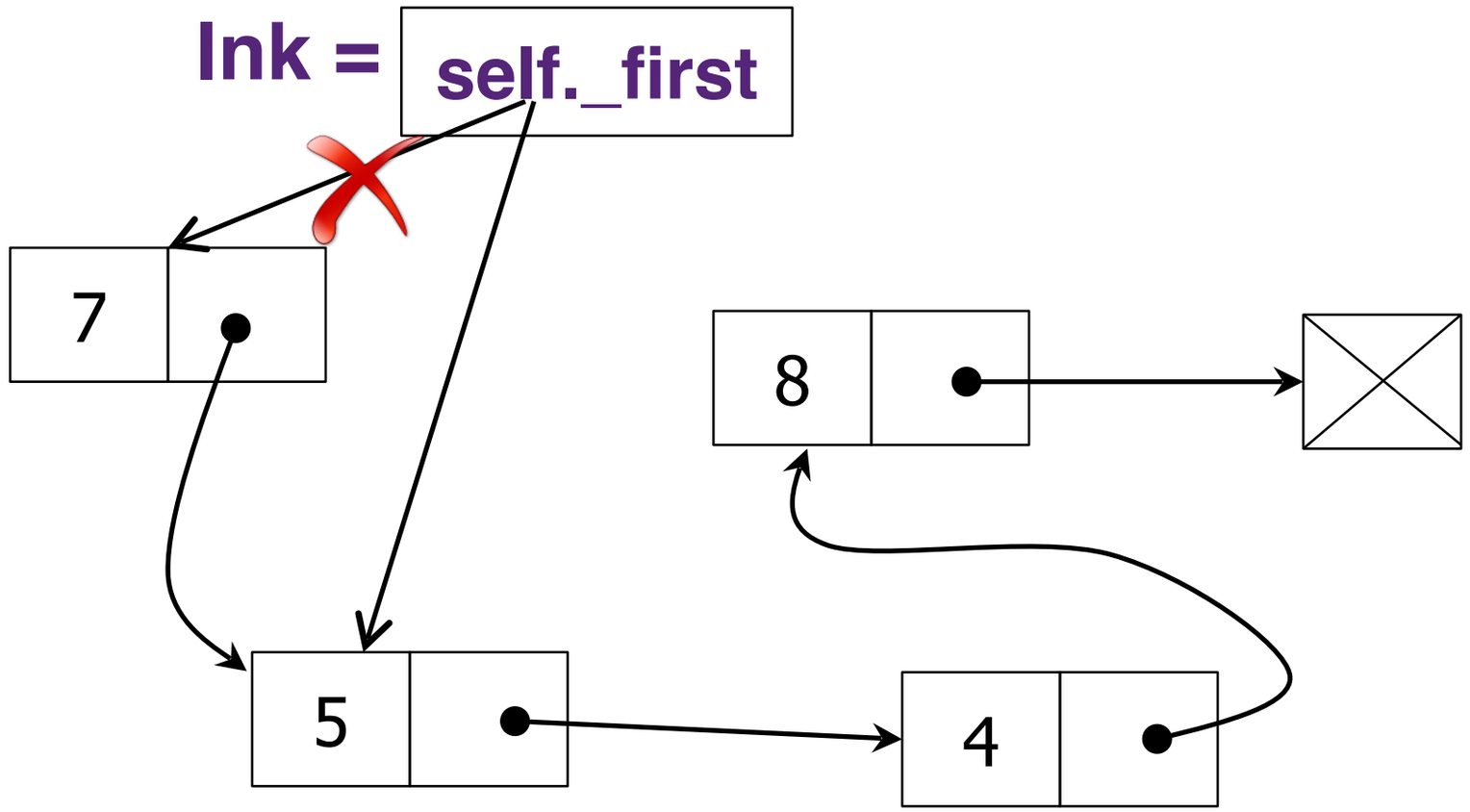
- But, let's visualize such operations first ...

Deletion

Delete from front

Easy: make _first reference the second node (garbage collection takes care of former first node automatically)

No need to walk the list ...



Deletion

Delete from front

Easy: make _first reference the second node (garbage collection takes care of former first node automatically)

Consider corner cases though:

What if only one node?

What if list is empty?

Delete at index

Deletion

```
def pop(self, index: int) -> Any:
    """Remove and return the item at the given
    index.

Raise IndexError if index >= len(self)
or index < 0.
"""
```

DEMO

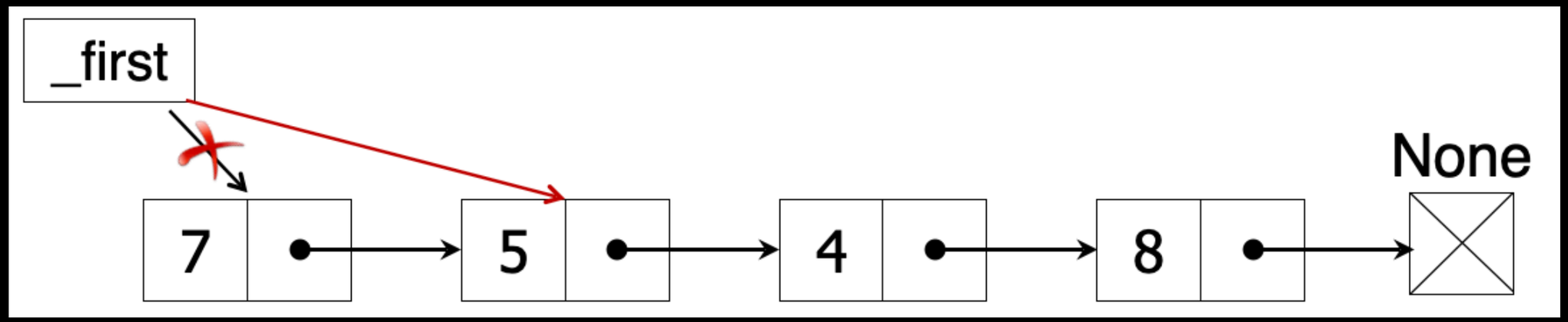
LinkedList.pop(self, index)

See thoughts_on_pop.pdf + linked_list.py

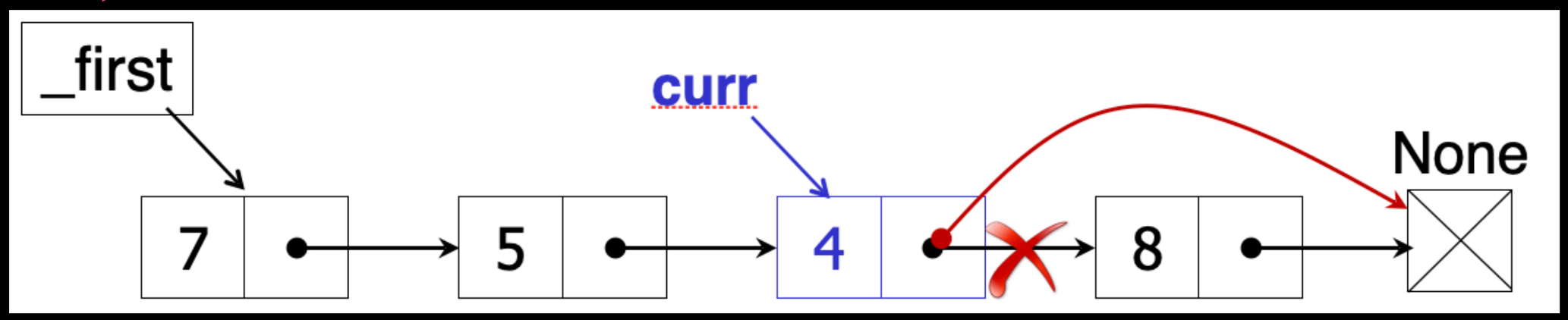
Same key ideas!

Deletion

1. Figure out when we need to modify `self._first` vs. a `_Node` in the list.



2. When `index` > 0, iterate to the (`index`-1)th node and update links.



Insertion/Deletion

The "problem of previous"

Strategy #1: iterate to the node *before* the desired position.

```
i = 0
curr = self._first
while not (curr is None or i == index - 1):
    curr = curr.next
    i += 1
```

The “problem of previous”

Strategy #2: track the previous node explicitly

```
i = 0
prev = None
curr = self._first
while not (curr is None or i == index):
    prev, curr = curr, curr.next
    i += 1
```

Practice

Add a remove method to our LinkedList class which removes the first occurrence of a given item.

see linked_list.py

Linked Lists Efficiency

Efficiency

Recall that...

Python's lists are array-based. Each list stores the ids of its elements in a contiguous block of memory.

Every insertion and deletion causes every element after the changed index to move.

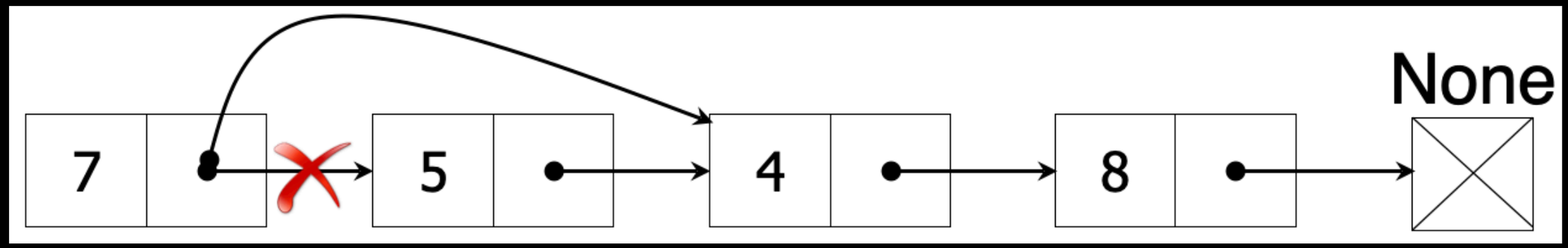
When analyzing running time, we use Big-Oh notation to capture the type of growth of running time as a function of input size.

E.g., O(1): "constant growth", O(n): "linear growth"

Efficiency

What about linked lists?

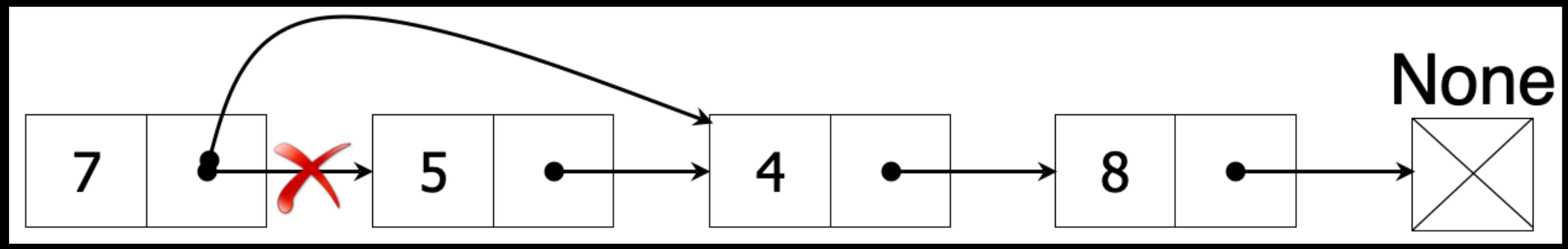
Remove an element from beginning – much faster (no need to "shift" anything)



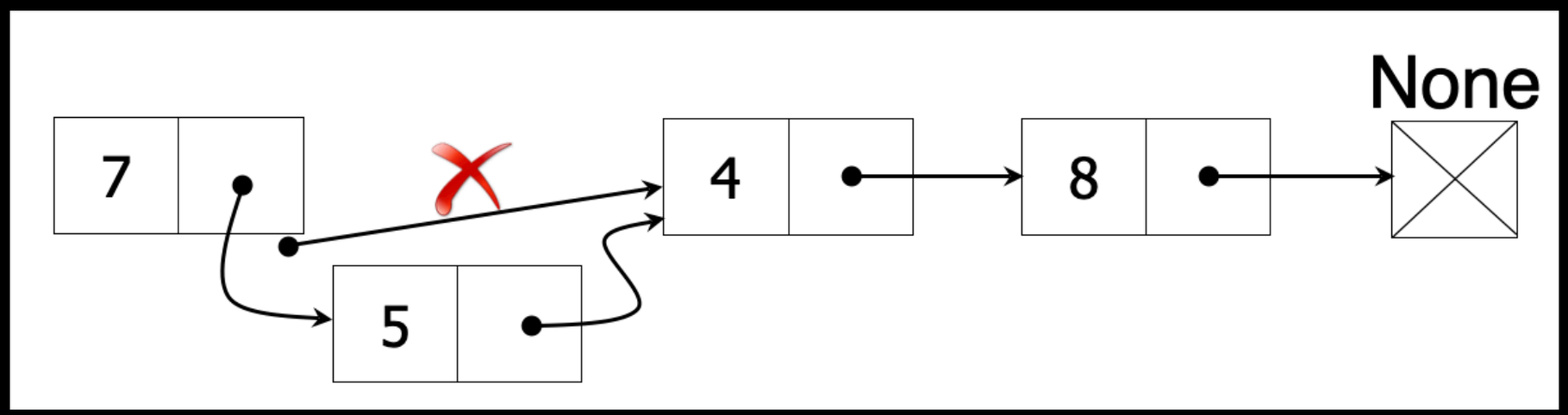
Efficiency

What about linked lists?

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



Insert an element near the beginning – no need to shift subsequent elements



Efficiency

What about linked lists?

Making changes to the beginning of a linked list:

Just adjust a couple of references, no moving memory!

Efficiency

Compare and contrast ...

At which end of a Python list would it be best to insert an element at? Which index would it be easiest to remove?

At which end of a linked list would it be best to insert an element at? Which index would it be easiest to remove?

Efficiency

LinkedList.__contains__

We care about running time as a function of input size:

“constant”	O(1)
“linear”	O(n)
“quadratic”	$O(n^2)$

WORKSHEET

Analyzing the Running Time of Linked List Operations