

Algorithms and Data Structures

Lists, Stacks & Queues (Ch 3)

Morgan Ericsson

Today

- » Abstract Data Type (ADT)
- » Lists
- » Stacks
- » Queues
- » Set / Bag

Abstract Data Type (ADT)

Abstract Data Type (ADT)

The concept ADT was introduced by Liskov and Zilles:

An abstract data type defines a class of abstract objects which is completely characterized by the operations available on those objects. This means that an abstract data type can be defined by defining the characterizing operations for that type. [...] Implementation information, such as how the object is represented in storage, is only needed when defining how the characterizing operations are to be implemented. The user of the object is not required to know or supply this information.

ADT

- » An ADT is a theoretical (mathematical) model
- » It defines the behavior of a data type, not its implementation
 - » The type is completely opaque from outside of its operations
- » The behavior is defined from the user's perspective, i.e., as a set of operations

ADT

- » A data type consists of a set of values or elements, the domain
 - » If finite, this can be specified via enumeration
 - » If not, via some rule describing the elements
- » and a set of operations
 - » Syntax, names and values
 - » Semantics, functionality / behavior

Example

- » Integer can be considered an abstract data type
 - » The domain is \mathbb{Z} (the integers)
- » The operations include
 - » $+$, $-$, \cdot , \div
 - » $=$, \neq , $<$, $>$, \geq , \leq
 - » ...

Semantics

- » The semantics can be specified in a few different ways
 - » Text
 - » Algebraic specification
 - » Operational specification

Example: Algebraic specification

Syntax for a Stack that can hold integers

<i>newstack</i>	:		\rightarrow	<i>STACK</i>
<i>push</i>	:	$STACK \times INT$	\rightarrow	<i>STACK</i>
<i>pop</i>	:	<i>STACK</i>	\rightarrow	<i>STACK</i>
<i>top</i>	:	<i>STACK</i>	\rightarrow	$INT \cup \{Error\}$
<i>empty</i>	:	<i>STACK</i>	\rightarrow	<i>BOOLEAN</i>

Example: Algebraic specification

Semantics for a Stack that can hold integers

$$\text{empty}(\text{newstack}) = \text{True}$$

$$\text{empty}(\text{push}(s, i)) = \text{False}$$

$$\text{pop}(\text{newstack}) = \text{new}$$

$$\text{pop}(\text{push}(s, i)) = s$$

$$\text{top}(\text{new}) = \text{Error}$$

$$\text{top}(\text{push}(s, i)) = i$$

where $s \in \text{STACK}$ and $i \in \text{INT}$.

Abstract vs Concrete (DT)

- » An ADT is a model, not an implementation
- » It is a theoretical tool, not a language construct
- » The model can be implemented in multiple ways
 - » Sometimes with restrictions

Restrictions

Numpy offers an implementation of the vector ADT with several operations.

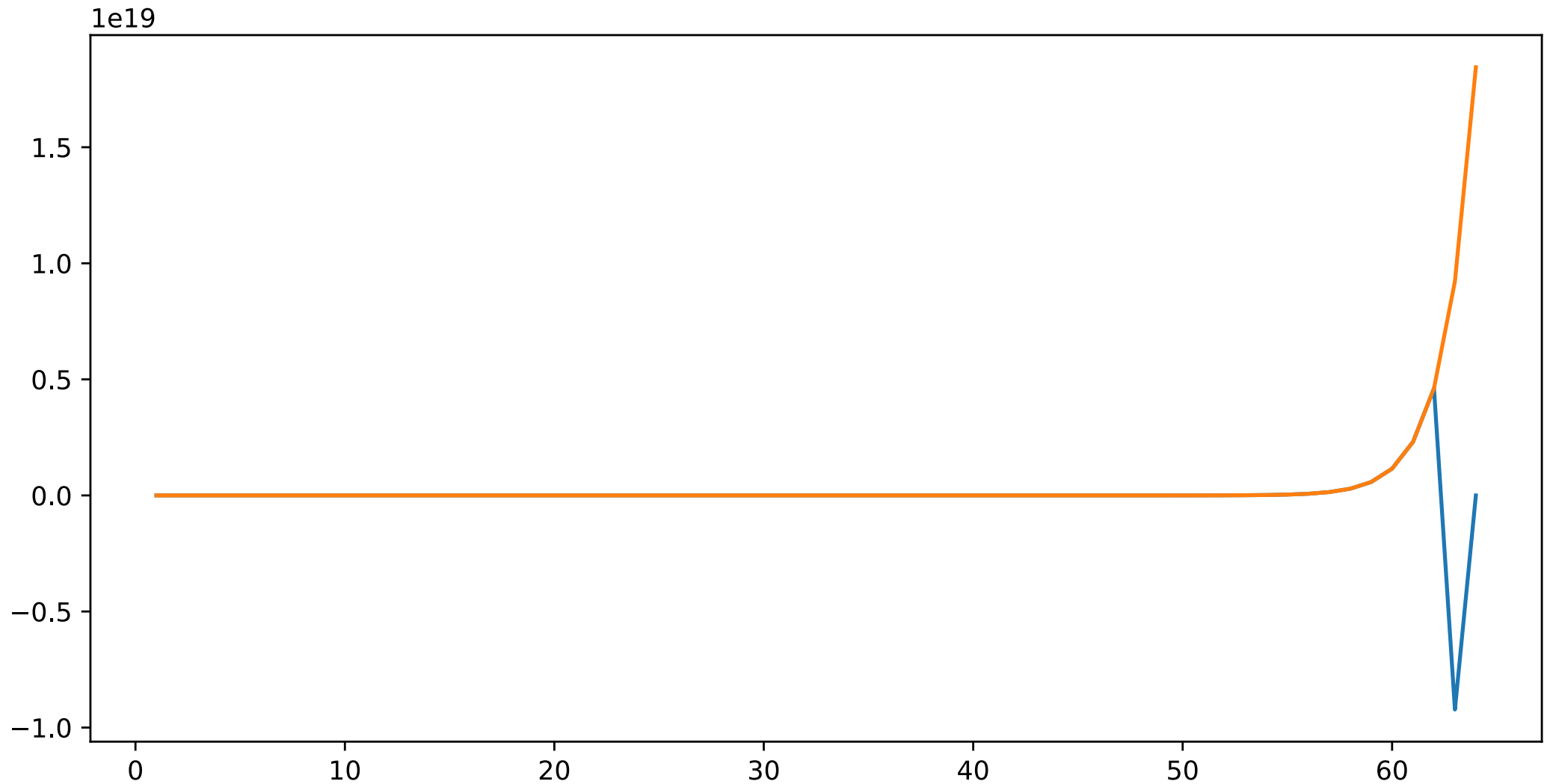
```
1 import numpy as np
2
3 m1 = np.arange(10)
4 m2 = m1 + 2
5 m3 = m2 ** 2
6 print(m3)
```

```
[ 4  9 16 25 36 49 64 81 100 121]
```

Restrictions

```
1 import matplotlib.pyplot as plt
2 import seaborn as sns
3 import numpy as np
4
5 x1 = np.arange(1, 65)
6 x2 = list(range(1, 65))
7
8 sns.lineplot(x1, 2**x1)
9 sns.lineplot(x2, [x**2 for x in x2])
10 plt.show()
```

Restrictions



Restrictions

```
1 x = np.arange(62, 65)
2 print(2**x)
3 print(2**63)
4 print(type(x[0]))
5 print(type(2))
```

```
[ 4611686018427387904 -9223372036854775808
0]
9223372036854775808
<class 'numpy.int64'>
<class 'int'>
```

Restrictions

- » The range of `numpy.int64` is $[-2^{63}, 2^{63} - 1]$
 - » So, 2^{63} is too large to fit
 - » Overflow makes it wrap around, so $2^{63} = -2^{63}$
 - » $2^{64} = -2^{63} + 2^{63} = 0$
- » The range of Python `int` is unbounded
 - » So we can represent 2^{1024} , for example
- » `int` and `numpy.int64` are two implementations of the integer ADT

Lists

The List ADT

- » A list is a finite, ordered sequence of elements
 - » Note that ordered means each element has a position, not that the list elements are sorted by value
- » A list of elements A_0, \dots, A_{N-1} has length N . An empty list has length 0
- » We can imagine various operations on a list, such as add, delete, get, etc.

A simple list

- » What do we need?
 - » Create a new list
 - » Empty
 - » Count / len
 - » Add value
- » To keep it simple, we only allow integers in our list
- » We will base our implementation on arrays

A sketch of the syntax

- » `init()`
- » `empty()` \rightarrow `bool`
- » `count()` \rightarrow `int`
- » `append(i:int)` \rightarrow `None`

A sketch of the semantics

```
1  sl = SimpleList()  
2  assert sl.empty()  
3  
4  sl.append(1)  
5  sl.append(2)  
6  sl.append(3)  
7  
8  assert not sl.empty()  
9  assert sl.count() == 3
```

A simple list

```
1 class SimpleList():
2     def __init__(self) -> None:
3         self.lst = np.empty(10, dtype=np.int64)
4         self.cnt = 0
```

Note

- » We use `numpy.empty` to simulate memory that we can use for our list
 - » It makes no sense to use a Python list to implement a list
- » In C, we would use `malloc` or `calloc` to create the memory
- » In Java, we can use normal arrays

A simple list

```
1 from fastcore.basics import patch
2
3 @patch
4 def empty(self:SimpleList) -> bool:
5     return self.cnt == 0
6
7 @patch
8 def count(self:SimpleList) -> int:
9     return self.cnt
```


A simple list

```
1 @patch
2 def append(self:SimpleList, d:int) -> None:
3     self.lst[self.cnt] = d
4     self.cnt += 1
```

Checking semantics

```
1  sl = SimpleList()  
2  assert sl.empty()  
3  
4  sl.append(1)  
5  sl.append(2)  
6  sl.append(3)  
7  
8  assert not sl.empty()  
9  assert sl.count() == 3  
10  
11 print(sl.count())
```

3

Extending the ADT

```
1 @patch
2 def delete(self:SimpleList, d:int) -> None:
3     found = False
4     for i in range(0, self.cnt):
5         if self.lst[i] == d:
6             found = True
7             break
8     if found:
9         for j in range(i+1, self.cnt):
10             self.lst[j-1] = self.lst[j]
11         self.cnt -= 1
```

Checking semantics

```
1  sl = SimpleList()
2
3  assert sl.empty() == True
4
5  sl.append(1)
6  sl.append(2)
7
8  assert sl.count() == 2
9
10 sl.delete(1)
11
12 assert sl.count() == 1
13
14 sl.delete(2)
15
16 assert sl.empty() == True
```

Alternative idea: free list

```
1 class SimpleListFL():
2     def __init__(self) -> None:
3         self.lst = np.empty(10, dtype=np.int64)
4         self.fl = np.ones(10, dtype=bool)
```

Alternative idea

```
1 @patch
2 def delete(self:SimpleListFL, d:int) -> None:
3     for ix, v in enumerate(self.lst):
4         if v == d:
5             self.fl[ix] = True
```

Alternative idea

```
1 @patch
2 def append(self:SimpleListFL, d:int) -> None:
3     for ix, v in enumerate(self.fl):
4         if v == True:
5             self.lst[ix] = d
6             self.fl[ix] = False
7         break
```

Alternative idea

```
1 @patch
2 def count(self:SimpleListFL) -> None:
3     s = 0
4     for v in self.fl:
5         if v == False:
6             s += 1
7     return s
8
9 @patch
10 def empty(self:SimpleListFL) -> None:
11     for v in self.fl:
12         if v == False:
13             return False
14     return True
```


Checking semantics

```
1  sl = SimpleListFL()
2
3  assert sl.empty() == True
4
5  sl.append(1)
6  sl.append(2)
7
8  assert sl.count() == 2
9
10 sl.delete(1)
11
12 assert sl.count() == 1
13
14 sl.delete(2)
15
16 assert sl.empty() == True
```




Making it more Python

```
1 @patch
2 def __len__(self:SimpleList) -> int:
3     return self.cnt
```

Making it more Python

```
1  s1 = SimpleList()  
2  
3  assert s1.empty() == True  
4  
5  s1.append(1)  
6  s1.append(2)  
7  s1.append(3)  
8  
9  assert len(s1) == 3
```

Python typing

- » Duck typing
 - » “If it walks like a  and it quacks like a , then it must be a ”
 - » An object is given a type if it has all methods and properties required by that type
- » Structural subtyping (static duck typing)
 - » Functionality is defined by protocols, e.g., `Sized`
 - » which require certain methods, e.g., `__len__()`
`-> int`

More than ten elements?

```
1 sl = SimpleList()  
2  
3 for i in range(15):  
4     sl.append(i)
```

IndexError

index 10 is out of bounds for axis 0 with size 10

A new append

```
1 @patch
2 def append(self:SimpleList, d:int) -> None:
3     if self.cnt == self.sz:
4         self.sz += 1
5         tmp = np.empty(self.sz, dtype=np.int64)
6         for i in range(self.cnt):
7             tmp[i] = self.lst[i]
8         self.lst = tmp
9
10    self.lst[self.cnt] = d
11    self.cnt += 1
```

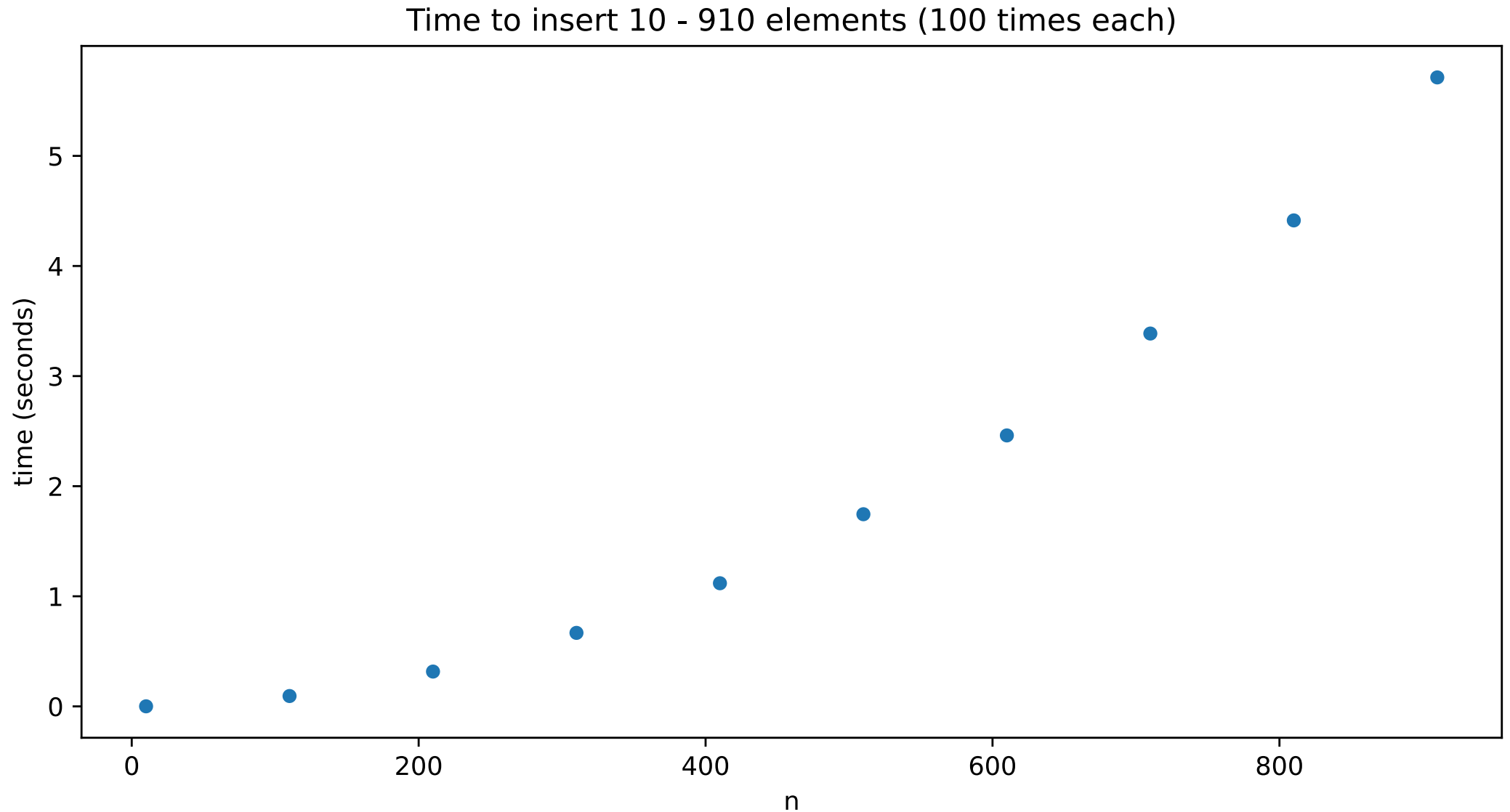
We need to fix init

```
1 @patch
2 def __init__(self:SimpleList) -> None:
3     self.lst = np.empty(10, dtype=np.int64)
4     self.sz = 10
5     self.cnt = 0
```

Again

```
1  sl = SimpleList()  
2  
3  for i in range(15):  
4      sl.append(i)  
5  
6  assert sl.count() == 15
```

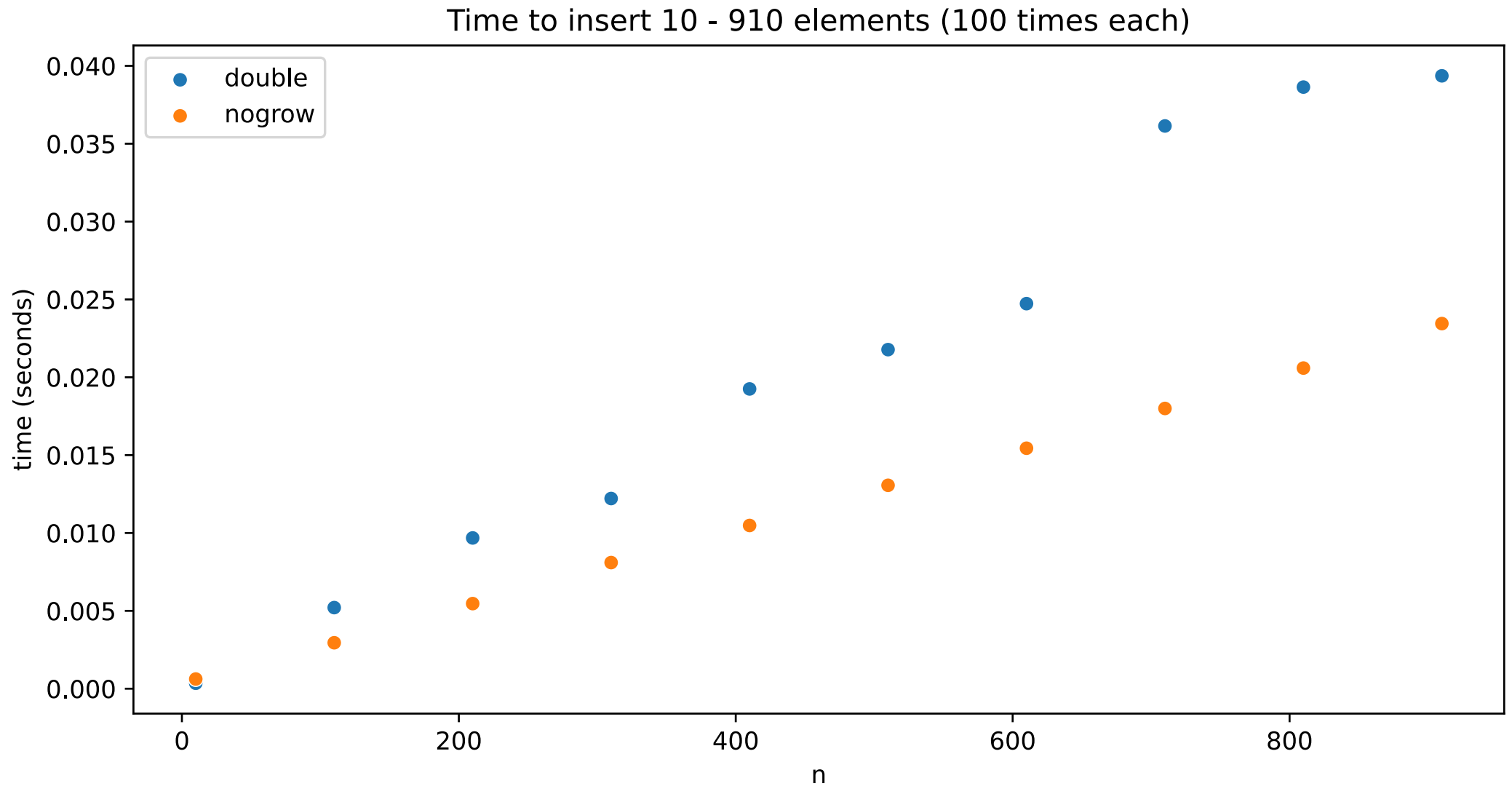

Problems?



A better approach

```
1 @patch
2 def append(self:SimpleList, d:int) -> None:
3     if self.cnt == self.sz:
4         self.sz *= 2
5         tmp = np.empty(self.sz, dtype=np.int64)
6         for i in range(self.cnt):
7             tmp[i] = self.lst[i]
8         self.lst = tmp
9
10    self.lst[self.cnt] = d
11    self.cnt += 1
```

Problems?



Problems

- » There is a cost to resizing the list
 - » Mainly due to copying
- » We can not get around this at this point, without limiting the maximum size
 - » But we can be smarter about when and how to extend
- » We should probably reduce the size when deleting enough elements

Another list

```
1 class LinkedList():  
2     def __init__(self) -> None:  
3         self.lst = None
```

Another list

```
1 @patch
2 def append(self:LinkedList, d:int) -> None:
3     if self.lst is None:
4         self.lst = LLNode(d)
5     else:
6         p = self.lst
7         while p.nxt is not None:
8             p = p.nxt
9         p.nxt = LLNode(d)
```

LLNode

```
1 from dataclasses import dataclass
2
3 @dataclass
4 class LLNode:
5     val: int
6     nxt: 'LLNode|None' = None
```

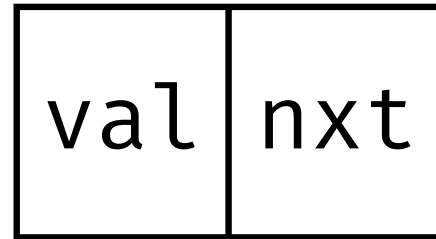
Does it work?

```
1 ll = LinkedList()  
2 ll.append(1)  
3 ll.append(2)  
4  
5 print(ll.lst)
```

```
LLNode(val=1, nxt=LLNode(val=2, nxt=None))
```

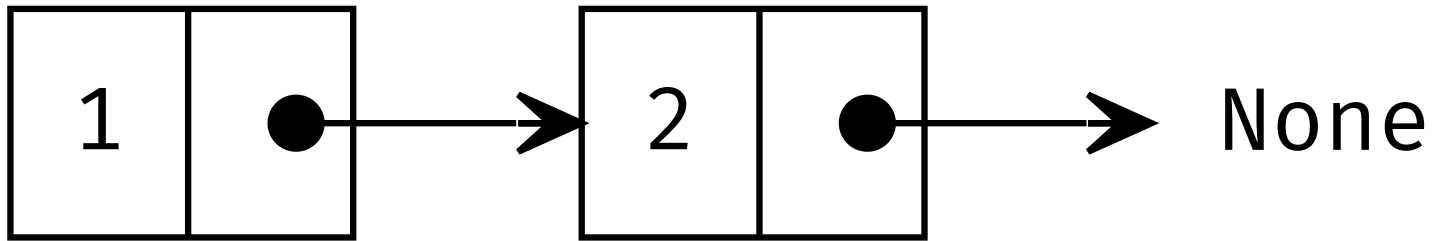

How does it work?

LLNode



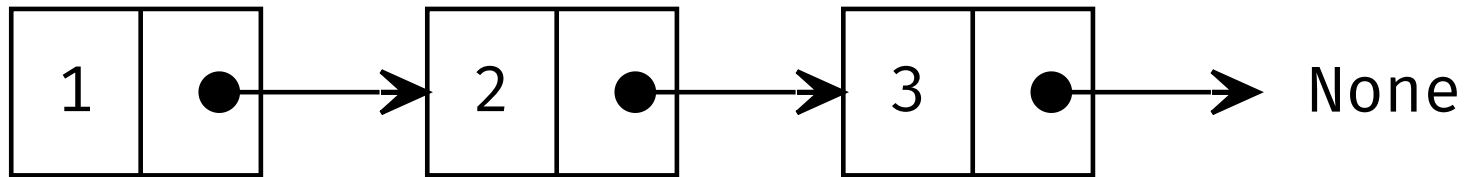
How does it work?

The LinkedList after appending the values 1 and 2.



How does it work?

The LinkedList after appending 3



Nicer print

```
1 @patch
2 def __str__(self:LinkedList) -> str:
3     s = '['
4     p = self.lst
5     while p is not None:
6         s += f' {p.val}, '
7         p = p.nxt
8     if s == '[':
9         return '[' ]'
10    else:
11        return s[:-1] + ' ]'
```

Does it work?

```
1 ll = LinkedList()  
2 ll.append(1)  
3 ll.append(2)  
4 ll.append(3)  
5  
6 print(ll)
```

```
[ 1, 2, 3 ]
```

Adding the other operations

```
1 @patch
2 def empty(self:LinkedList) -> bool:
3     return self.lst is None
```

Adding the other operations

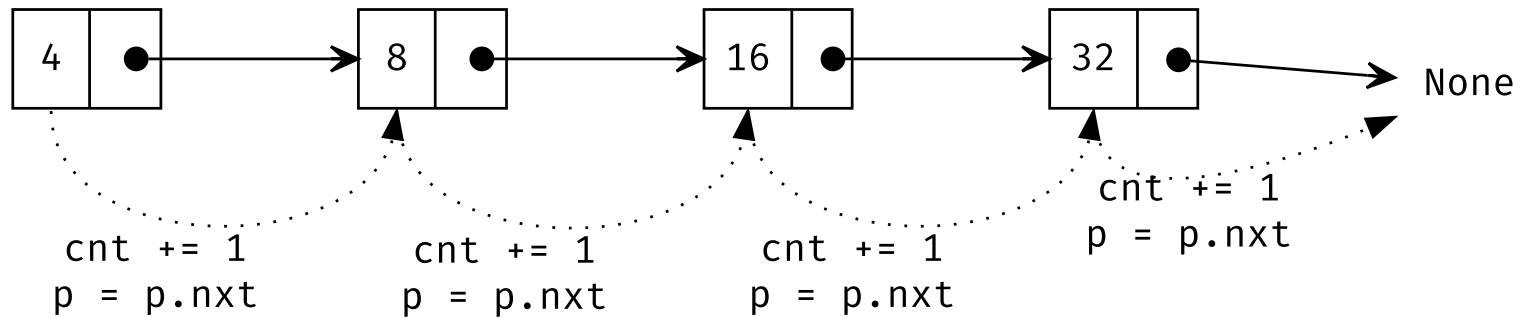
```
1 @patch
2 def __len__(self:LinkedList) -> int:
3     cnt, p = 0, self.lst
4     while p is not None:
5         cnt += 1
6         p = p.nxt
7
8     return cnt
```

Does it work?

```
1 ll = LinkedList()  
2  
3 assert ll.empty() == True  
4  
5 ll.append(1)  
6 ll.append(2)  
7 ll.append(3)  
8  
9 assert len(ll) == 3
```


How does it work?

Count of a list



Adding the other operations

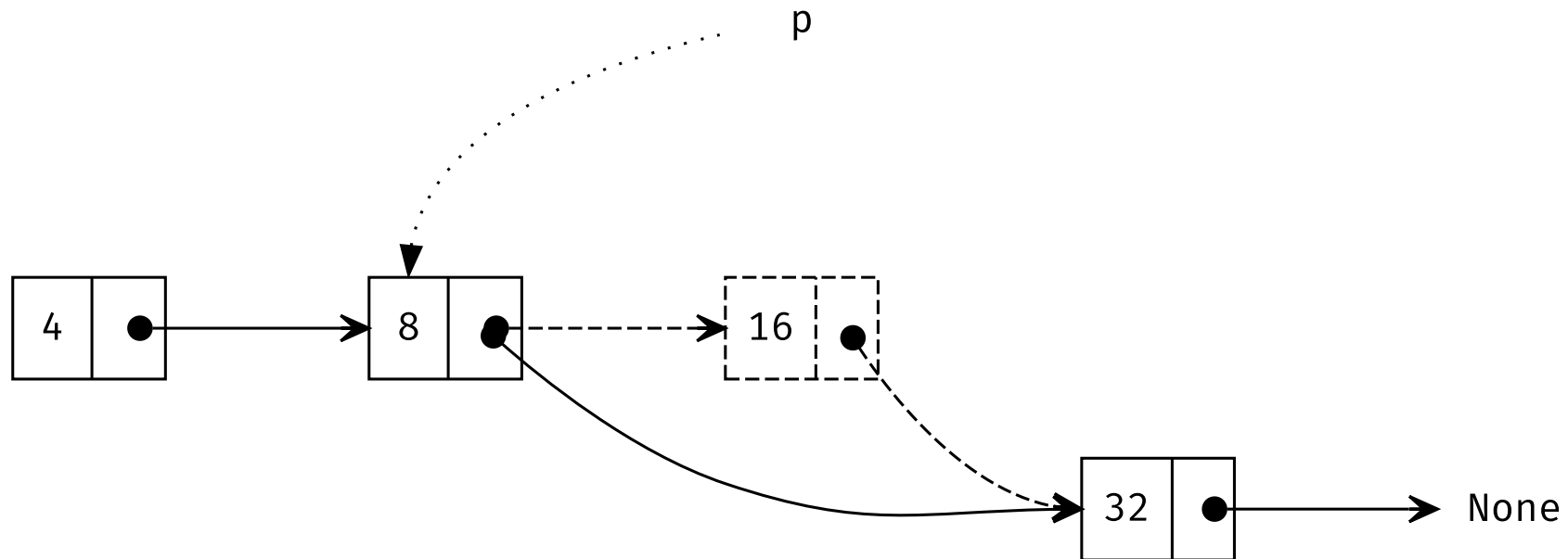
```
1  @patch
2  def delete(self:LinkedList, d:int) -> None:
3      if self.lst is not None:
4          if self.lst.val == d:
5              if self.lst.nxt is None:
6                  self.lst = None
7              else:
8                  self.lst = self.lst.nxt
9          else:
10             p = self.lst
11             while p.nxt is not None and p.nxt.val != d:
12                 p = p.nxt
13             if p.nxt is not None:
14                 p.nxt = p.nxt.nxt
```

Checking delete

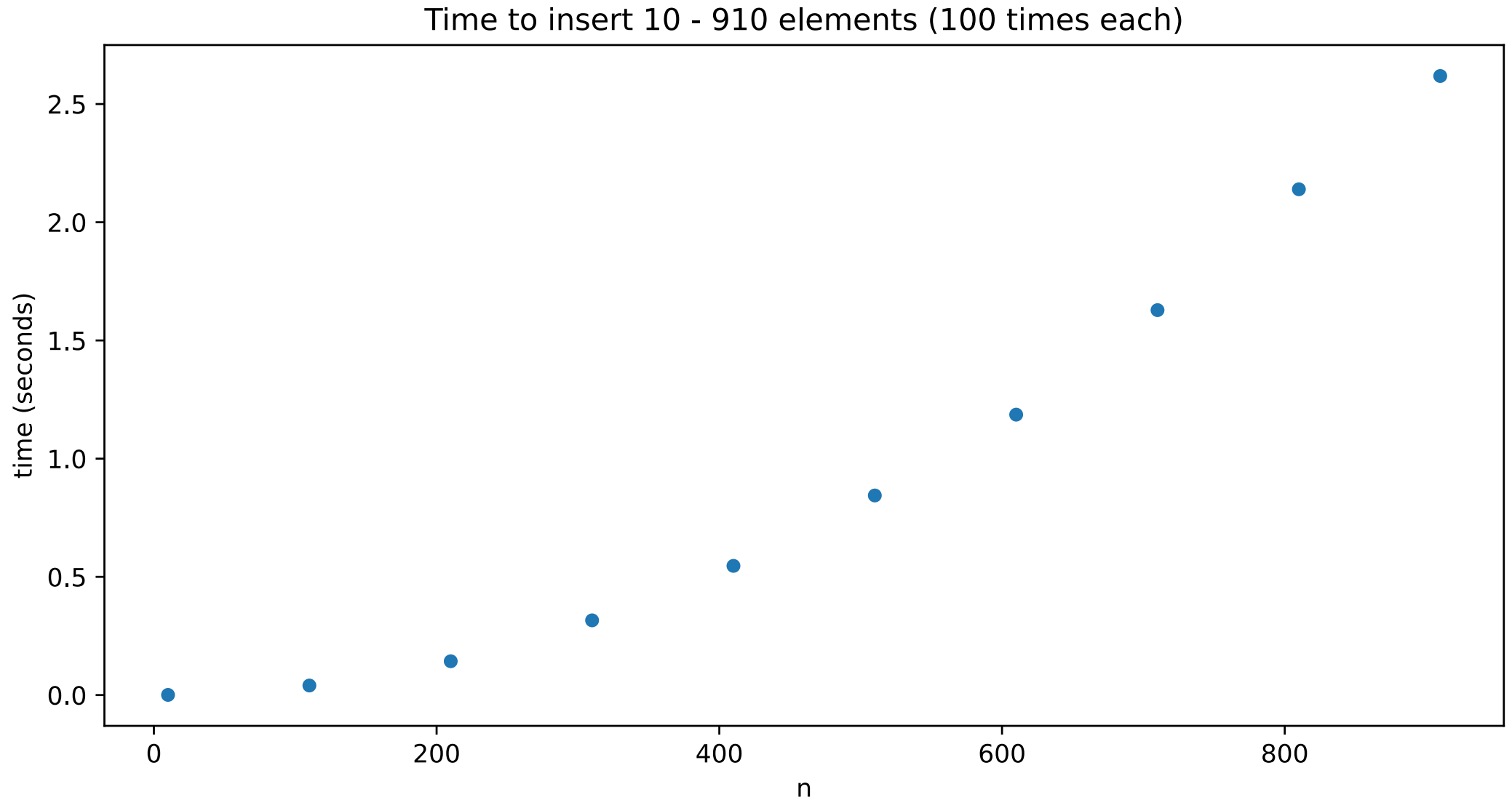
```
1 ll = LinkedList()
2
3 assert ll.empty() == True
4
5 ll.append(1)
6 ll.append(2)
7
8 assert len(ll) == 2
9
10 ll.delete(1)
11
12 assert len(ll) == 1
13
14 ll.delete(2)
15
16 assert ll.empty() == True
```

How does it work?

Deleting node with value 16



Performance?



Fix?

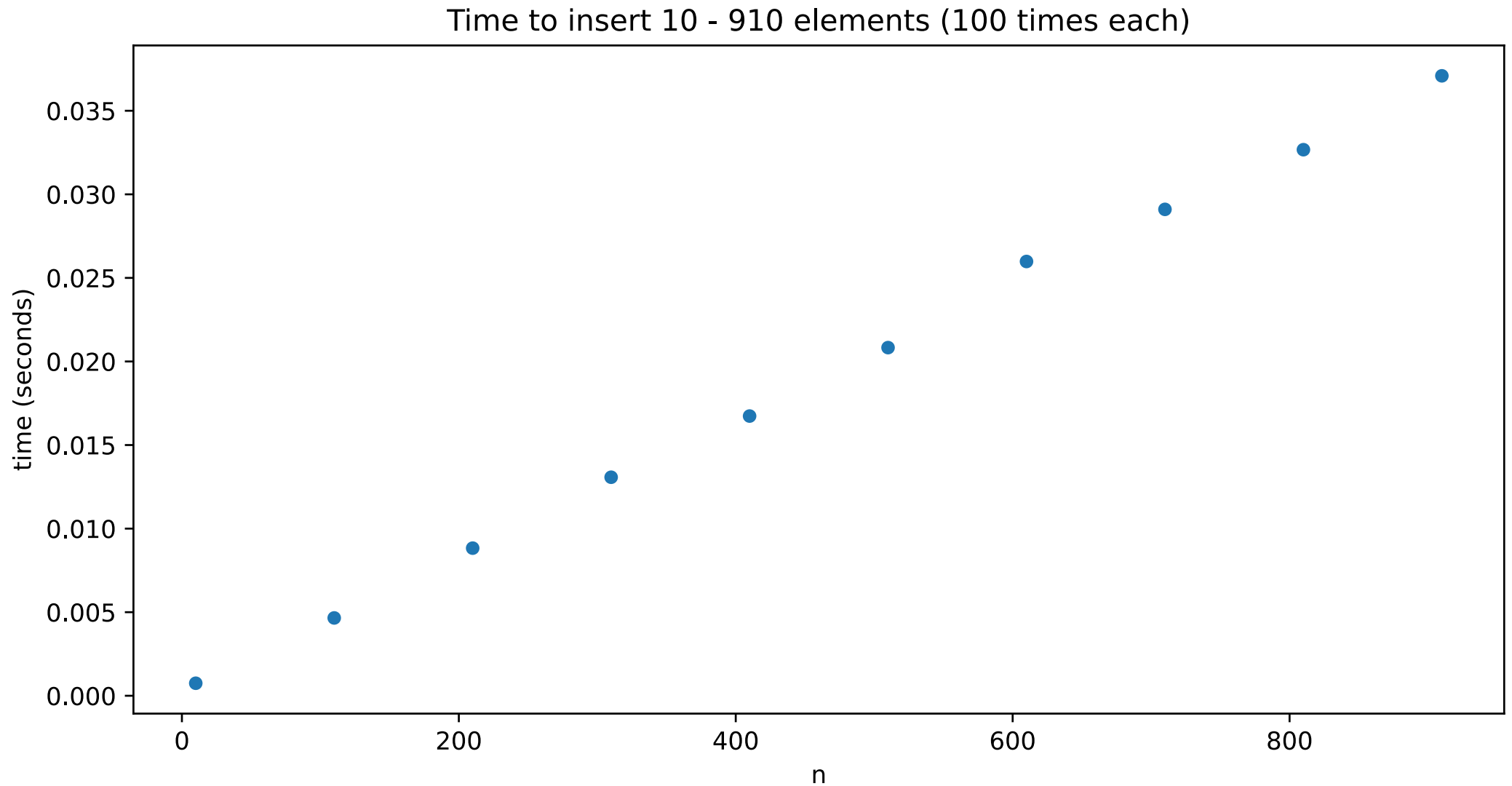
```
1 @patch
2 def append(self:LinkedList, d:int) -> None:
3     if self.lst is None:
4         self.lst = LLNode(d)
5         self.last = self.lst
6     else:
7         self.last.nxt = LLNode(d)
8         self.last = self.last.nxt
```

Updating the constructor

```
1 def __init__(self:LinkedList) -> None:
2     self.lst = None
3     self.last = None
```

- » Note, we also need to update `delete` to ensure that `self.last` points to the correct element if the last element is deleted

Better?



Alternatives

- » We could change to insert at front
 - » No change if order does not matter
 - » If order matters, we need a pointer to the beginning
 - » ... and pointers in both directions (double linked)

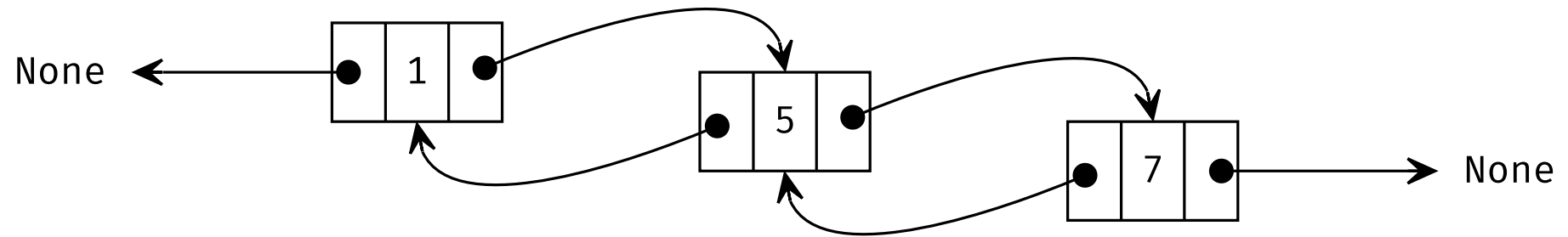
DLLNode

```
1 @dataclass
2 class DLLNode:
3     val: int
4     prv: 'LLNode | None' = None
5     nxt: 'LLNode | None' = None
```

Double linked

- » Double linked makes it possible to move in both directions
- » Can make some operations easier, e.g.,
 - » delete
- » But more references (pointers) to keep track of
- » ... and they use more memory (one vs two pointers)

Double linked list



Iterators

- » An iterator is an object that allows you to traverse something, e.g., a list
- » Used by `for` loops in Python
 - » `for v in mylist` uses the iterator of the Python List
- » We can add iterators to our classes

Iterator for LinkedList

```
1 class LLIterator:
2     def __init__(self, head:LLNode) -> None:
3         self.current = head
```

Iterator for LinkedList

```
1 @patch
2 def __iter__(self:LinkedList) -> LLIterator:
3     return LLIterator(self.lst)
```

Iterator for LinkedList

```
1 @patch
2 def __iter__(self:LLIterator) -> LLIterator:
3     return self
```


Iterator for LinkedList

```
1 @patch
2 def __next__(self:LLIterator) -> int:
3     if self.current is None:
4         raise StopIteration
5     else:
6         tmp = self.current.val
7         self.current = self.current.nxt
8         return tmp
```

Testing

```
1 ll = LinkedList()  
2 for i in range(10):  
3     ll.append(i)  
4  
5 s = 0  
6 for v in ll:  
7     s += v  
8  
9 assert s == 45
```

Enables some cool stuff

```
1 ll = LinkedList()  
2 for i in range(10):  
3     ll.append(i**2)  
4  
5 s = sum(ll)  
6  
7 assert s == 285
```

How does it work?

```
1 ll = LinkedList()
2 for i in range(10):
3     ll.append(i**3)
4
5 s = 0
6 it = iter(ll)
7 while True:
8     try:
9         s += next(it)
10    except StopIteration:
11        break
12
13 assert s == 2025
```

Iterators

- » We can of course add an iterator to SimpleList as well, but it's trivial

```
1 @patch
2 def __next__(self:SLIterator) -> int:
3     if self.pos == self.cnt:
4         raise StopIteration
5     else:
6         tmp = self.lst[pos]
7         self.pos += 1
8         return tmp
```

Accessing a specific position

```
1 @patch
2 def __getitem__(self:SimpleList, key:int) -> int:
3     if key < self.cnt:
4         return self.lst[key]
5     else:
6         raise IndexError
```

Checking

```
1 sl = SimpleList()
2 for i in range(10):
3     sl.append(i**2)
4
5 assert sl[5] == 25
```

Trickier in LinkedList

```
1 @patch
2 def __getitem__(self:LinkedList, key:int) -> int:
3     cpos = 0
4     p = self.lst
5     while p is not None:
6         if cpos == key:
7             return p.val
8         cpos += 1
9         p = p.nxt
10    raise IndexError
```


Checking

```
1 ll = LinkedList()  
2 for i in range(10):  
3     ll.append(i**2)  
4  
5 assert ll[5] == 25
```

Setting and deleting

- » We can implement `__setitem__` and `__delitem__` to allow the user to set or remove a specific item
 - » Similar to `__getitem__`, just do `self.lst[key] = v` (basically)
- » In reality, indexing requires a bit more work in Python
 - » Slicing does not work in our implementation
 - » But, that's for a course on Python, not algorithms...

Summary

- » List is an ADT for ordered elements
- » There is a wide range of possible operations
 - » We have looked at counting, adding, removing, getting and iterating.
- » There can be many different implementations
 - » Linked list, array (memory) based, etc.
 - » With different tradeoffs
 - » Insert in an ideal memory-based list is $O(1)$ and $O(N)$ in the trivial linked list

Stacks

Stack

- » A stack is an ADT where values are inserted and removed from the “top”
 - » Similar to a stack of items in the real world
- » LIFO (Last In, First Out)
- » Two main operations are *push* and *pop* (insert and remove)
- » Many uses in computer science
 - » E.g., to handle function calls when running programs

Implementing a Stack with a linked list

» We will follow the ADT defined previously, so the operations are

» `__init__(newstack)`

» `push(v:int) -> None`

» `pop() -> None`

» `top() -> int|None`

» `empty() -> bool`

Implementing a Stack with a linked list

```
1 stack = LLStack()  
2 assert stack.empty() == True  
3  
4 stack.push(1)  
5 assert stack.empty() == False  
6 assert stack.top() == 1  
7  
8 stack.pop()  
9 assert stack.empty() == True
```

This is not executing yet, since we have not implemented **LLStack**

Implementing a Stack with a linked list

```
1 class LLStack:
2     def __init__(self) -> None:
3         self.stack = None
4
5     def empty(self) -> bool:
6         return self.stack is None
7
8     def top(self) -> int|None:
9         if self.stack is None:
10             return None
11         else:
12             return self.stack.val
```


Implementing a Stack with a linked list

```
1 @patch
2 def push(self:LLStack, v:int) -> None:
3     tmp = LLNode(v)
4     tmp.nxt = self.stack
5     self.stack = tmp
```

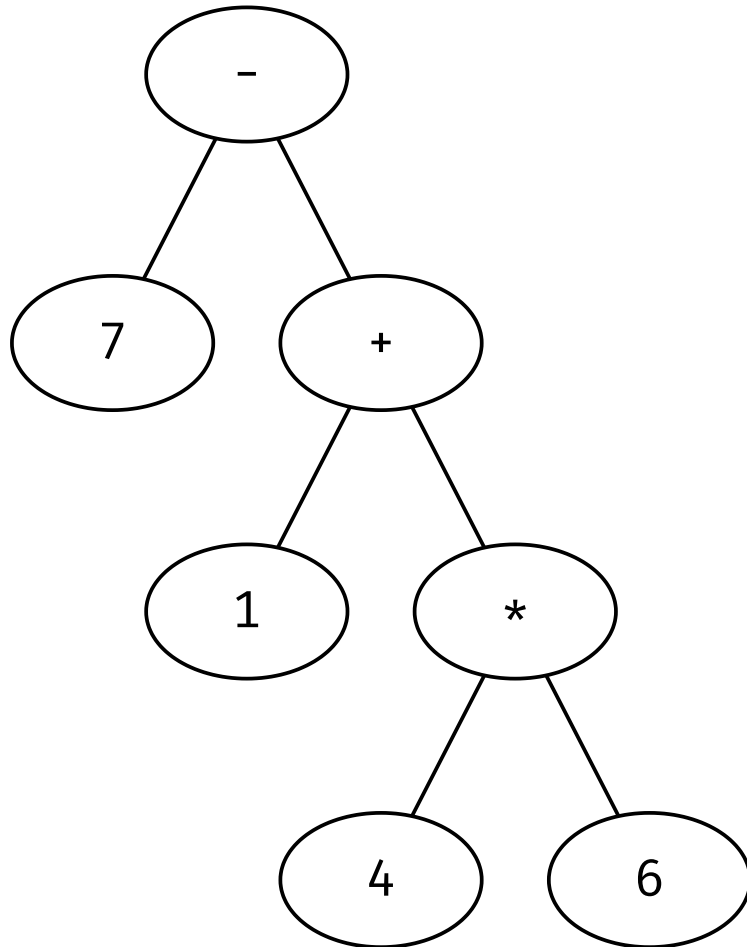
Implementing a Stack with a linked list

```
1 @patch
2 def pop(self:LLStack) -> None:
3     if self.stack is not None:
4         self.stack = self.stack.nxt
```

Testing it

```
1 stack = LLStack()  
2 assert stack.empty() == True  
3  
4 stack.push(1)  
5 assert stack.empty() == False  
6 assert stack.top() == 1  
7  
8 stack.pop()  
9 assert stack.empty() == True
```

Using the stack: calculator



- » Suppose we want to compute $1 + 4 \cdot 6 - 7$
- » We can use a stack to represent the tree

Using the stack: calculator

```
1 from enum import IntEnum
2
3 class Op(IntEnum):
4     ADD = 1
5     DIV = 2
6     MUL = 3
7     SUB = 4
```

Using the stack: calculator

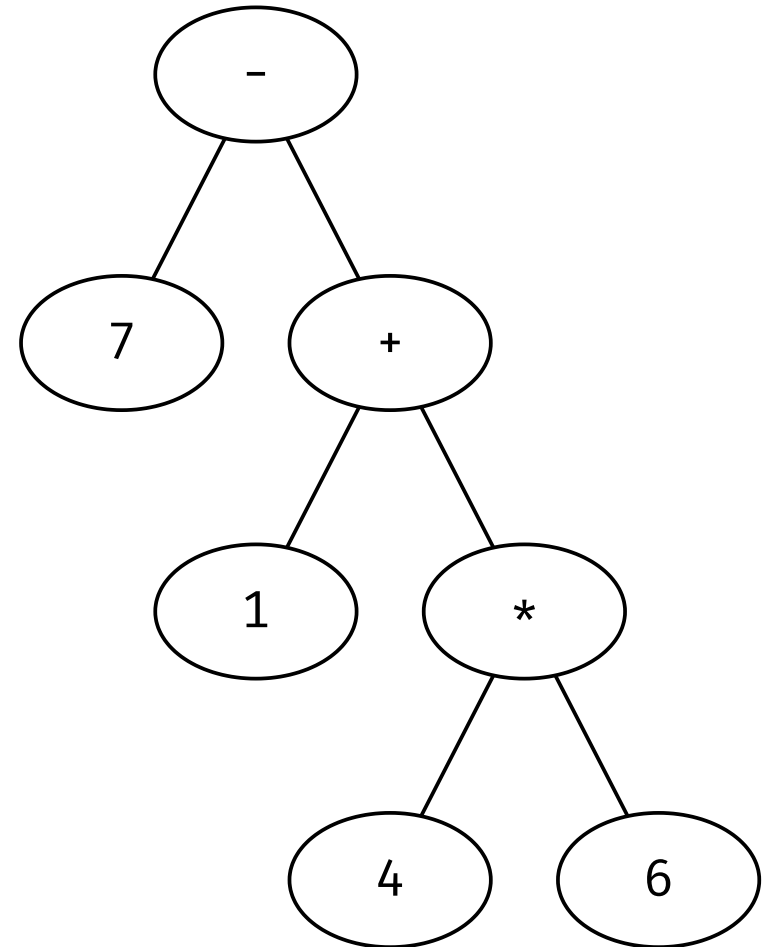
```
1  def calc(ops:LLStack, dig:LLStack) -> int:
2      while not ops.empty():
3          o = Op(ops.top())
4          d1 = dig.top()
5          dig.pop()
6          d2 = dig.top()
7          dig.pop()
8          dig.push(doop(o, d1, d2))
9          ops.pop()
10     tmp = dig.top()
11     dig.pop()
12     return tmp
```

Using the stack: calculator

```
1  def doop(op:Op, d1:int, d2:int) -> int:
2      match op:
3          case Op.ADD:
4              return d1 + d2
5          case Op.DIV:
6              return d1 / d2
7          case Op.MUL:
8              return d1 * d2
9          case Op.SUB:
10             return d1 - d2
```

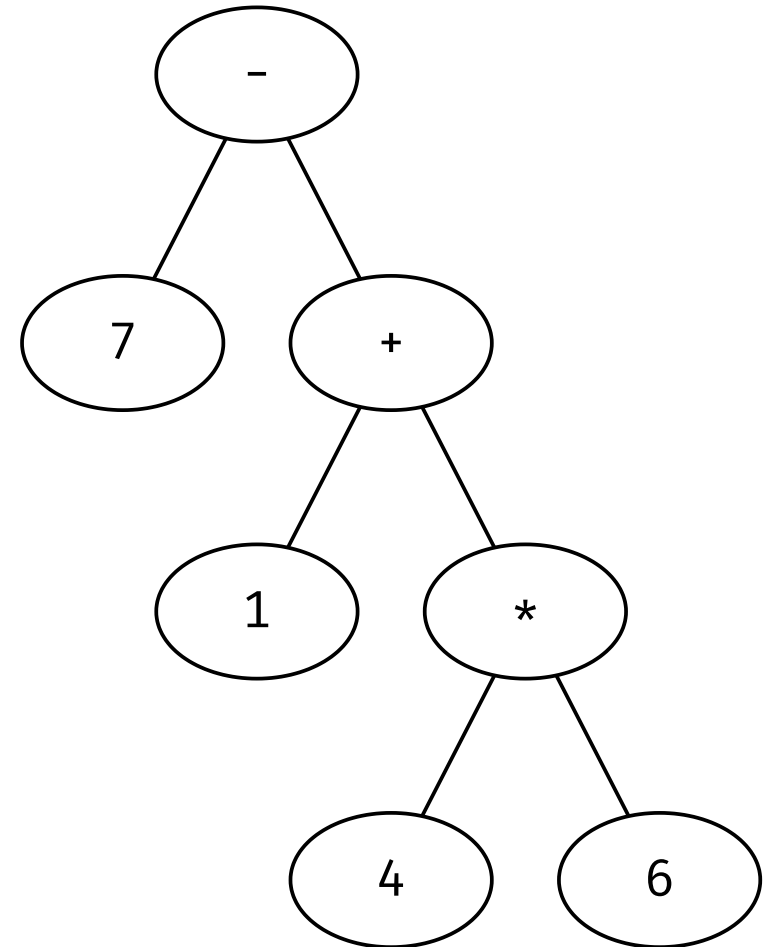
Using the stack: calculator

```
1 myops = LLStack()  
2 mydigs = LLStack()  
3  
4 myops.push(Op.SUB.value)  
5 mydigs.push(7)  
6 myops.push(Op.ADD.value)  
7 mydigs.push(1)  
8 myops.push(Op.MUL.value)  
9 mydigs.push(6)  
10 mydigs.push(4)  
11  
12 tmp = calc(myops, mydigs)  
13 assert tmp == 18
```



Using the stack: calculator

1. Pop operator **Op.MUL**
2. Pop 4 and 6
3. Push 24 ($4 * 6$)
4. Pop operator **Op.ADD**
5. Pop 24 and 1
6. Push 25 ($24 + 1$)
7. Pop operator **Op.SUB**
8. pop 25 and 7
9. push 18 ($25 - 7$)
10. pop 18 (Result, no more operators)



Queues

Queue

- » A queue is an ADT where values are inserted and removed from each end
 - » Similar to a queue/waiting line in the real world
- » FIFO (First In, First Out)
- » Two main operations are *enqueue* and *dequeue* (insert and remove)
- » Many uses in computer science
 - » E.g., to multiple requests, network traffic, processes in an operating system

Queue ADT: operations

» Operations

» `__init__`

» `enqueue(v:int) -> None`

» `dequeue() -> int|None`

» `empty() -> bool`

» `count() -> int`

Queue ADT: semantics

```
1 q = SimpleQueue()  
2 assert q.empty() == True  
3  
4 q.enqueue(1)  
5 assert q.empty() == False  
6 assert q.count() == 1  
7  
8 assert q.dequeue() == 1  
9 assert q.empty()
```

This is not executing yet, since we have not implemented `SimpleQueue`

Implementing a queue with an array

```
1 class SimpleQueue():
2     def __init__(self) -> None:
3         self.lst = np.empty(10, dtype=np.int64)
4         self.sz = 10
5         self.front = 0
6         self.back = 0
```

Implementing a queue with an array

```
1 @patch
2 def enqueue(self:SimpleQueue, v:int) -> None:
3     if self.back < self.sz:
4         self.lst[self.back] = v
5         self.back += 1
```

Implementing a queue with an array

```
1 @patch
2 def dequeue(self:SimpleQueue) -> int|None:
3     if self.front < self.sz and self.front < self.back:
4         tmp = self.lst[self.front]
5         self.front += 1
6         return tmp
7     return None
```


Implementing a queue with an array

```
1 @patch
2 def count(self:SimpleQueue) -> int:
3     return self.back - self.front
4
5 @patch
6 def empty(self:SimpleQueue) -> bool:
7     return self.back == self.front
```

Does it work?

```
1 q = SimpleQueue()
2 assert q.empty() == True
3
4 q.enqueue(1)
5 assert q.empty() == False
6 assert q.count() == 1
7
8 assert q.dequeue() == 1
9 assert q.empty()
```

Does it work?

```
1 q = SimpleQueue()  
2 q.enqueue(0)  
3 for i in range(1, 15):  
4     q.enqueue(i)  
5     q.dequeue()
```

It will work, but the queue will be full for the last few enqueues and empty for the last few dequeues.

Our implementations ignores when the queue becomes full and returns None if it is empty.

We know how to fix that!

- » If the queue becomes full
 - » Increase the size (to the double)
 - » Copy the old queue to the new queue
- » Good idea?

We do not know how to fix that!

- » Will work, but can be a poor solution
- » What happens if enqueue and dequeue are interleaved?

How long is the queue?

```
1 q = SimpleQueue()  
2 q.enqueue(0)  
3 for i in range(1, 15):  
4     q.enqueue(i)  
5     q.dequeue()
```

The queue is at most two elements long. We enqueue and dequeue once in each iteration.

We do not know how to fix that!

- » If we increase the size of the backing array in the example
 - » We end up wasting a lot of space
- » Each element we enqueue end up on the index of its value
 - » So, 14 ends up on index 14
- » When done, the back of the queue is at index 14
 - » The queue can hold a maximum of six elements before we need to grow

Circular queues

- » The back and front will move as we enqueue and dequeue
- » A lot of space will potentially be wasted
- » So, let's reclaim it by wrapping around when we hit the end
 - » If the positions are free...

Example

- » Assume we have a queue of size 10
- » The back is at index 9 and the front is at index 7
- » We could grow, but wasteful
- » Instead, we insert at $(9 + 1) \% 10$
 - » This wraps around the end and we are back at index 0

Fixing our implementation

```
1 @patch
2 def enqueue(self:SimpleQueue, v:int) -> None:
3     if self.count() < self.sz:
4         self.lst[self.back] = v
5         self.back = (self.back + 1) % self.sz
```

Fixing our implementation

```
1 @patch
2 def count(self:SimpleQueue) -> int:
3     if self.back >= self.front:
4         return self.back - self.front
5     else:
6         return self.sz - (self.front - self.back)
```

Fixing our implementation

```
1 @patch
2 def dequeue(self:SimpleQueue) -> int|None:
3     if self.count() > 0:
4         tmp = self.lst[self.front]
5         self.front = (self.front + 1) % self.sz
6         return tmp
7     return None
```

Checking

```
1 q = SimpleQueue()  
2 q.enqueue(0)  
3  
4 gots = []  
5 for i in range(1,1001):  
6     q.enqueue(i)  
7     gots.append(q.dequeue())  
8  
9 assert gots == list(range(1000))
```

Solved?

- » No, our has a finite size
 - » and can fill up
- » To fix it, **enqueue** should extend when the queue is full
 - » and copy the data

A linked queue

```
1 class LinkedQueue():
2     def __init__(self) -> None:
3         self.front = None
4         self.back = None
5         self.cnt = 0
6
7     def count(self) -> int:
8         return self.cnt
```

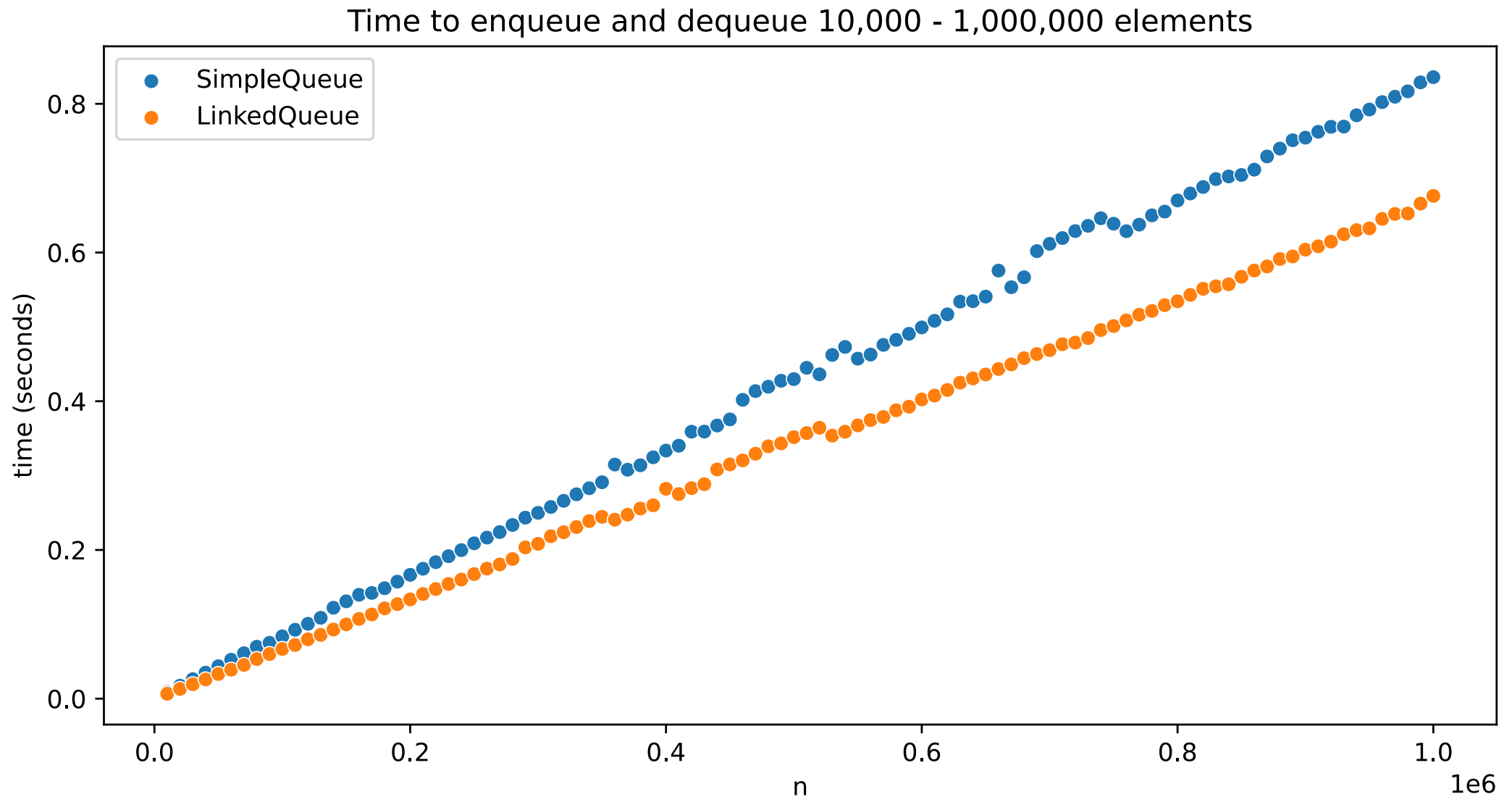
A linked queue

```
1 @patch
2 def enqueue(self:LinkedQueue, v:int) -> None:
3     if self.back is None:
4         self.back = LLNode(v)
5         self.front = self.back
6         self.cnt += 1
7     else:
8         self.back.nxt = LLNode(v)
9         self.back = self.back.nxt
10        self.cnt += 1
```

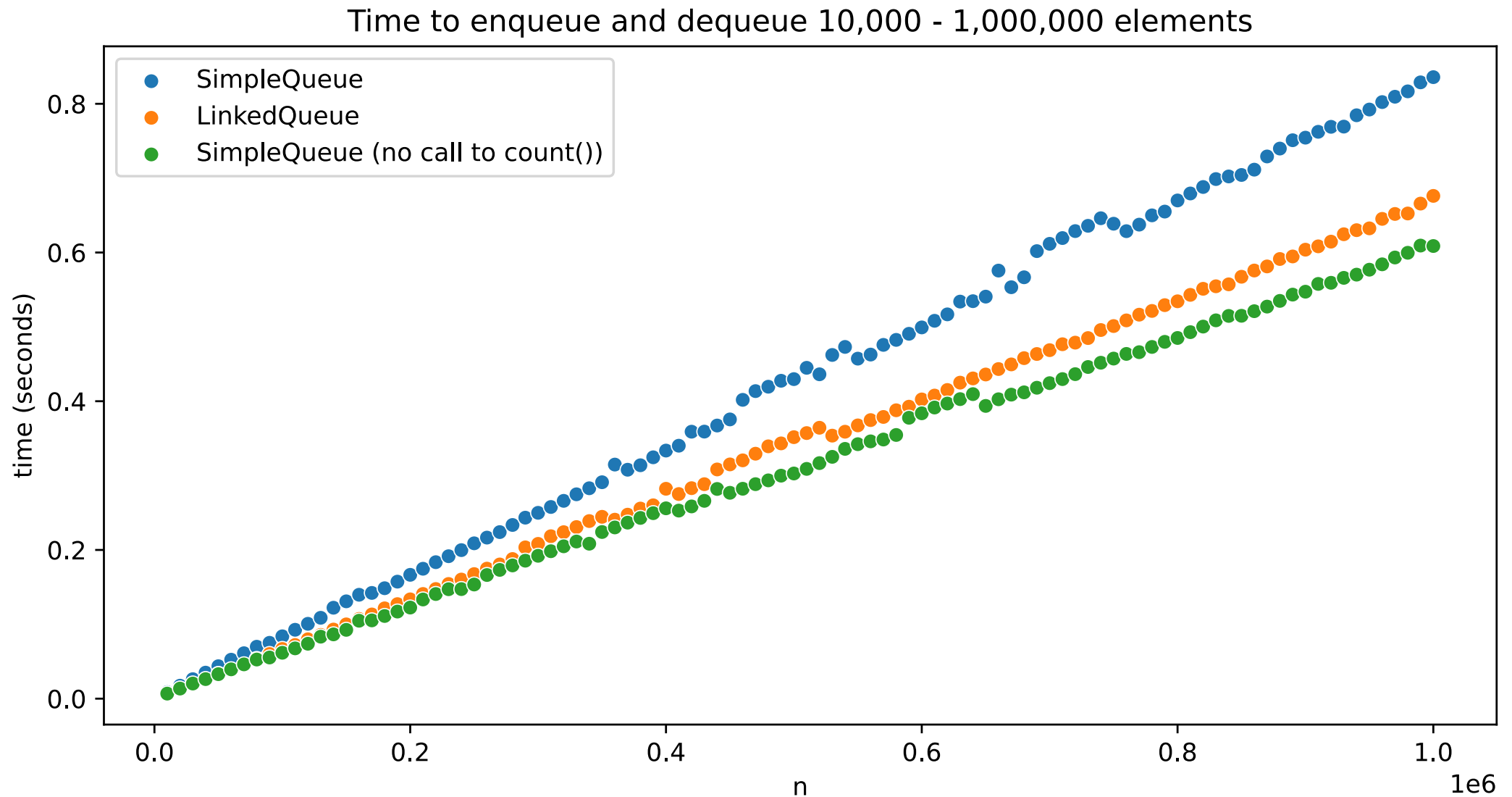

A linked queue

```
1 @patch
2 def dequeue(self:LinkedQueue) -> int|None:
3     if self.front is not None:
4         tmp = self.front.val
5         self.front = self.front.nxt
6         self.cnt -= 1
7         if self.front is None:
8             self.back = None
9         return tmp
10    return None
```

How do they compare?



What?



Set / Bag

Set

- » A set is an unordered collection of unique elements
- » The main operation is set membership
 - » as well as way to combine sets, e.g., union
- » Mutable and immutable (frozen)

Set ADT: operations

» Operations:

» `__init__`

» `add(v:int) -> None`

» `delete(v:int) -> None`

» `contains(v:int) -> bool`

Set ADT: semantics

```
1 s = LLSet()
2 assert s.contains(1) == False
3
4 s.add(1)
5 assert s.contains(1) == True
6
7 s.delete(1)
8 assert s.contains(1) == False
9
10 s.add(1)
11 s.add(1)
12 s.add(1)
13 s.delete(1)
14 assert s.contains(1) == False
```

Implementation

```
1 class LLSet:
2     def __init__(self) -> None:
3         self.lst = None
```


Implementation

```
1 @patch
2 def add(self:LLSet, v:int) -> None:
3     if self.lst is None:
4         self.lst = LLNode(v)
5     else:
6         p, pp = self.lst, None
7         found = False
8         while p is not None:
9             if p.val == v:
10                 found = True
11                 break
12             pp = p
13             p = p.nxt
14         if not found:
15             pp.nxt = LLNode(v)
```

Implementation

```
1 @patch
2 def delete(self:LLSet, v:int) -> None:
3     if self.lst is not None:
4         if self.lst.val == v:
5             self.lst = self.lst.nxt
6         else:
7             p, pp = self.lst.nxt, self.lst
8             found = False
9             while p is not None:
10                 if p.val == v:
11                     found = True
12                     break
13                 pp = p
14                 p = p.nxt
15             if found:
16                 pp.nxt = p.nxt if p is not None else None
```

Implementation

```
1 @patch
2 def contains(self:LLSet, v:int) -> bool:
3     p = self.lst
4     while p is not None:
5         if p.val == v:
6             return True
7         p = p.nxt
8     return False
```

Does it work?

```
1 s = LLSet()
2 assert s.contains(1) == False
3
4 s.add(1)
5 assert s.contains(1) == True
6
7 s.delete(1)
8 assert s.contains(1) == False
9
10 s.add(1)
11 s.add(1)
12 s.add(1)
13 s.delete(1)
14 assert s.contains(1) == False
```

Some python



```
1 @patch
2 def __contains__(self:LLSet, v:int) -> bool:
3     p = self.lst
4     while p is not None:
5         if p.val == v:
6             return True
7         p = p.nxt
8     return False
```

Does it work

```
1 s = LLSet()
2 assert 1 not in s
3
4 s.add(1)
5 assert 1 in s
6
7 s.delete(1)
8 assert 1 not in s
9
10 s.add(1)
11 s.add(1)
12 s.add(1)
13 s.delete(1)
14 assert 1 not in s
```

We can add an iterator

```
1 @patch
2 def __iter__(self:LLSet) -> LLIterator:
3     return LLIterator(self.lst)
```

We can reuse `LLIterator` since we use the same structure as `LinkedList`

To get all the values

```
1 s = LLSet()  
2  
3 s.add(1)  
4 s.add(2)  
5 s.add(3)  
6  
7 for v in s:  
8     print(v)
```

1
2
3

Additional operations

» Union (\cup)

» Merges to sets, $\{1, 2\} \cup \{2, 3\} = \{1, 2, 3\}$

» Intersection (\cap)

» Common elements, $\{1, 2\} \cap \{2, 3\} = \{2\}$

» Difference ($-$)

» Subtraction, $\{1, 2\} - \{2, 3\} = \{1\}$

Adding the operations

```
1 @patch
2 def union(self:LLSet, other:LLSet) -> None:
3     for v in other:
4         self.add(v)
```

Note that union mutates the set. We could also return a new set that is the union of the two.

Adding the operations

```
1 @patch
2 def intersection(self:LLSet, other:LLSet) -> LLSet:
3     tmp = LLSet()
4     for v in self:
5         if v in other:
6             tmp.add(v)
7     return tmp
```

This is a non-mutating intersection.

Adding the operations

```
1 @patch
2 def difference(self:LLSet, other:LLSet) -> None:
3     for v in other:
4         self.delete(v)
```

Checking semantics

```
1  s1 = LLSet()  
2  s1.add(1)  
3  s1.add(2)  
4  
5  s2 = LLSet()  
6  s2.add(2)  
7  s2.add(3)  
8  
9  s1.union(s2)  
10 assert 3 in s1
```

Checking semantics

```
1  s1 = LLSet()  
2  s1.add(1)  
3  s1.add(2)  
4  
5  s2 = LLSet()  
6  s2.add(2)  
7  s2.add(3)  
8  
9  s3 = s1.intersection(s2)  
10 assert 1 not in s3  
11 assert 2 in s3
```

Checking semantics

```
1  s1 = LLSet()  
2  s1.add(1)  
3  s1.add(2)  
4  
5  s2 = LLSet()  
6  s2.add(2)  
7  s2.add(3)  
8  
9  s1.difference(s2)  
10 assert 1 in s1  
11 assert 2 not in s1
```

Bags

- » Checking for duplicates is expensive
- » So, a bag is a set that allows duplicate values
- » Makes insert easier
 - » but some of the other operations a bit more complicated
 - » e.g., delete must now delete all instances of a value
- » Can save time if many adds
 - » at the cost of space

In Python

Lists, Stacks & Queues in Python

- » Python (of course) provides implementations of all these
- » We have used list several times
 - » and it can also act as a stack (or queue)

Calc using Python lists

```
1  from typing import List
2
3  def calc(ops:List[int], dig:List[int]) -> int:
4      while ops:
5          o = Op(ops.pop())
6          d1 = dig.pop()
7          d2 = dig.pop()
8          dig.append(doop(o, d1, d2))
9      tmp = dig.pop()
10     return tmp
```

Queues

- » Can be implemented using normal lists
- » A better (more efficient) alternative is to use `collections.deque`

Set

- » Sets can be defined via `set()` or `{...}`
 - » `set([1,2,3]) == {1,2,3}`
- » Note that you use `add` for sets, not `append`
- » Note that `{1,2,3}` is a set, while `{1:2,2:3}` is a dictionary.
 - » and `{}` is a dictionary.
 - » use `set()` for an empty set

Reading instructions

Reading instructions

» Ch. 3