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 finitie, 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

$$
\begin{array}{llll}
newstack & : & & \rightarrow \quad STACK \\
push & : & STACK \times INT & \rightarrow \quad STACK \\
pop & : & STACK & \rightarrow \quad STACK \\
top & : & STACK & \rightarrow \quad INT \cup \{Error\} \\
empty & : & STACK & \rightarrow \quad BOOLEAN
\end{array}
$$

Example: Algebraic specification

Semantics for a Stack that can hold integers

$$\begin{aligned}
empty(newstack) &= True \\
empty(push(s, i)) &= False \\
pop(newstack) &= new \\
pop(push(s, i)) &= s \\
top(new) &= Error \\
top(push(s, i)) &= i
\end{aligned}$$

where $s \in STACK$ and $i \in INT$.

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.

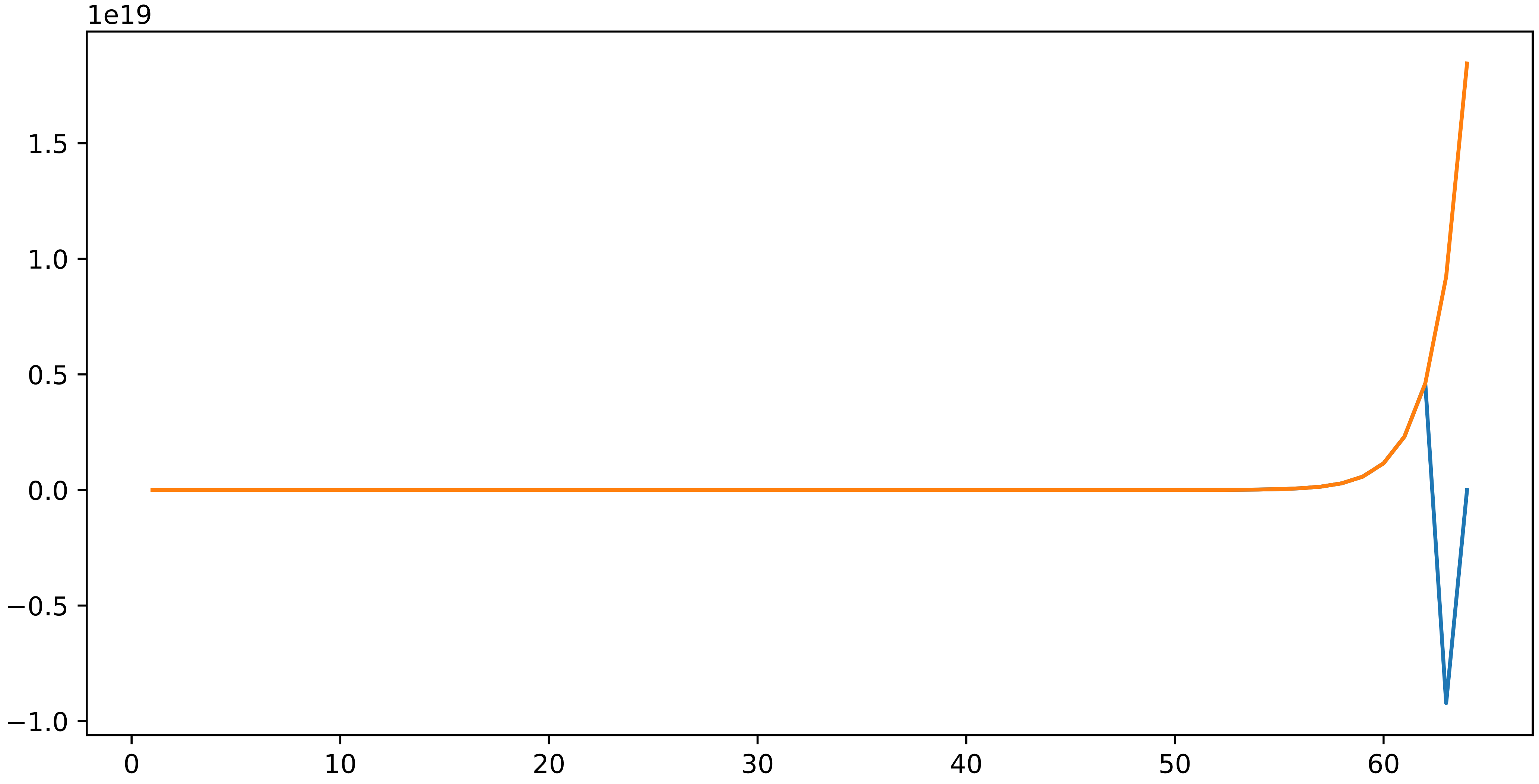
```python
import numpy as np

m1 = np.arange(10)
m2 = m1 + 2
m3 = m2 ** 2
print(m3)
```

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

Restrictions

```python
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

x1 = np.arange(1, 65)
x2 = list(range(1, 65))

sns.lineplot(x1, 2**x1)
sns.lineplot(x2, [x**2 for x in x2])
plt.show()
```

Restrictions



Restrictions

```python
x = np.arange(62, 65)
print(2**x)
print(2**63)
print(type(x[0]))
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 $A_0, \ldots, A_{N-1}$ 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() -> bool`

» `count() -> int`

» `append(i:int) -> None`

A sketch of the semantics

```python
sl = SimpleList()
assert sl.empty()

sl.append(1)
sl.append(2)
sl.append(3)

assert not sl.empty()
assert sl.count() == 3
```

A simple list

```python
class SimpleList():
  def __init__(self) -> None:
    self.lst = np.empty(10, dtype=np.int64)
    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

```python
from fastcore.basics import patch

@patch
def empty(self:SimpleList) -> bool:
  return self.cnt == 0

@patch
def count(self:SimpleList) -> int:
  return self.cnt
```

A simple list

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

Checking semantics

```python
sl = SimpleList()
assert sl.empty()

sl.append(1)
sl.append(2)
sl.append(3)

assert not sl.empty()
assert sl.count() == 3

print(sl.count())
```

```
3
```

Extending the ADT

```python
@patch
def delete(self:SimpleList, d:int) -> None:
  found = False
  for i in range(0, self.cnt):
    if self.lst[i] == d:
      found = True
      break
  if found:
    for j in range(i+1, self.cnt):
      self.lst[j-1] = self.lst[j]
    self.cnt -= 1
```

Checking semantics

```python
sl = SimpleList()

assert sl.empty() == True

sl.append(1)
sl.append(2)

assert sl.count() == 2

sl.delete(1)

assert sl.count() == 1

sl.delete(2)

assert sl.empty() == True
```

Alternative idea: free list

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

Alternative idea

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

Alternative idea

```python
@patch
def append(self:SimpleListFL, d:int) -> None:
  for ix, v in enumerate(self.fl):
    if v == True:
      self.lst[ix] = d
      self.fl[ix] = False
      break
```

Alternative idea

```python
@patch
def count(self:SimpleListFL) -> None:
  s = 0
  for v in self.fl:
    if v == False:
      s += 1
  return s

@patch
def empty(self:SimpleListFL) -> None:
  for v in self.fl:
    if v == False:
      return False
  return True
```

Checking semantics

```python
sl = SimpleListFL()

assert sl.empty() == True

sl.append(1)
sl.append(2)

assert sl.count() == 2

sl.delete(1)

assert sl.count() == 1

sl.delete(2)

assert sl.empty() == True
```

Making it more Python

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

Making it more Python

```python
sl = SimpleList()

assert sl.empty() == True

sl.append(1)
sl.append(2)
sl.append(3)

assert len(sl) == 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?

```python
sl = SimpleList()

for i in range(15):
    sl.append(i)
```

IndexError

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

A new append

```python
@patch
def append(self:SimpleList, d:int) -> None:
  if self.cnt == self.sz:
    self.sz += 1
    tmp = np.empty(self.sz, dtype=np.int64)
    for i in range(self.cnt):
      tmp[i] = self.lst[i]
    self.lst = tmp

  self.lst[self.cnt] = d
  self.cnt += 1
```

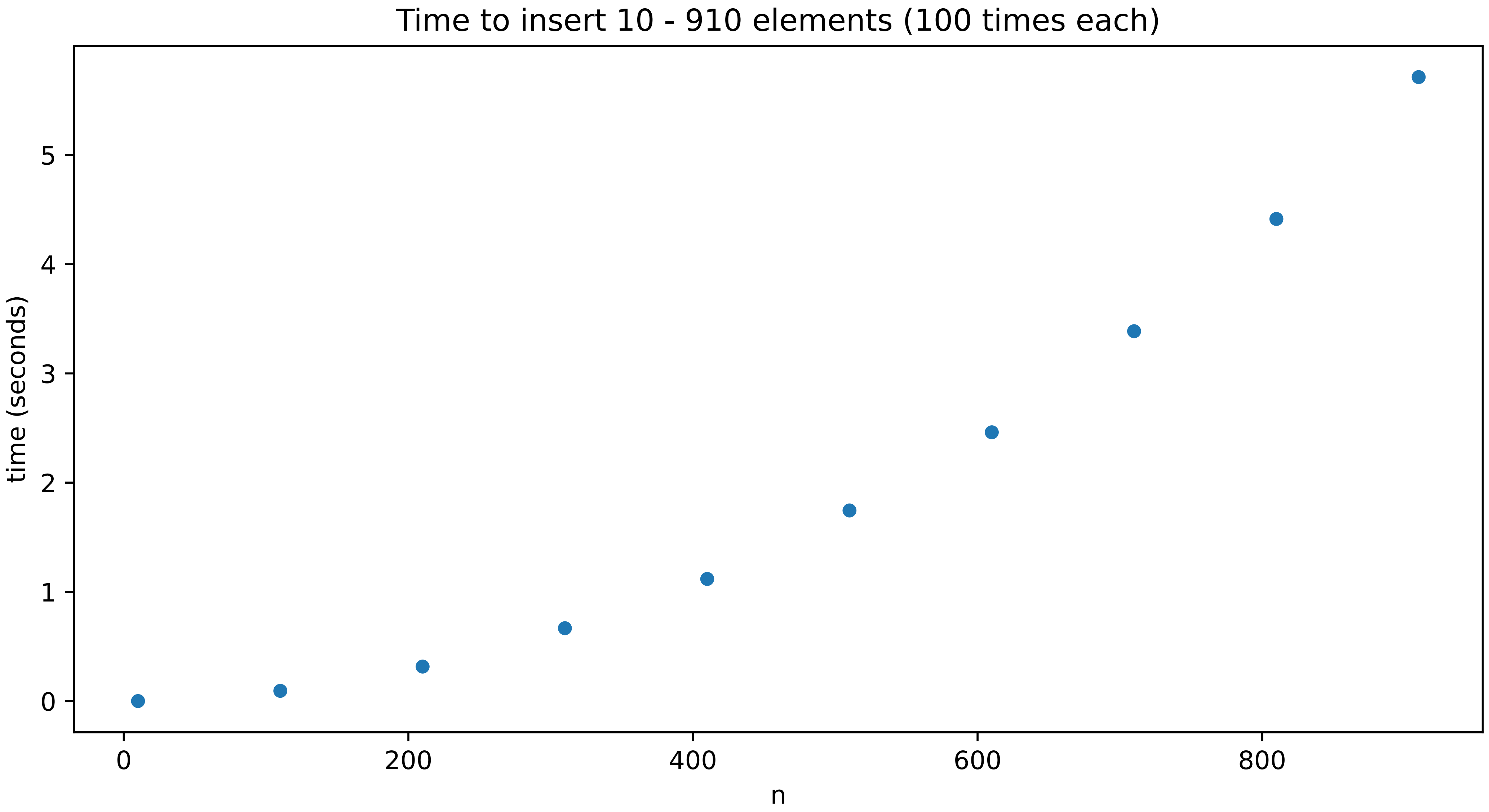
We need to fix init

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

Again

```python
sl = SimpleList()

for i in range(15):
  sl.append(i)

assert sl.count() == 15
```

Problems?

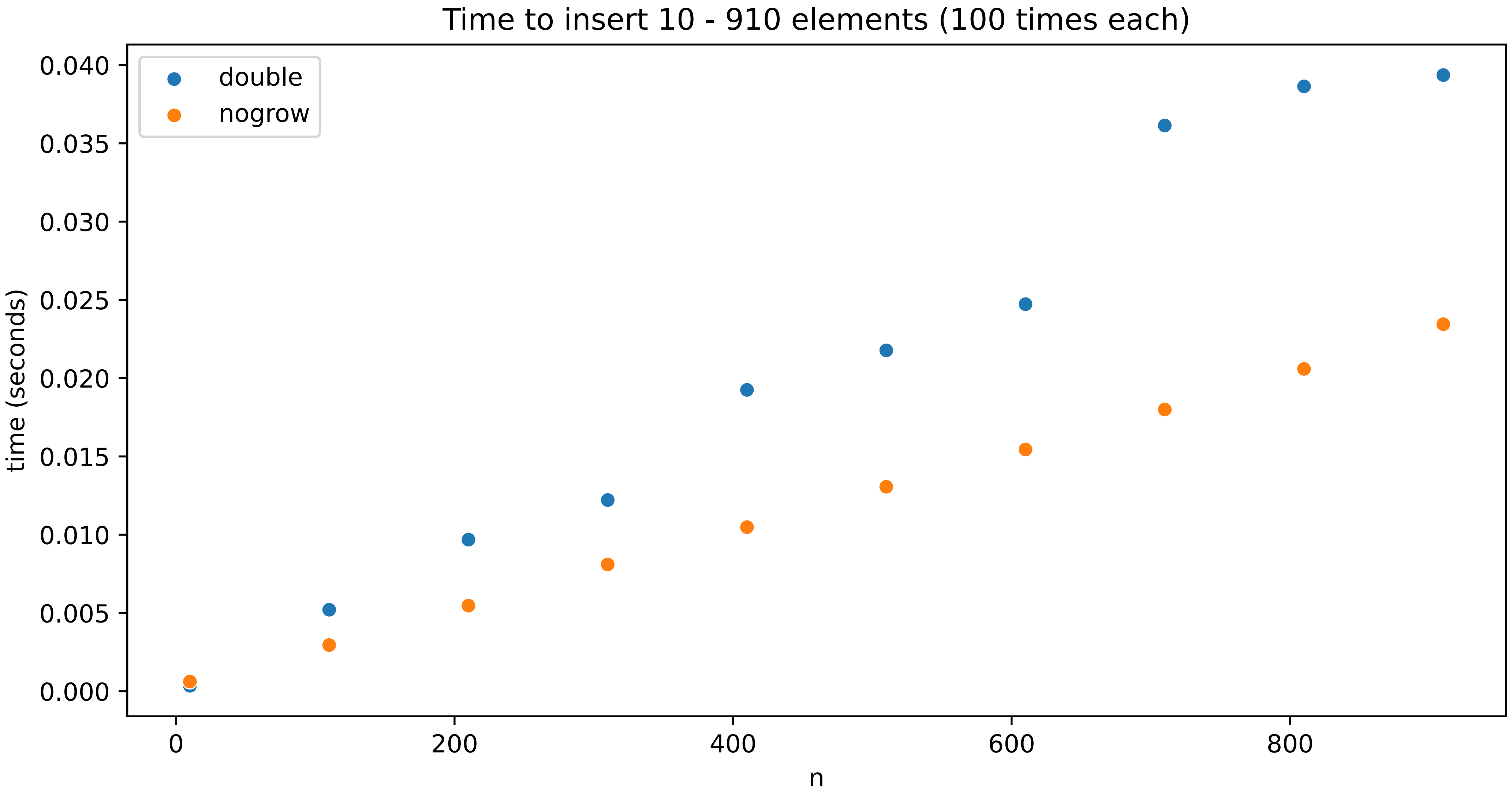


Time to insert 10 - 910 elements (100 times each)
time (seconds)
0
1
2
3
4
5
0
200
400
600
800
n

A better approach

```python
@patch
def append(self:SimpleList, d:int) -> None:
  if self.cnt == self.sz:
    self.sz *= 2
    tmp = np.empty(self.sz, dtype=np.int64)
    for i in range(self.cnt):
      tmp[i] = self.lst[i]
    self.lst = tmp

  self.lst[self.cnt] = d
  self.cnt += 1
```

Problems?


Time to insert 10 - 910 elements (100 times each)
double
nogrow
time (seconds)
n
0.000
0.005
0.010
0.015
0.020
0.025
0.030
0.035
0.040
0
200
400
600
800

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

```python
class LinkedList():
  def __init__(self) -> None:
    self.lst = None
```

Another list

```python
@patch
def append(self:LinkedList, d:int) -> None:
  if self.lst is None:
    self.lst = LLNode(d)
  else:
    p = self.lst
    while p.nxt is not None:
      p = p.nxt
    p.nxt = LLNode(d)
```

LLNode

```python
from dataclasses import dataclass

@dataclass
class LLNode:
  val: int
  nxt: 'LLNode|None' = None
```

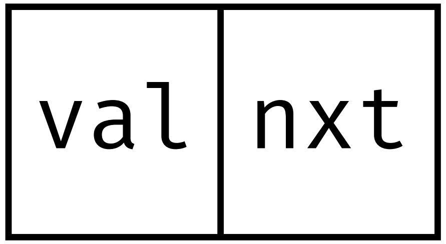
Does it work?

```python
ll = LinkedList()
ll.append(1)
ll.append(2)

print(ll.lst)
```

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

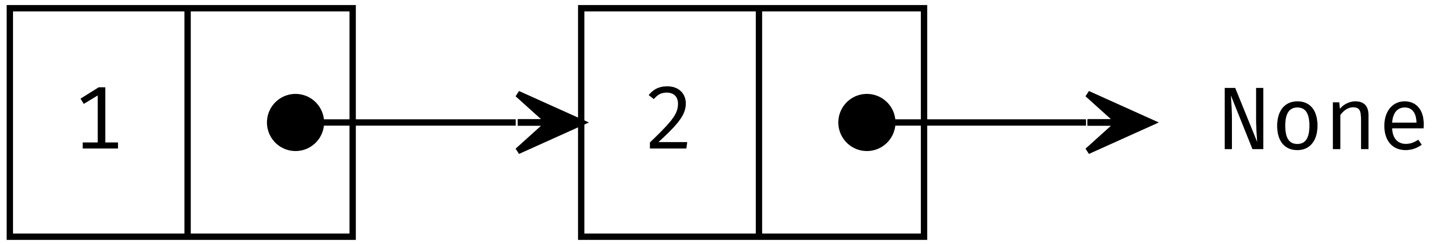
How does it work?

LLNode



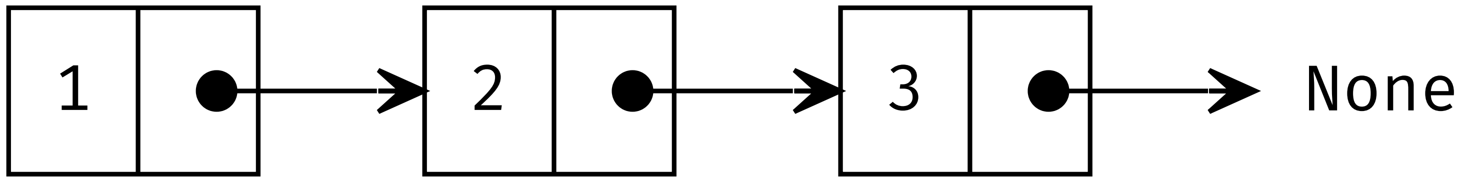
How does it work?

The LinkedList after appending the values 1 and 2.



How does it work?

The LinkedList after appending 3



Nicer print

```python
@patch
def __str__(self:LinkedList) -> str:
  s = '['
  p = self.lst
  while p is not None:
    s += f' {p.val},'
    p = p.nxt
  if s == '[':
    return '[ ]'
  else:
    return s[:-1] +' ]'
```

Does it work?

```python
ll = LinkedList()
ll.append(1)
ll.append(2)
ll.append(3)

print(ll)
```

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

Adding the other operations

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

Adding the other operations

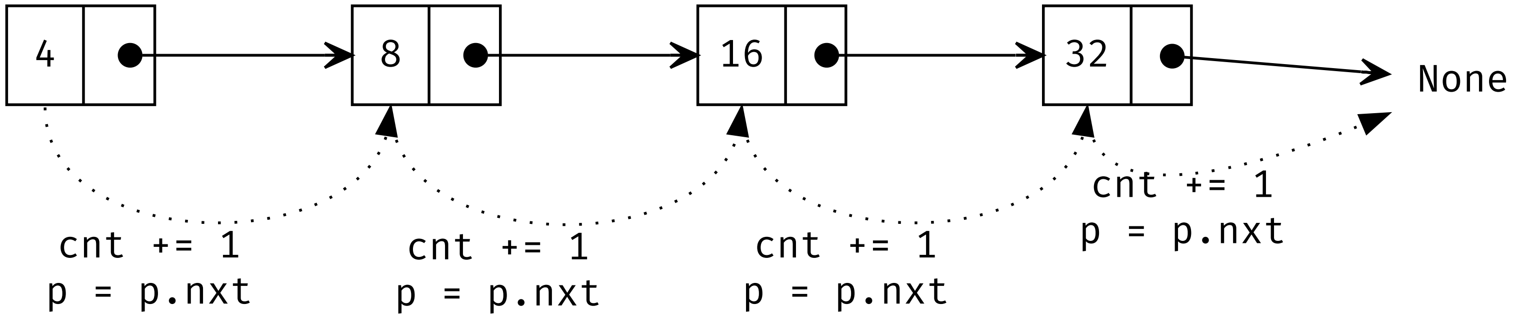
```python
@patch
def __len__(self:LinkedList) -> int:
  cnt, p = 0, self.lst
  while p is not None:
    cnt += 1
    p = p.nxt

  return cnt
```

Does it work?

```python
ll = LinkedList()

assert ll.empty() == True

ll.append(1)
ll.append(2)
ll.append(3)

assert len(ll) == 3
```

How does it work?

Count of a list



Adding the other operations

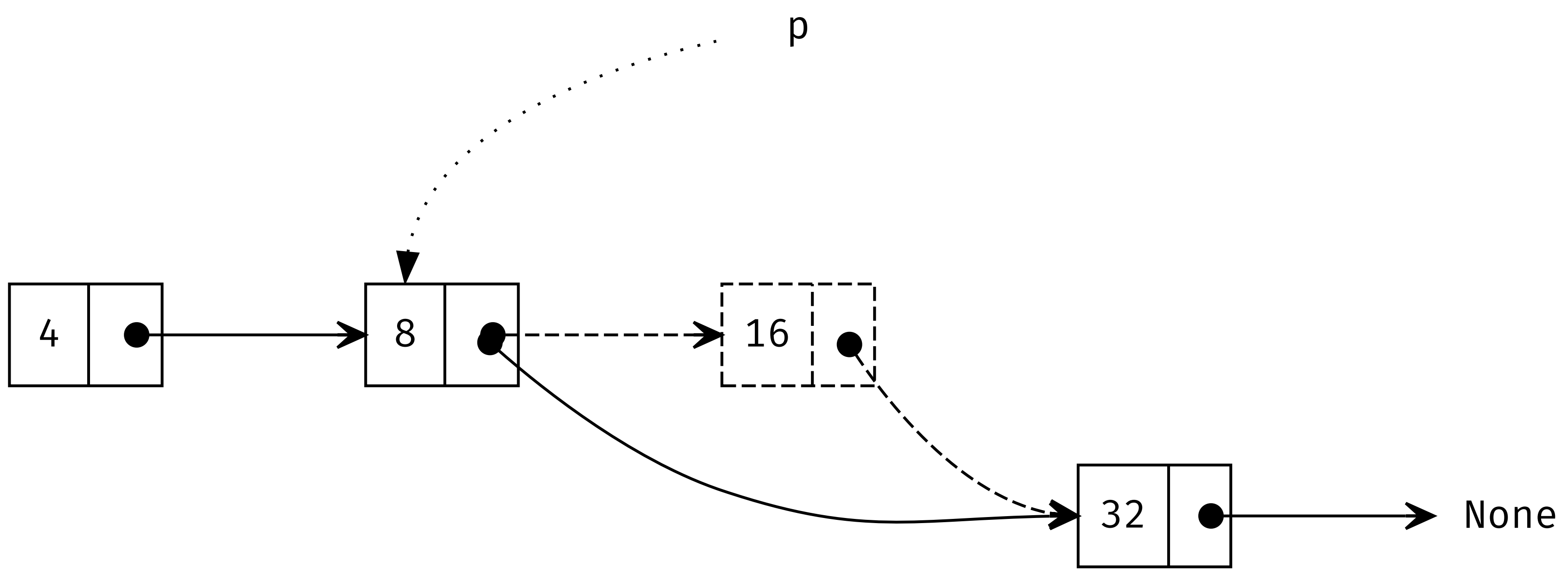
```python
@patch
def delete(self:LinkedList, d:int) -> None:
  if self.lst is not None:
    if self.lst.val == d:
      if self.lst.nxt is None:
        self.lst = None
      else:
        self.lst = self.lst.nxt
    else:
      p = self.lst
      while p.nxt is not None and p.nxt.val != d:
        p = p.nxt
      if p.nxt is not None:
        p.nxt = p.nxt.nxt
```

Checking delete

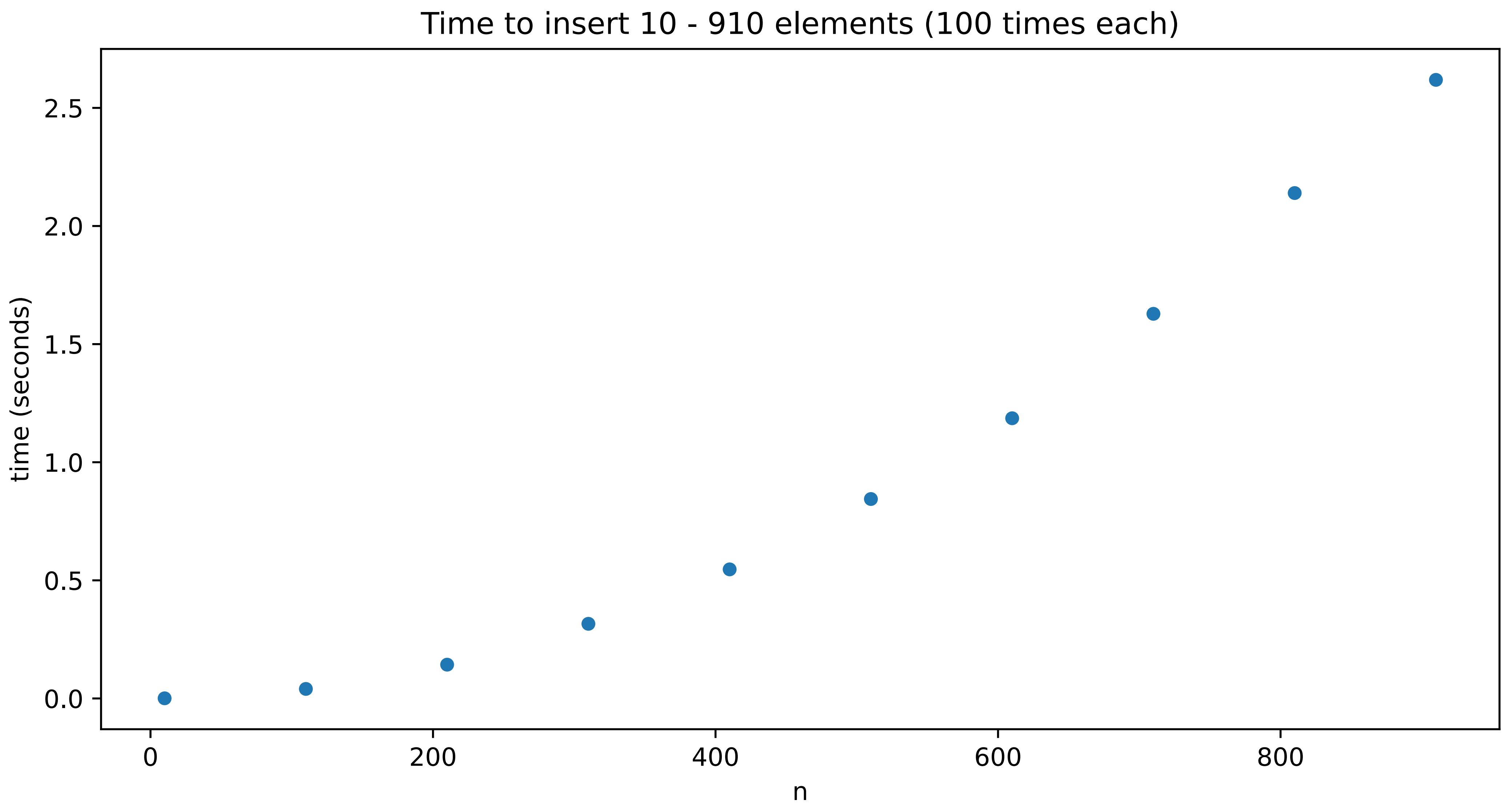
```python
ll = LinkedList()

assert ll.empty() == True

ll.append(1)
ll.append(2)

assert len(ll) == 2

ll.delete(1)

assert len(ll) == 1

ll.delete(2)

assert ll.empty() == True
```

How does it work?

Deleting node with value 16



Performance?



Fix?

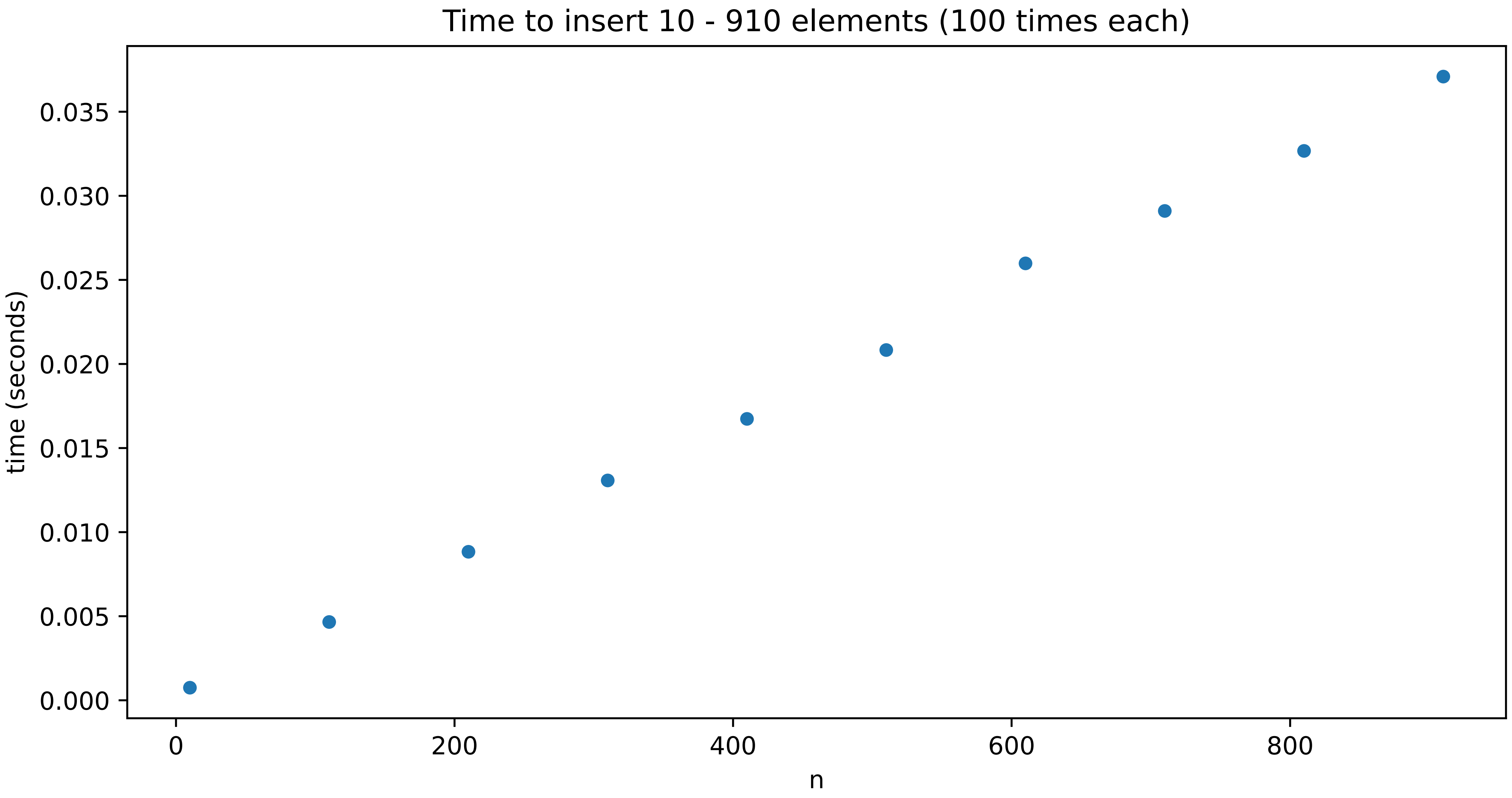
```python
@patch
def append(self:LinkedList, d:int) -> None:
  if self.lst is None:
    self.lst = LLNode(d)
    self.last = self.lst
  else:
    self.last.nxt = LLNode(d)
    self.last = self.last.nxt
```

Updating the constructor

```python
def __init__(self:LinkedList) -> None:
  self.lst = None
  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?


Time to insert 10 - 910 elements (100 times each)
time (seconds)
0.000
0.005
0.010
0.015
0.020
0.025
0.030
0.035
n
0
200
400
600
800

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)

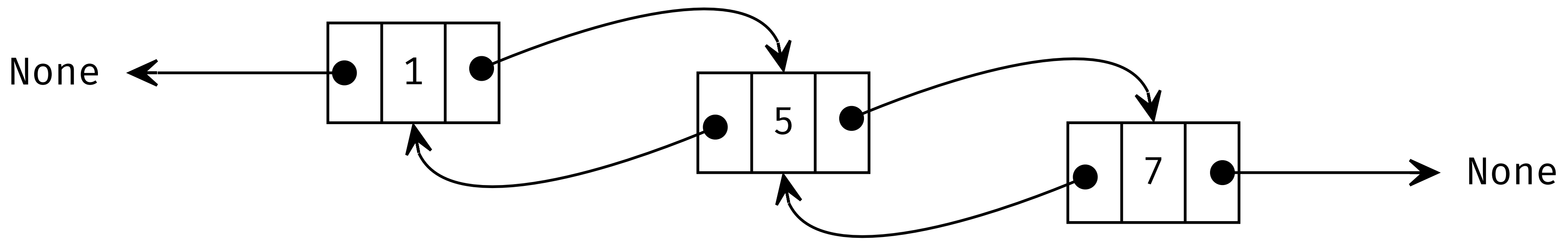
DLLNode

```python
@dataclass
class DLLNode:
  val: int
  prv: 'LLNode|None' = None
  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 travers 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

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

Iterator for LinkedList

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

Iterator for LinkedList

```python
@patch
def __iter__(self:LLIterator) -> LLIterator:
    return self
```

Iterator for LinkedList

```python
@patch
def __next__(self:LLIterator) -> int:
  if self.current is None:
    raise StopIteration
  else:
    tmp = self.current.val
    self.current = self.current.nxt
    return tmp
```

Testing

```python
ll = LinkedList()
for i in range(10):
  ll.append(i)

s = 0
for v in ll:
  s += v

assert s == 45
```

Enables some cool stuff

```python
ll = LinkedList()
for i in range(10):
  ll.append(i**2)

s = sum(ll)

assert s == 285
```

How does it work?

```python
ll = LinkedList()
for i in range(10):
  ll.append(i**3)

s = 0
it = iter(ll)
while True:
  try:
    s += next(it)
  except StopIteration:
    break

assert s == 2025
```

Iterators

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

```python
@patch
def __next__(self:SLIterator) -> int:
  if self.pos == self.cnt:
    raise StopIteration
  else:
    tmp = self.lst[pos]
    self.pos += 1
    return tmp
```

Accessing a specific position

```python
@patch
def __getitem__(self:SimpleList, key:int) -> int:
  if key < self.cnt:
    return self.lst[key]
  else:
    raise IndexError
```

Checking

```python
sl = SimpleList()
for i in range(10):
  sl.append(i**2)

assert sl[5] == 25
```

Trickier in LinkedList

```python
@patch
def __getitem__(self:LinkedList, key:int) -> int:
  cpos = 0
  p = self.lst
  while p is not None:
    if cpos == key:
      return p.val
    cpos += 1
    p = p.nxt
  raise IndexError
```

Checking

```
ll = LinkedList()
for i in range(10):
  ll.append(i**2)

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

```python
class LLStack:
  def __init__(self) -> None:
    self.stack = None

  def empty(self) -> bool:
    return self.stack is None

  def top(self) -> int|None:
    if self.stack is None:
      return None
    else:
      return self.stack.val
```

Implementing a Stack with a linked list

```python
@patch
def push(self:LLStack, v:int) -> None:
  tmp = LLNode(v)
  tmp.next = self.stack
  self.stack = tmp
```

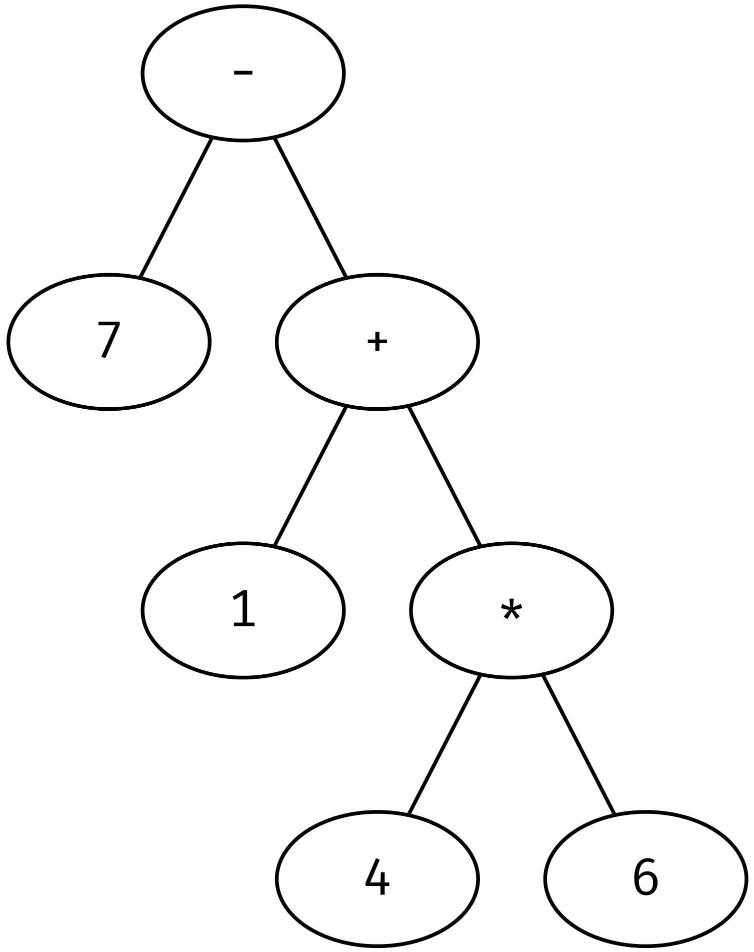
Implementing a Stack with a linked list

```python
@patch
def pop(self:LLStack) -> None:
  if self.stack is not None:
    self.stack = self.stack.next
```

Testing it

```python
stack = LLStack()
assert stack.empty() == True

stack.push(1)
assert stack.empty() == False
assert stack.top() == 1

stack.pop()
assert stack.empty() == True
```

Using the stack: calculator



» Suppose we want to compute $1 + 4 * 6 - 7$

» We can use a stack to represent the tree

Using the stack: calculator

```python
from enum import IntEnum

class Op(IntEnum):
    ADD = 1
    DIV = 2
    MUL = 3
    SUB = 4
```

Using the stack: calculator

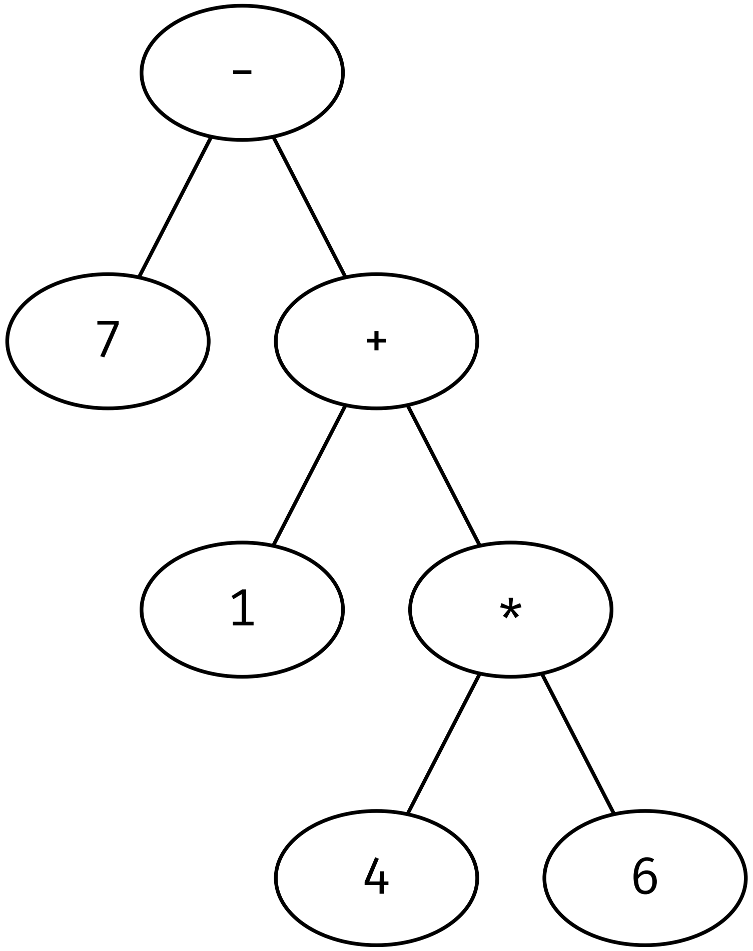
```
def calc(ops:LLStack, dig:LLStack) -> int:
    while not ops.empty():
        o = Op(ops.top())
        d1 = dig.top()
        dig.pop()
        d2 = dig.top()
        dig.pop()
        dig.push(doop(o, d1, d2))
        ops.pop()
    tmp = dig.top()
    dig.pop()
    return tmp
```

Using the stack: calculator

```python
def doop(op, d1, d2):
    match op:
        case Op.ADD:
            return d1 + d2
        case Op.DIV:
            return d1 / d2
        case Op.MUL:
            return d1 * d2
        case Op.SUB:
            return d1 - d2
```

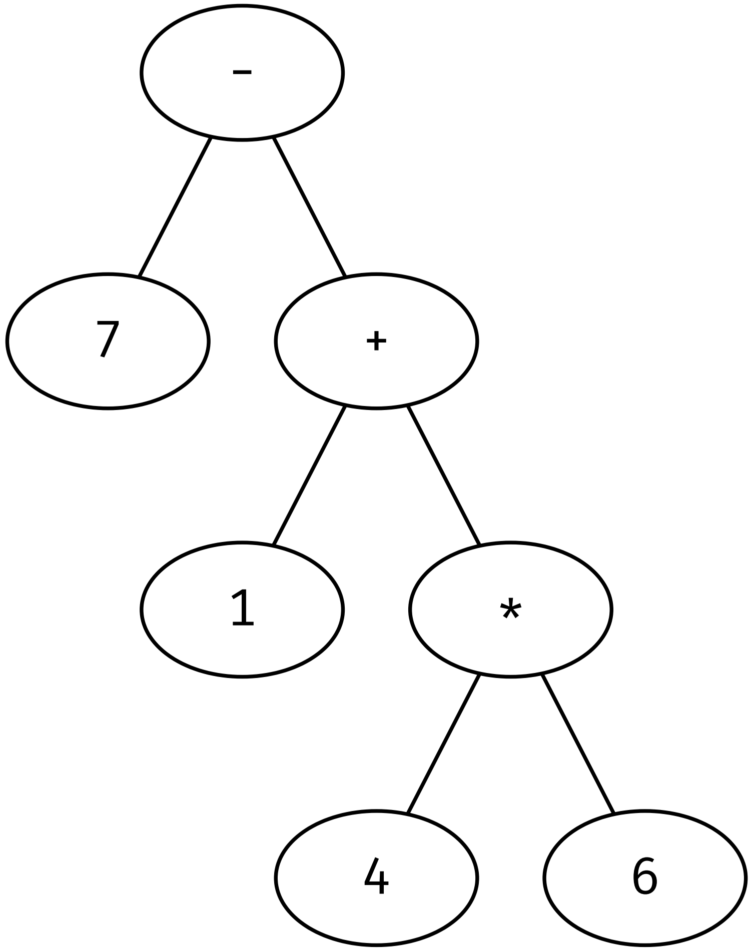
Using the stack: calculator

```python
myops = LLStack()
mydigs = LLStack()

myops.push(Op.SUB.value)
mydigs.push(7)
myops.push(Op.ADD.value)
mydigs.push(1)
myops.push(Op.MUL.value)
mydigs.push(6)
mydigs.push(4)

tmp = calc(myops, mydigs)
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

```python
q = SimpleQueue()
assert q.empty() == True

q.enqueue(1)
assert q.empty() == False
assert q.count() == 1

assert q.dequeue() == 1
assert q.empty()
```

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

Implementing a queue with an array

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

Implementing a queue with an array

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

Implementing a queue with an array

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

Implementing a queue with an array

```python
@patch
def count(self:SimpleQueue) -> int:
  return self.back - self.front

@patch
def empty(self:SimpleQueue) -> bool:
  return self.back == self.front
```

Does it work?

```python
q = SimpleQueue()
assert q.empty() == True

q.enqueue(1)
assert q.empty() == False
assert q.count() == 1

assert q.dequeue() == 1
assert q.empty()
```

Does it work?

```python
q = SimpleQueue()
q.enqueue(0)
for i in range(1,15):
    q.enqueue(i)
    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?

```python
q = SimpleQueue()
q.enqueue(0)
for i in range(1,15):
    q.enqueue(i)
    q.dequeue()
```

The queue is at most one element 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

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

Fixing our implementation

```python
@patch
def count(self:SimpleQueue) -> int:
  if self.back >= self.front:
    return self.back - self.front
  else:
    return self.sz - (self.front - self.back)
```

Fixing our implementation

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

Checking

```python
q = SimpleQueue()
q.enqueue(0)

gots = []
for i in range(1,1001):
    q.enqueue(i)
    gots.append(q.dequeue())

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

```python
class LinkedQueue():
  def __init__(self) -> None:
    self.front = None
    self.back = None
    self.cnt = 0

  def count(self) -> int:
    return self.cnt
```

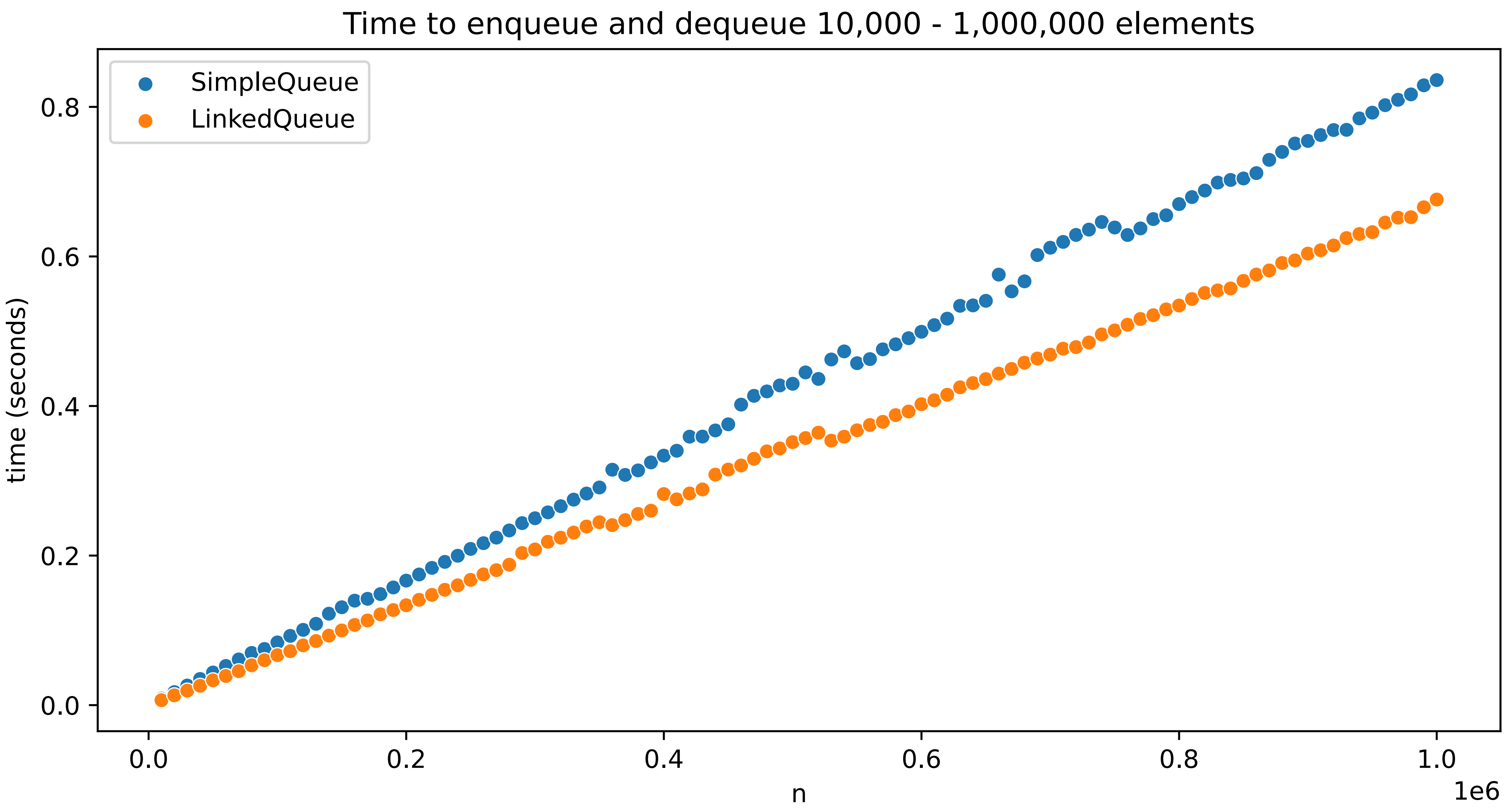
A linked queue

```python
@patch
def enqueue(self:LinkedQueue, v:int) -> None:
  if self.back is None:
    self.back = LLNode(v)
    self.front = self.back
    self.cnt += 1
  else:
    self.back.nxt = LLNode(v)
    self.back = self.back.nxt
    self.cnt += 1
```

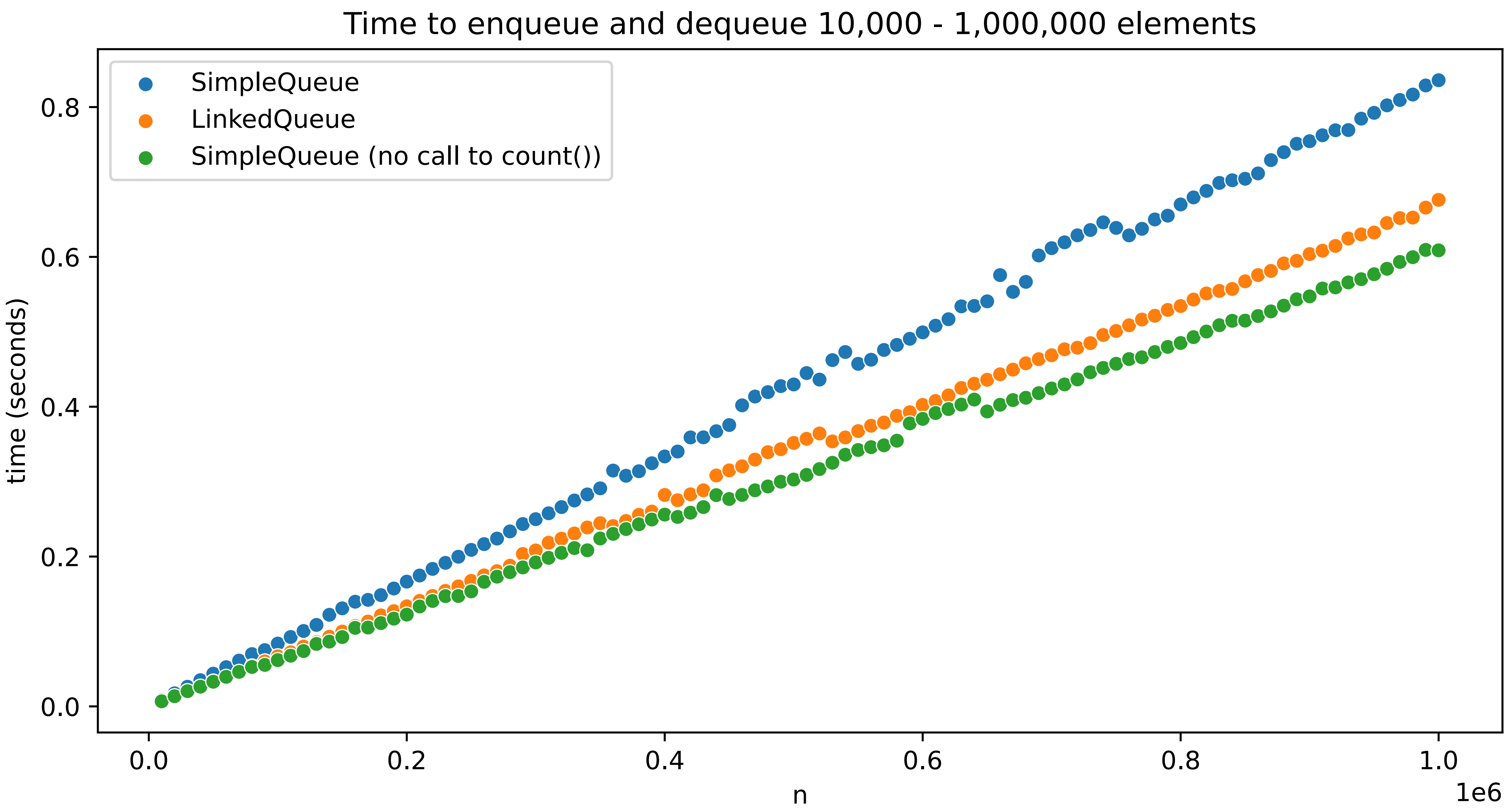
A linked queue

```python
@patch
def dequeue(self:LinkedQueue) -> int|None:
  if self.front is not None:
    tmp = self.front.val
    self.front = self.front.nxt
    self.cnt -= 1
    if self.front is None:
      self.back = None
    return tmp
  return None
```

How do they compare?



What?



Time to enqueue and dequeue 10,000 - 1,000,000 elements
● SimpleQueue
● LinkedQueue
● SimpleQueue (no call to count())
time (seconds)
0.0
0.2
0.4
0.6
0.8
n
0.0
0.2
0.4
0.6
0.8
1.0
1e6

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 unmutable (frozen)

Set ADT: operations

- Operations:
  - `__init__`
  - `add(v:int) -> None`
  - `delete(v:int) -> None`
  - `contains(v:int) -> None`

Set ADT: semantics

```python
s = LLSet()
assert s.contains(1) == False

s.add(1)
assert s.contains(1) == True

s.delete(1)
assert s.contains(1) == False

s.add(1)
s.add(1)
s.add(1)
s.delete(1)
assert s.contains(1) == False
```

Implementation

```python
class LLSet:
  def __init__(self) -> None:
    self.lst = None
```

Implementation

```python
@patch
def add(self:LLSet, v:int) -> None:
  if self.lst is None:
    self.lst = LLNode(v)
  else:
    p, pp = self.lst, None
    found = False
    while p is not None:
      if p.val == v:
        found = True
        break
      pp = p
      p = p.nxt
    if not found:
      pp.nxt = LLNode(v)
```

Implementation

```python
@patch
def delete(self:LLSet, v:int) -> None:
  if self.lst is not None:
    if self.lst.val == v:
      self.lst = self.lst.nxt
    else:
      p, pp = self.lst.nxt, self.lst
      found = False
      while p is not None:
        if p.val == v:
          found = True
          break
        pp = p
        p = p.nxt
      if found:
        pp.nxt = p.nxt if p is not None else None
```

Implementation

```python
@patch
def contains(self:LLSet, v:int) -> bool:
  if self is None:
    return False
  else:
    p = self.lst
    while p is not None:
      if p.val == v:
        return True
      p = p.nxt
    return False
```

Does it work?

```python
s = LLSet()
assert s.contains(1) == False

s.add(1)
assert s.contains(1) == True

s.delete(1)
assert s.contains(1) == False

s.add(1)
s.add(1)
s.add(1)
s.delete(1)
assert s.contains(1) == False
```

Some python 🦆

```python
@patch
def __contains__(self:LLSet, v:int) -> bool:
  if self is None:
    return False
  else:
    p = self.lst
    while p is not None:
      if p.val == v:
        return True
      p = p.nxt
    return False
```

Does it work

```
s = LLSet()
assert 1 not in s

s.add(1)
assert 1 in s

s.delete(1)
assert 1 not in s

s.add(1)
s.add(1)
s.add(1)
s.delete(1)
assert 1 not in s
```

We can add an iterator

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

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

To get all the values

```python
s = LLSet()

s.add(1)
s.add(2)
s.add(3)

for v in s:
  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

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

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

Adding the operations

```python
@patch
def intersection(self:LLSet, other:LLSet) -> LLSet:
  tmp = LLSet()
  for v in self:
    if v in other:
      tmp.add(v)
  return tmp
```

This is a non-mutating intersection.

Adding the operations

```python
@patch
def difference(self:LLSet, other:LLSet) -> LLSet:
  for v in other:
    self.delete(v)
```

Checking semantics

```
s1 = LLSet()
s1.add(1)
s1.add(2)

s2 = LLSet()
s2.add(2)
s2.add(3)

s1.union(s2)
assert 3 in s1
```

Checking semantics

```
s1 = LLSet()
s1.add(1)
s1.add(2)

s2 = LLSet()
s2.add(2)
s2.add(3)

s3 = s1.intersection(s2)
assert 1 not in s3
assert 2 in s3
```

Checking semantics

```python
s1 = LLSet()
s1.add(1)
s1.add(2)

s2 = LLSet()
s2.add(2)
s2.add(3)

s1.difference(s2)
assert 1 in s1
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

```python
from typing import List

def calc(ops:List[int], dig:List[int]) -> int:
    while ops:
        o = Op(ops.pop())
        d1 = dig.pop()
        d2 = dig.pop()
        dig.append(doop(o, d1, d2))
    tmp = dig.pop()
    return tmp
```

Queues

- Can be implemented using norma lists
- A better (more efficient) alterantive 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