CS 61A More OOP, Linked Lists, Complexity
Spring 2021 Lost 6: March 12, 2021 Solutions

1 Learning Goals

- Get some hands-on practice with OOP
- Understand the structure of linked lists (woohoo interview prep!)
- Begin to understand how to analyze the runtime of a program

2 OOP

2.1 What is the relationship between a class and an ADT?

In general, we can think of an abstract data type as defined by some collection of selectors and constructors, together with some behavior conditions. As long as the behavior conditions are met (such as the division property above), these functions constitute a valid representation of the data type.

There are two different layers to the abstract data type:

1) The program layer, which uses the data, and
2) The concrete data representation that is independent of the programs that use the data. The only communication between the two layers is through selectors and constructors that implement the abstract data in terms of the concrete representation.

Classes are a way to implement an Abstract Data Type. But, ADTs can also be created using a collection of functions, as shown by the rational number example. (See Composing Programs 2.2)

2.2 What is the definition of a Class? What is the definition of an Instance?

Class: a template for all objects whose type is that class that defines attributes and methods that an object of this type has.
Instance: A specific object created from a class. Each instance shares class attributes and stores the same methods and attributes. But the values of the attributes are independent of other instances of the class. For example, all humans have two eyes but the color of their eyes may vary from person to person.

2.3 What is a Class Attribute? What is an Instance Attribute?

Class Attribute: A static value that can be accessed by any instance of the class and is shared among all instances of the class.
Instance Attribute: A field or property value associated with that specific instance of the object.

2.4 What Would Python Display?

```
class Foo():
    x = 'bam'
    def __init__(self, x):
        self.x = x
    def baz(self):
        return self.x

class Bar(Foo):
```

```
        x = 'boom'
        def __init__(self, x):
                Foo.__init__(self, 'er' + x)
        def baz(self):
                return Bar.x + Foo.baz(self)

foo = Foo('boo')

Foo.x

'bam'

foo.x

'boo'

foo.baz()

'boo'

Foo.baz()

Error

Foo.baz(foo)

'boo'

bar = Bar('ang')
Bar.x

'boom'

bar.x

'erang'

bar.baz()

'boomerang'
```

2.5 What Would Python Display?

```
class Student:
        def __init__(self, subjects):
                self.current_units = 16
                self.subjects_to_take = subjects
                self.subjects_learned = {}
                self.partner = None

        def learn(self, subject, units):
```

```
                print('I just learned about ' + subject)
                self.subjects_learned[subject] = units
                self.current_units -= units

        def make_friends(self):
                if len(self.subjects_to_take) > 3:
                        print('Whoa! I need more help!')
                        self.partner = Student(self.subjects_to_take[1:])
                else:
                        print("I'm on my own now!")
                        self.partner = None

        def take_course(self):
                course = self.subjects_to_take.pop()
                self.learn(course, 4)
                if self.partner:
                        print('I need to switch this up!')
                        self.partner = self.partner.partner
                        if not self.partner:
                                print('I have failed to make a friend :(')

tim = Student(['Chem1A', 'Bio1B', 'CS61A', 'CS70', 'CogSci1'])
tim.make_friends()
```

```
Whoa! I need more help!
```

```
print(tim.subjects_to_take)
```

```
['Chem1A', 'Bio1B', 'CS61A', 'CS70', 'CogSci1']
```

```
tim.partner.make_friends()
```

```
Whoa! I need more help!
```

```
tim.take_course()
```

```
I just learned about CogSci1
I need to switch this up!
```

```
tim.partner.take_course()
```

```
I just learned about CogSci1
```

```
tim.take_course()
```

```
I just learned about CS70
I need to switch this up!
I have failed to make a friend :(
```

```
tim.make_friends()
```

I'm on my own now!

2.6 Fill in the implementation for the Cat and Kitten classes. When a cat meows, it should say "Meow, (name) is hungry" if it is hungry, and "Meow, my name is (name)" if not. Kittens do the same thing as cats, except they say "i'm baby" instead of "meow", and they say "I want mama (parent's name)" after every call to meow().

```
>>>cat = Cat('Tuna')
>>>kitten = kitten('Fish', cat)
>>>cat.meow()
meow, Tuna is hungry
>>>kitten.meow()
i'm baby, Fish is hungry
I want mama Tuna
>>>cat.eat()
meow
>>>cat.meow()
meow, my name is Tuna
>>>kitten.eat()
i'm baby
>>>kitten.meow()
meow, my name is Fish
I want mama Tuna
```

```
class Cat():
    noise = 'meow'
    def __init__(self, name):

        self.name = name
        self.hungry = True
    def meow(self):

        if self.hungry:
            print(self.noise + ', ' + self.name ' + is hungry!')
        else:
            print(self.noise + ', my name is ' + self.name)
    def eat(self):
        print(self.noise)
        self.hungry = False

class Kitten(Cat):

    noise = "i'm baby"
    def __init__(self, name, parent):
        Cat.__init__(self, name)
        self.parent = parent
```

```
    def meow(self):
        Cat.meow(self)
        print('I want mama' + parent.name)
```

3 Linked Lists

3.1 Introductory

3.1 What is a linked list? Why do we consider it a naturally recursive structure?

A linked list is a data structure with a first and a rest, where the first is some arbitrary element and the rest MUST be another linked list (or Link.empty)

3.2 Draw a box and pointer diagram for the following:

```
Link('c', Link(Link(6, Link(1, Link('a'))), Link('s')))
```

3.3 The Link class can represent lists with cycles. That is, a list may contain itself as a sublist. Implement **has_cycle** that returns whether its argument, a Link instance, contains a cycle. There are two ways to do this: iteratively with two pointers, or keeping track of Link objects we've seen already. Try to come up with both!

```
def has_cycle(link):
    """
    >>> s = Link(1, Link(2, Link(3)))
    >>> s.rest.rest.rest = s
    >>> has_cycle(s)
    True
    """


    # solution 1
    tortoise = link
    hare = link.rest
    while tortoise.rest and hare.rest and hare.rest.rest:
        if tortoise is hare:
            return True
        tortoise = tortoise.rest
        hare = hare.rest.rest
    return False

    # solution 2
    seen = []
    while link.rest:
        if link in seen:
            return True
        seen.append(link)
        link = link.rest
    return False
```

3.4 Fill in the following function, which checks to see if **sub_link**, a particular sequence of items in one linked list, can be found in another linked list (the items have to be in order, but not necessarily consecutive).

```
def seq_in_link(link, sub_link):
    """
    >>> lnk1 = Link(1, Link(2, Link(3, Link(4))))
    >>> lnk2 = Link(1, Link(3))
    >>> lnk3 = Link(4, Link(3, Link(2, Link(1))))
    >>> seq_in_link(lnk1, lnk2)
    True
    >>> seq_in_link(lnk1, lnk3)
    False
    """

    if sub_link is Link.empty:
        return True
    if link is Link.empty:
        return False
    if link.first == sub_link.first:
        return seq_in_link(link.rest, sub_link.rest)
    else:
        return seq_in_link(link.rest, sub_link)
```

3.2 Medium

3.1 Write a function that takes a sorted linked list of integers and mutates it so that all duplicates are removed.

```
def remove_duplicates(lnk):
    """
    >>> lnk = Link(1, Link(1, Link(1, Link(1, Link(5)))))
    >>> remove_duplicates(lnk)
    >>> lnk
    Link(1, Link(5))
    """
```

Recursive solution:

```
    if lnk is Link.empty or lnk.rest is Link.empty:
        return
    if lnk.first == lnk.rest.first:
        lnk.rest = lnk.rest.rest
        remove_duplicates(lnk)
    else:
        remove_duplicates(lnk.rest)
```

For a list of one or no items, there are no duplicates to remove.

Now consider two possible cases:

- If there is a duplicate of the first item, we will find that the first and second items in the list will have the same values (that is, `lnk.first == lnk.rest.first`). We can confidently state this because we were told that the input linked list is in sorted order, so duplicates are adjacent to each other. We'll remove the second item from the list.

  Finally, it's tempting to recurse on the remainder of the list (`lnk.rest`), but remember that there could still be more duplicates of the first item in the rest of the list! So we have to recurse on `lnk` instead. Remember that we have removed an item from the list, so the list is one element smaller than before. Normally, recursing on the same list wouldn't be a valid subproblem.

- Otherwise, there is no duplicate of the first item. We can safely recurse on the remainder of the list.

Iterative solution:

```
    while lnk is not Link.empty and lnk.rest is not Link.empty:
        if lnk.first == lnk.rest.first:
            lnk.rest = lnk.rest.rest
        else:
            lnk = lnk.rest
```

The loop condition guarantees that we have at least one item left to consider with `lnk`.

For each item in the linked list, we pause and remove all adjacent items that have the same value. Once we see that `lnk.first != lnk.rest.first`, we can safely advance to the next item. Once again, this takes advantage of the property that our input linked list is sorted.

3.3 Hard

3.1 Define `reverse`, which takes in a linked list and reverses the order of the links. The function may *not* return a new list; it must mutate the original list. Return a pointer to the head of the reversed list.

```
def reverse(lnk):
    """
    >>> a = Link(1, Link(2, Link(3)))
    >>> r = reverse(a)
    >>> r.first
    3
    >>> r.rest.first
    2
    """
```

Recursive solution:

```
    if lnk is Link.empty or lnk.rest is Link.empty:
        return lnk
    rest_rev = reverse(lnk.rest)
    lnk.rest.rest = lnk
    lnk.rest = Link.empty
    return rest_rev
```

For the base case, a linked list with no items or a single item is trivial to reverse.

Let's formally name our variables to make the explanation of the following process a bit easier. The original list is `lnk`, we reverse `lnk.rest` recursively and get back a pointer to the head of the reversed version of `lnk.rest`, which is `rest_rev`.

Notice that `lnk.rest` is the last item of the list referred to by `rest_rev`. All we have to do is attach the first item of `lnk` to the end of the reversed rest, and then make sure that `lnk.rest` is the empty list as it is now the last item in the reversed list.

Video walkthrough

Iterative solution (1):

```
    if lnk is Link.empty:
        return lnk
    cur = lnk
    nxt = lnk.rest
    cur.rest = Link.empty

    while nxt is not Link.empty:
        after = nxt.rest
```

```
        nxt.rest = cur
        cur = nxt
        nxt = after
    return cur
```

The iterative solution is quite different from the recursive solution. We go through every item in our linked list, and reattach them in reverse order. That is, we attach the second item back to the first item, the third item back to the second item, and so on.

The tricky part is figuring what information we need to keep track of in order to do this. We use a two pointer method that tracks a current and a following position in a linked list. The logic is not too complicated, but the best way to understand it is to work through an example with a box and pointer diagram.

Iterative solution (2):

```
    new_lnk = Link.empty
    while lnk is not Link.empty:
        new_lnk, lnk.rest, lnk = lnk, new_lnk, lnk.rest
    return new_lnk
```

Here's yet another iterative approach, different from both the previous iterative and recursive approaches.

We begin by asking how we'd gather the values in a linked list into a list (not a linked list, but a regular Python list) in reverse order.

```
>>> xs = Link(1, Link(2, Link(3)))
>>> xs
Link(1, Link(2, Link(3)))
>>> reverse(xs)
[3, 2, 1]
```

We could do this by iterating through the linked list and inserting the values into the front of the list:

```
def reverse(lnk):
    new_list = []
    while lnk is not Link.empty:
        new_list.insert(0, lnk.first)  # Insert the link value before all existing elements of
    new_list
        lnk = lnk.rest
    return new_list
```

This works because if value A comes before value B in the original list, then B will be inserted before value A in our new list.

We could keep this same approach, but have `new_lnk` be a linked list instead of an ordinary list.

```
def reverse(lnk):
    new_lnk = Link.empty
```

```
    while lnk is not Link.empty:
        new_lnk = Link(lnk.first, new_lnk)  # Create a new link and insert it before all existing
    links in new_lnk
        lnk = lnk.rest
    return new_lnk
```

This works, but we are not allowed to create new `Link` instances.

So, instead of copying `lnk` by constructing a new `Link` with the same first attribute as `lnk`, we should just make `lnk` the new head of `new_lnk`. But that means we simultaneously need to make three updates:

- `new_lnk` should point at what `lnk` was pointing at (because `lnk` is the new head of `new_lnk`).
- `lnk.rest` should point at what `new_lnk` was pointing at (this makes `lnk` the new head of `new_lnk`).
- `lnk` should point at what `lnk.rest` was pointing at (because we still need to iterate through the original list).

This leads us the final solution presented earlier.

4 Growth

4.1 What is the runtime of the following function?

```
def one(n):
    if 1 == 1:
        return None
    return n
```

a. Theta(1) b. Theta(log n) c. Theta(n) d. Theta(n^2) e. Theta(2^n)

Theta(1) - the function always returns None, because 1 == 1 is always True. And even if it was a false statement, the function would just return n. So since the runtime of the function doesn't change with respect to the size of the input, it is constant time.

4.2 What is the runtime of the following function?

```
def two(n):
    for i in range(n):
        print(n)
```

a. Theta(1) b. Theta(log n) c. Theta(n) d. Theta(n^2) e. Theta(2^n)

Theta(n) - the function iterates n times; if n increases by 1, the function loops 1 additional time. Therefore there is a linear relationship between the input size and runtime.

4.3 What is the runtime of the following function?

```
def three(n):
    while n > 0:
       n = n // 2
```

a. Theta(1) b. Theta(log n) c. Theta(n) d. Theta(n^2) e. Theta(2^n)

Theta(log n) - The function continues to loop as long as n ¿ 0. Inside the while loop, we divide n by 2 every loop. So to get the function to loop one additional time, we need to double our original input size. This is a logarithmic relationship between input size and runtime.

4.4 What is the runtime of the following function?

```
def four(n):
    for i in range(n):
        for j in range(i):
            print(str(i), str(j))
```

a. Theta(1) b. Theta(log n) c. Theta(n) d. Theta(n^2) e. Theta(2^n)

d. Theta(n^2) - The outer loop loops through every number from 0 to n. The inner loop loops corresponding to the outer loop. So the total number of loops from the inner loop looks like this: 0 + 1 + 2 + 3 + 4 ... + n. This is the summation of the first n natural numbers = n(n + 1)/2, which asymptotically is Theta(n^2)

4.5 What is the runtime of the following function?

```
def five(n):
    if n <= 0:
        return 1
    return five(n - 1) + five(n - 2)
```

a. Theta(1) b. Theta(log n) c. Theta(n) d. Theta(n^2) e. Theta(2^n)

e. Theta(2^n) - Draw out the tree of recursive calls. You should see that every node branches out into 2 more nodes. Since the base case returns when n <= 0, and each recursive call subtracts 1 or 2 from n, the height of our tree is n. We're branching out by a factor of 2 each layer for n layers – that means we'll have 2^n nodes in our tree of recursive calls. Each 'node' represents 1 'unit of work' as all the function does is return something. So 1 unit of work across 2^n nodes is 2^n total.

4.6 What is the runtime of the following function?

```
def five(n):
    if n <= 0:
        return 1
    return five(n//2) + five(n//2)
```

a. Theta(1) b. Theta(log n) c. Theta(n) d. Theta(n^2) e. Theta(2^n)

c. Theta(n) - Draw out the tree of recursive calls. You should see that every node branches out into 2 more nodes. Since the base case returns when n <= 0, and each recursive call divides n by 2, the height of our tree is logn (by the same logic as three(n): if we want one additional layer in our tree, our original input has to be doubled, which is a logarithmic relationship). We're branching out by a factor of 2 each layer for logn layers – that means we'll have $2^(logn) = n$ nodes in our tree of recursive calls. Each 'node' represents 1 'unit of work' as all the function does is return something. So 1 unit of work across n nodes is n total.