http://bit.ly/1C7xlgT

Instructions

Form a group of 3-5. Start on problem 0. Check off with a lab assistant when everyone in your group understands how to solve problem 0. Repeat for problem 1, 2, ...
You're not allowed to move on from a problem until you check off with a lab assistant. You are allowed to use any and all resources at your disposal, including the interpreter, lecture notes and slides, discussion notes, and labs. You may consult the lab assistants, **but only after you have asked everyone else in your group. The purpose of this section is to have all the students working together to learn the material and get better at Computer Science together.**

Reference

Definitions of the Link and Tree classes can be found in the appendix.

Introduction

0. What is an Abstract Data Type?

A structure that stores data but separates how it is stored and how it is used.

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.

In addition to selectors and constructors, there are behavior conditions under which the selectors and constructors give an appropriate response. That is, if we construct a rational number data type x from integers n and d, then it should be in case that numer(x)/denom(x) is equal to n/d.

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.

1. What is the relationship between a class and an ADT?

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. Define the following:

Object - An instance of a particular class. A combination of local state and behavior hidden behind an abstraction barrier. They can communicate with each other, share common attributes and methods, and inherit characteristics from related classes.

Instance - A specific object created from a class. Each instance shares class ni variables
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.

Class - Template for all objects whose type is that class that defines attributes and methods that an object of this type has.

Parent Class - The class from which this class inherits from. A parent class typically has a more general purpose or definition compared to this class. For example, the Dog class inherits from the Animal class. There are qualities such as a need for water and food that all animals share that can be defined in the Animal class.

Class Variable - A static variable that can be accessed by any instance of the class and is shared among all instances of the class.

Instance Variable - A field or property value associated with that specific instance of the object.

Method - Functions that operate on the object or perform object-specific computations.

3. How are linked lists and trees related? (How can we represent a linked list as a tree?)

Linked lists and trees are both recursive objects. The “rest” of a linked list is another linked list. The branches of a tree are also trees. We can make a pseudo-list as a tree by representing each element in the list as a tree with one branch that contains the rest of the list.

4. Why can we change the value of the root of a tree when using the Tree class implementation but not the ADT?

We cannot change the value at the root of a tree in an ADT because we do not know how the tree is implemented. Constructors make a tree, selectors get values from a tree, but there are no functions defined that allow us to change a value. It is possible, however, to modify the branches of an ADT. This is because we can mutate the list of branches that is returned from a call to branches.

5. How can we modify the tree ADT in order to handle root mutation?

Create a function `set_entry(tree, new_entry)` to set the entry of a tree.

Object Oriented Programming

0. What would Python print?

```
y = 42

class Foo(object):
   x = 'bam'
   y = 'barbie'

   def __init__(self, x):
       self.x = x

   def baz(self):
       return type(self).x + self.x

   def aussie(self):
       return y

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

class Foobar(Bar):
   y = 'mate'
```

```
>>> foo = Foo('boo')
>>> Foo.x
'bam'
>>> foo.x
'boo'
>>> foo.baz()
'bamboo'
>>> Foo.baz()
Error
>>> Foo.baz(foo)
'bamboo'
>>> bar = Foobar('ang')
>>> Bar.x
'boom'
>>> bar.x
'erang'
>>> bar.baz()
'boomerang'
>>> foo.aussie()
42
>>> Foobar.baz = lambda best: best.y
>>> bar.baz()
'mate'
>>> bar.baz = lambda shrimp: shrimp.y
>>> bar.baz(foo)
'barbie'
```

1.Complete the `Person`, `Residence`, and `Apartment` classes in the implementation on the next few pages so that the doctests below pass.

```
>>> christy = Person('Christy')
>>> katherine = Person('Katherine')
>>> max = Person('Max')
>>> palace = Apartment(12345)
>>> palace.add_roommate(christy)
>>> palace.add_roommate(katherine)
>>> palace.add_roommate(max)
Too many roommates!
>>> palace.enter(katherine, 12345)
Katherine got home
>>> palace.clean(katherine)
Thanks for cleaning, Katherine
>>> palace.enter(christy, 12333)
Break-in attempted!
>>> Residence.enter(palace, christy)
Christy got home
>>> palace.enter(max, 12345)
Break-in attempted!
>>> estate = Residence()
>>> estate.add_roommate(max)
>>> estate.clean(max)
Max is not home
>>> estate.enter(max)
Max got home
>>> estate.clean(max)
Thanks for cleaning, Max
```

```
class Person(object):
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return self.name

class Residence(object):

    def __init__(self):
        self.roommates = []
        self.at_home = []

    def add_roommate(self, person):
        self.roommates += [person]

    def enter(self, person):
        if person not in self.roommates:
            print('Break-in attempted!')
        else:
            print(person, 'got home')
            self.at_home += [person]

    def exit(self, person):
        if person in self.at_home:
            self.at_home.remove(person)

    def clean(self, cleaner):
        if cleaner not in self.at_home:
            print(cleaner, 'is not home')
        else:
            print('Thanks for cleaning,', cleaner)
```

```
class Apartment(Residence):
    max_roommates = 2

    def __init__(self, door_code):
        Residence.__init__(self)
        self.door_code = door_code

    def add_roommate(self, person):
        if len(self.roommates) >= self.max_roommates:
            print('Too many roommates!')
        else:
            Residence.add_roommate(self, person)

    def enter(self, person, door_code):
        if door_code != self.door_code:
            print('Break-in attempted!')
        else:
            Residence.enter(self, person)
```

Linked Lists

0. Draw box and pointer diagrams for the following code. What would Python print?

```
>>> x = Link(1, Link(4))
>>> y = Link(3)
>>> z = Link(x, y)
>>> z.first.rest.rest = z.rest
>>> z.first
Link(1, Link(4, Link(3)))
>>> y.rest = z
>>> a = Link(z, x)
>>> a.first.rest.rest.first.rest.first
4
```

1. Write a function `link_range` that creates a list with all integers between `low` (inclusive) and `high` (exclusive), similar to Python's `range`.

```
def link_range(low, high):
    """
    >>> link_range(1, 5)
    Link(1, Link(2, Link(3, Link(4))))
    """
    if low >= high:
        return Link.empty
    else:
        return Link(low, link_range(low + 1, high))
```

2. Write a function `stretch` that takes a list and creates a new list with each element duplicated `factor` times.

```
def stretch(link, factor):
    """
    >>> stretch(link_range(1, 4), 2)
    Link(1, Link(1, Link(2, Link(2, Link(3, Link(3))))))
    """
    if link is Link.empty:
        return Link.empty
    else:
        stretched = stretch(link.rest, factor)
        for _ in range(factor):
            stretched = Link(link.first, stretched)
        return stretched
```

3. Write a function `extend` that takes two linked lists and appends the second list to the first. You should not create any new links (you cannot call the `Link` constructor). You may assume that the first list is nonempty.

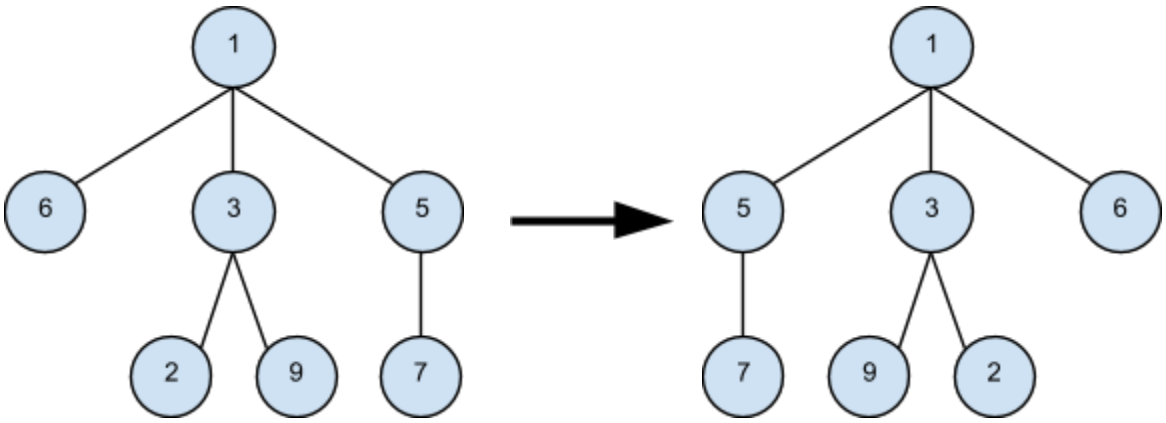
```
def extend(left, right):
    """
    >>> left = link_range(1, 4)
    >>> right = link_range(4, 7)
    >>> extend(left, right)
    >>> left
    Link(1, Link(2, Link(3, Link(4, Link(5, Link(6))))))
    """
    if left.rest is Link.empty:
        left.rest = right
    else:
        extend(left.rest, right)
```

4. Write a function `cycle` that takes a nonempty linked list and makes it cyclic. Your solution should be one line (hint: you can use previous parts).

```
def cycle(link):
    """
    >>> link = link_range(1, 7)
    >>> cycle(link)
    >>> [link[i] for i in range(20)]
    [1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6, 1, 2]
    """
    extend(link, link)
```

Trees

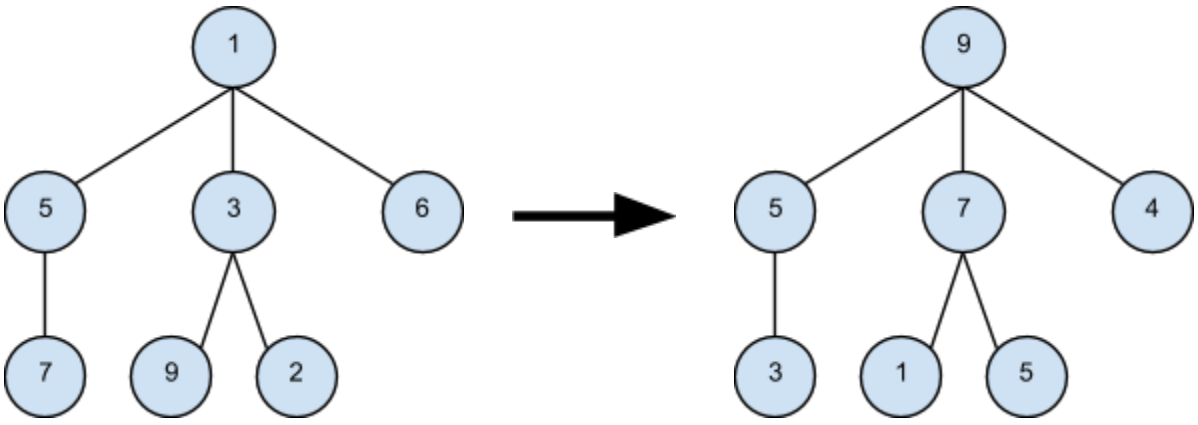
0. Write a function `reflect` that takes in a `Tree` and reflects it over its center. This function should mutate its argument and not return a copy. For example, if `t` refers to the following tree



then after calling `reflect(t)`, `t` should now refer to the tree

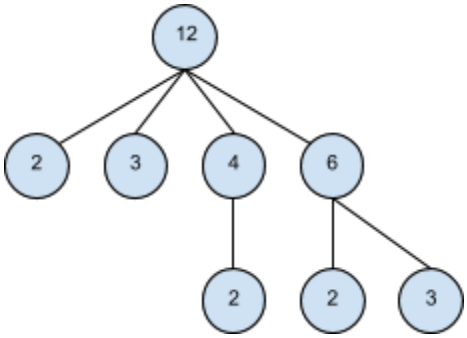
```
def reflect(t):
    t.branches = [reflect(b) for b in t.branches[::-1]]
```

1. Write a function `map_mutate` that takes in a `Tree` and a `function` and applies `function` to every entry. This function should mutate its argument and not return a copy. For example, if `t` is tree on the left, then after `map_mutate(lambda x: 10 - x, t)`, `t` should be the tree on the right:



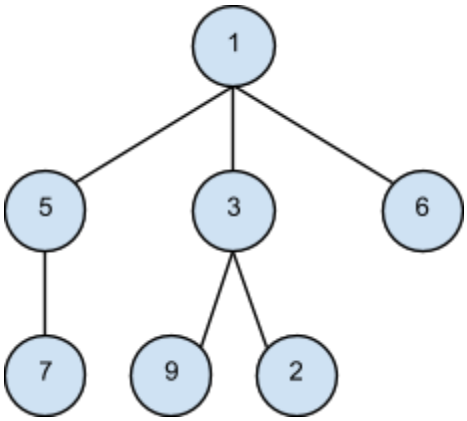

```
def map_mutate(fn, tree):
    tree.entry = fn(tree.entry)
    for b in tree.branches:
        map_mutate(fn, b)
```

2. Write a function `make_factor_tree` that takes in a natural num `num` and returns a tree with `num` at the root and all of `num`'s factors (excluding 1 and `num`) as branches. Each branch should also contain all of it's entry's factors. For example, `make_factor_tree(12)` is




```
def make_factor_tree(num):
    factors = [make_factor_tree(i) for i in range(2, num) if num % i == 0]
    return Tree(num, factors)
```

3. Write a function `post_order` that takes in a tree and returns a linked list of the entries of the tree in post-order. Post-order has all of the entries of a tree's branches from left to right before the root. For example in the tree below, the post-order is 7, 5, 9, 2, 3, 6, 1.



(Hint: define a helper function with a partially-constructed list as a parameter.)

```
def post_order(t):
    """
    >>> post_order(make_factor_tree(12))
    Link(2, Link(3, Link(2, Link(4, Link(2, Link(3, Link(6, Link(12)))))))
    """
    def post_order_explore(t, rest):
        rest = Link(t.entry, rest)
        for b in t.branches[::-1]:
            rest = post_order_explore(b, rest)
        return rest
    return post_order_explore(t, Link.empty)
```

Appendix

```
class Link:
    """A linked list.

    >>> s = Link(1, Link(2, Link(3, Link(4))))
    >>> len(s)
    4
    >>> s[2]
    3
    >>> s
    Link(1, Link(2, Link(3, Link(4))))
    """
    empty = ()

    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest

    def __getitem__(self, i):
        if i == 0:
            return self.first
        else:
            return self.rest[i-1]

    def __len__(self):
        return 1 + len(self.rest)

    def __repr__(self):
        if self.rest is not Link.empty:
            rest_str = ', ' + repr(self.rest)
        else:
            rest_str = ''
        return 'Link({0}{1})'.format(repr(self.first), rest_str)
```

```
class Tree:
    """A rooted tree."""
    def __init__(self, entry, branches=[]):
        self.entry = entry
        for branch in branches:
            assert isinstance(branch, Tree)
        self.branches = branches

    def __repr__(self):
        if self.branches:
            rest_str = ', ' + repr(self.branches)
        else:
            rest_str = ''
        return 'Tree({0}{1})'.format(repr(self.entry), rest_str)
```