Homework 5: Object-Oriented Programming, Linked Lists, Trees

> Adapted from cs61a of UC Berkeley.

Readings

You might find the following references useful:

- [Section 2.5](#)
- [Section 2.9](#)

Starter Files

Get your starter file by cloning the repository: https://github.com/JacyCui/sicp-hw05.git

```
1  git clone https://github.com/JacyCui/sicp-hw05.git
```

`hw05.zip` is the starter file you need, you might need to unzip the file to get the skeleton code.

```
1  unzip hw05.zip
```

`README.md` is the handout for this homework. `solution` is a probrab solution of the homework. However, I might not give my solution exactly when the homework is posted. You need to finish the task on your own first. If any problems occurs, please make use of the comment section.

Required Questions

OOP

Q1: Vending Machine

Create a class called `VendingMachine` that represents a vending machine for some product. A `VendingMachine` object returns strings describing its interactions.

Fill in the `VendingMachine` class, adding attributes and methods as appropriate, such that its behavior matches the following doctests:

```
1  class VendingMachine:
2      """A vending machine that vends some product for some price.
```

```
 3
 4          >>> v = VendingMachine('candy', 10)
 5          >>> v.vend()
 6          'Inventory empty. Restocking required.'
 7          >>> v.add_funds(15)
 8          'Inventory empty. Restocking required. Here is your $15.'
 9          >>> v.restock(2)
10          'Current candy stock: 2'
11          >>> v.vend()
12          'You must add $10 more funds.'
13          >>> v.add_funds(7)
14          'Current balance: $7'
15          >>> v.vend()
16          'You must add $3 more funds.'
17          >>> v.add_funds(5)
18          'Current balance: $12'
19          >>> v.vend()
20          'Here is your candy and $2 change.'
21          >>> v.add_funds(10)
22          'Current balance: $10'
23          >>> v.vend()
24          'Here is your candy.'
25          >>> v.add_funds(15)
26          'Inventory empty. Restocking required. Here is your $15.'
27
28          >>> w = VendingMachine('soda', 2)
29          >>> w.restock(3)
30          'Current soda stock: 3'
31          >>> w.restock(3)
32          'Current soda stock: 6'
33          >>> w.add_funds(2)
34          'Current balance: $2'
35          >>> w.vend()
36          'Here is your soda.'
37          """
38          "*** YOUR CODE HERE ***"
```

You may find Python string formatting syntax useful. A quick example:

```
1   >>> ten, twenty, thirty = 10, 'twenty', [30]
2   >>> '{0} plus {1} is {2}'.format(ten, twenty, thirty)
3   '10 plus twenty is [30]'
```

Use Ok to test your code:

```
1   python3 ok -q VendingMachine --local
```

Q2: Mint

Complete the `Mint` and `Coin` classes so that the coins created by a mint have the correct year and worth.

- Each `Mint` instance has a `year` stamp. The `update` method sets the `year` stamp to the `current_year` class attribute of the `Mint` class.
- The `create` method takes a subclass of `Coin` and returns an instance of that class stamped with the `mint`'s year (which may be different from `Mint.current_year` if it has not been updated.)
- A `Coin`'s `worth` method returns the `cents` value of the coin plus one extra cent for each year of age beyond 50. A coin's age can be determined by subtracting the coin's year from the `current_year` class attribute of the `Mint` class.

```python
1  class Mint:
2      """A mint creates coins by stamping on years.
3
4      The update method sets the mint's stamp to Mint.current_year.
5
6      >>> mint = Mint()
7      >>> mint.year
8      2020
9      >>> dime = mint.create(Dime)
10     >>> dime.year
11     2020
12     >>> Mint.current_year = 2100  # Time passes
13     >>> nickel = mint.create(Nickel)
14     >>> nickel.year      # The mint has not updated its stamp yet
15     2020
16     >>> nickel.worth()  # 5 cents + (80 - 50 years)
17     35
18     >>> mint.update()    # The mint's year is updated to 2100
19     >>> Mint.current_year = 2175     # More time passes
20     >>> mint.create(Dime).worth()    # 10 cents + (75 - 50 years)
21     35
22     >>> Mint().create(Dime).worth()  # A new mint has the current year
23     10
24     >>> dime.worth()     # 10 cents + (155 - 50 years)
25     115
26     >>> Dime.cents = 20  # Upgrade all dimes!
27     >>> dime.worth()     # 20 cents + (155 - 50 years)
28     125
29     """
30     current_year = 2020
31
32     def __init__(self):
33         self.update()
34
35     def create(self, kind):
36         "*** YOUR CODE HERE ***"
37
```

```
38        def update(self):
39            "*** YOUR CODE HERE ***"
40
41    class Coin:
42        def __init__(self, year):
43            self.year = year
44
45        def worth(self):
46            "*** YOUR CODE HERE ***"
47
48    class Nickel(Coin):
49        cents = 5
50
51    class Dime(Coin):
52        cents = 10
```

Use Ok to test your code:

```
1    python3 ok -q Mint --local
```

Linked Lists

Q3: Store Digits

Write a function `store_digits` that takes in an integer `n` and returns a linked list where each element of the list is a digit of `n`.

> Note: do not use any string manipulation functions like `str` and `reversed`

```
1    def store_digits(n):
2        """Stores the digits of a positive number n in a linked list.
3
4        >>> s = store_digits(1)
5        >>> s
6        Link(1)
7        >>> store_digits(2345)
8        Link(2, Link(3, Link(4, Link(5))))
9        >>> store_digits(876)
10       Link(8, Link(7, Link(6)))
11       >>> # a check for restricted functions
12       >>> import inspect, re
13       >>> cleaned = re.sub(r"#.*\\n", '', re.sub(r'"{3}[\s\S]*?"{3}', '',
     inspect.getsource(store_digits)))
14       >>> print("Do not use str or reversed!") if any([r in cleaned for r in ["str",
     "reversed"]]) else None
15       """
16       "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
1  python3 ok -q store_digits --local
```

Extra Questions

Trees

Q4: Is BST

Write a function `is_bst`, which takes a Tree `t` and returns `True` if, and only if, `t` is a valid binary search tree, which means that:

- Each node has at most two children (a leaf is automatically a valid binary search tree)
- The children are valid binary search trees
- For every node, the entries in that node's left child are less than or equal to the label of the node
- For every node, the entries in that node's right child are greater than the label of the node

An example of a BST is:
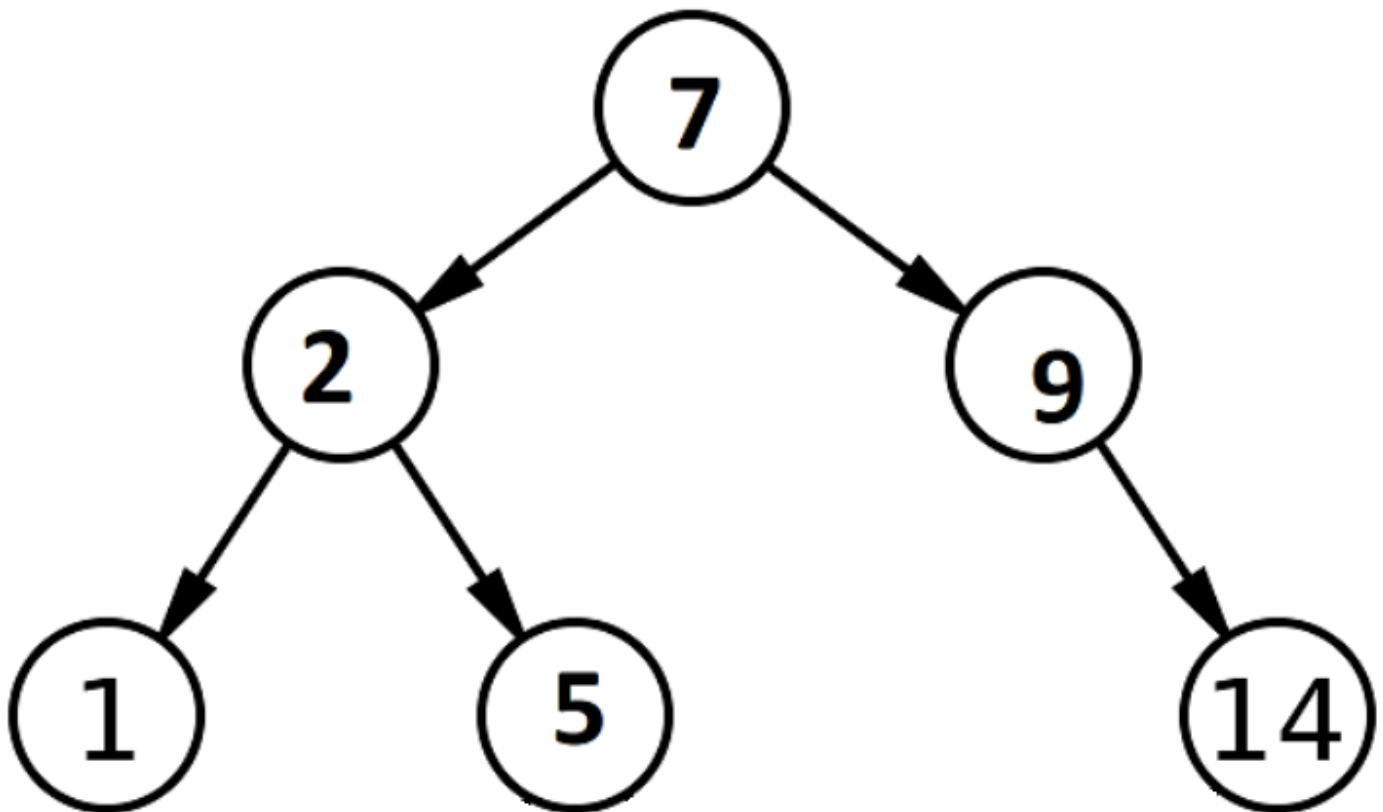
Note that, if a node has only one child, that child could be considered either the left or right child. You should take this into consideration.

> *Hint:* It may be helpful to write helper functions `bst_min` and `bst_max` that return the minimum and maximum, respectively, of a Tree if it is a valid binary search tree.

```
1   def is_bst(t):
2       """Returns True if the Tree t has the structure of a valid BST.
3
4       >>> t1 = Tree(6, [Tree(2, [Tree(1), Tree(4)]), Tree(7, [Tree(7), Tree(8)])])
5       >>> is_bst(t1)
6       True
7       >>> t2 = Tree(8, [Tree(2, [Tree(9), Tree(1)]), Tree(3, [Tree(6)]), Tree(5)])
8       >>> is_bst(t2)
9       False
10      >>> t3 = Tree(6, [Tree(2, [Tree(4), Tree(1)]), Tree(7, [Tree(7), Tree(8)])])
11      >>> is_bst(t3)
12      False
13      >>> t4 = Tree(1, [Tree(2, [Tree(3, [Tree(4)])])])
14      >>> is_bst(t4)
15      True
16      >>> t5 = Tree(1, [Tree(0, [Tree(-1, [Tree(-2)])])])
17      >>> is_bst(t5)
18      True
19      >>> t6 = Tree(1, [Tree(4, [Tree(2, [Tree(3)])])])
20      >>> is_bst(t6)
21      True
22      >>> t7 = Tree(2, [Tree(1, [Tree(5)]), Tree(4)])
23      >>> is_bst(t7)
24      False
25      """
26      "*** YOUR CODE HERE ***"
```
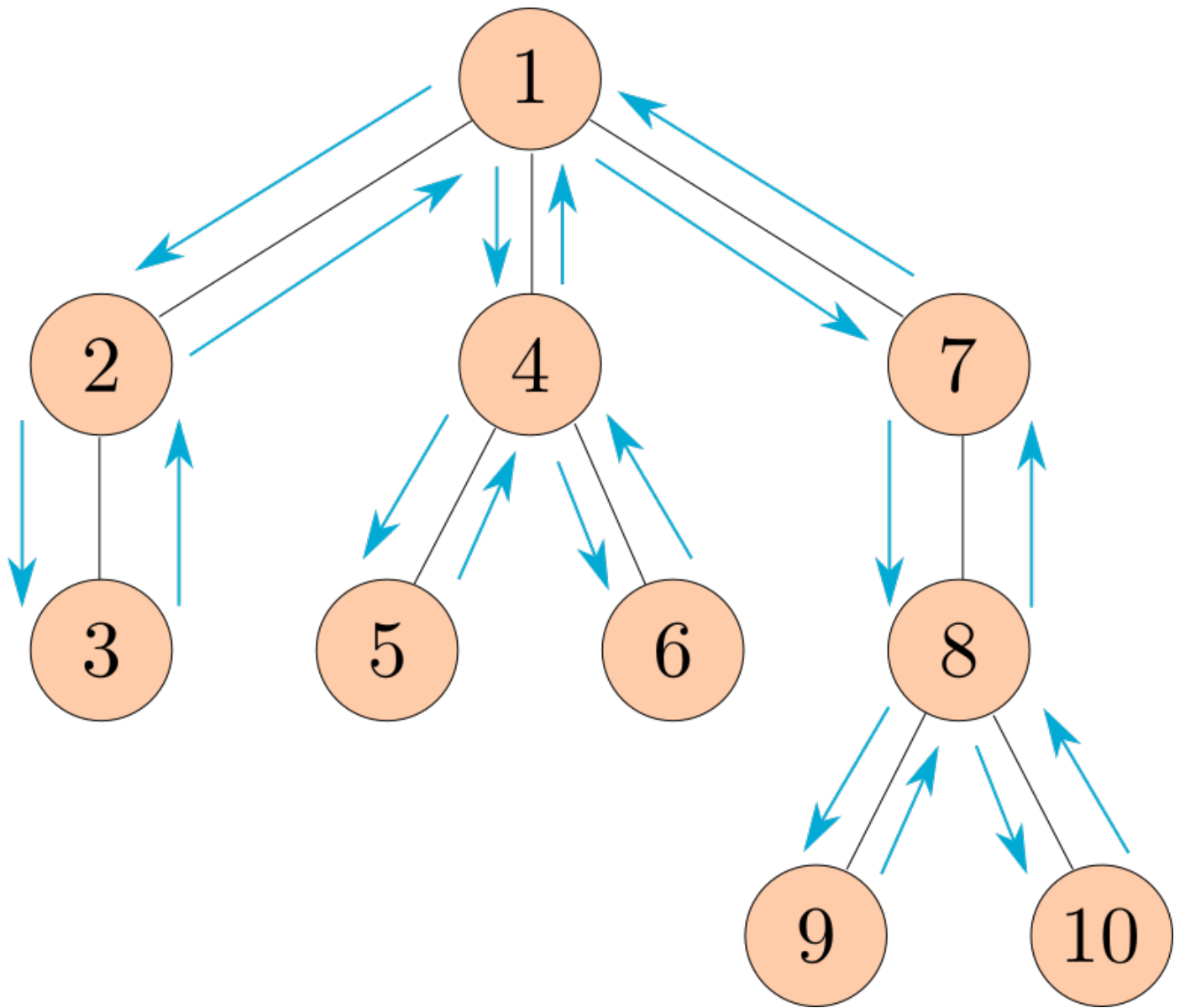
Use Ok to test your code:

```
1   python3 ok -q is_bst --local
```

Q5: Preorder

Define the function `preorder`, which takes in a tree as an argument and returns a list of all the entries in the tree in the order that `print_tree` would print them.

The following diagram shows the order that the nodes would get printed, with the arrows representing function calls.

> *Note*: This ordering of the nodes in a tree is called a preorder traversal.

```
1   def preorder(t):
2       """Return a list of the entries in this tree in the order that they
3       would be visited by a preorder traversal (see problem description).
4
5       >>> numbers = Tree(1, [Tree(2), Tree(3, [Tree(4), Tree(5)]), Tree(6,
        [Tree(7)])])
6       >>> preorder(numbers)
7       [1, 2, 3, 4, 5, 6, 7]
8       >>> preorder(Tree(2, [Tree(4, [Tree(6)])]))
9       [2, 4, 6]
10      """
11      "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
1   python3 ok -q preorder --local
```

Generators/Trees

Q6: Yield Paths

Define a generator function `path_yielder` which takes in a Tree `t`, a value `value`, and returns a generator object which yields each path from the root of `t` to a node that has label `value`.

`t` is implemented with a class, not as the function-based ADT.

Each path should be represented as a list of the labels along that path in the tree. You may yield the paths in any order.

We have provided a (partial) skeleton for you. You do not need to use this skeleton, but if your implementation diverges significantly from it, you might want to think about how you can get it to fit the skeleton.

```
1   def path_yielder(t, value):
2       """Yields all possible paths from the root of t to a node with the label value
3       as a list.
4
5       >>> t1 = Tree(1, [Tree(2, [Tree(3), Tree(4, [Tree(6)]), Tree(5)]), Tree(5)])
6       >>> print(t1)
7       1
8         2
9           3
10          4
11            6
12          5
13        5
14      >>> next(path_yielder(t1, 6))
15      [1, 2, 4, 6]
16      >>> path_to_5 = path_yielder(t1, 5)
17      >>> sorted(list(path_to_5))
18      [[1, 2, 5], [1, 5]]
19
20      >>> t2 = Tree(0, [Tree(2, [t1])])
21      >>> print(t2)
22      0
23        2
24          1
25            2
26              3
27              4
28                6
29              5
30            5
31      >>> path_to_2 = path_yielder(t2, 2)
32      >>> sorted(list(path_to_2))
```

```
33          [[0, 2], [0, 2, 1, 2]]
34      """
35
36      "*** YOUR CODE HERE ***"
37
38      for _____ in _____:
39          for _____ in _____:
40
41              "*** YOUR CODE HERE ***"
```

> Hint: If you're having trouble getting started, think about how you'd approach this problem if it wasn't a generator function. What would your recursive calls be? With a generator function, what happens if you make a "recursive call" within its body?

Use Ok to test your code:

```
1   python3 ok -q path_yielder --local
```

Finally, you can run doctest to check your answer again.

```
1   python3 -m doctest hw05.py
```