Homework 5

Due Friday, March 1 at 5:00 PM

"I think we can all agree that junior high is filled with embarrassing and awkward and downright humiliating moments, right?" - Lizzie McGuire

Handing In

To hand in a homework, go to the directory where your work is saved and run `cs0160_handin hwX` where X is the number of the homework. Make sure that your written work is saved as a .pdf file, and any Python problems are completed in the same directory or a subdirectory. You can re-handin any work by running the handin script again. We'll only grade your most recent submission. To install stencil Python files for a homework, run `cs0160_install hwX`. **You will lose points if you do not hand in your written work as a .pdf file.**

Silly Premise

While Lizzie McGuire is riding her Razor scooter to meet up with Miranda and Gordo, she becomes inspired by the trees she sees to organize her vast collection of Silly Bandz a certain way.

1 Written Problems

Problem 5.1

Binary perfection

Lizzie wants to be able to organize her silly bandz in the manner of a *perfect binary tree*, so that she can easily keep track of her immense collection. As she scooters home, she thinks about how she can prove this is an efficient way to organize her silly bandz.

In class, we proved that the total number of *nodes* in a perfect binary tree of height h is $2^{h+1} - 1$. For this problem, you are going to prove that a perfect binary tree has a certain number of *leaves*.

Our recursive definition of a perfect binary tree is a binary tree T where one of the following two properties holds:

1. The root of T is a leaf, meaning the tree only has one node
2. The root of T can be considered to be at height $h + 1$. The root of T has left and right subtrees, and each subtree is a perfect binary trees of height h.

Now, **prove** by induction that a perfect binary tree of height h has 2^h leaves, for all $h \geq 0$.

Note: In your proof, you may only make assumptions based on the definition provided above. Other assumptions, including the ones made in class, will not earn you full credit.

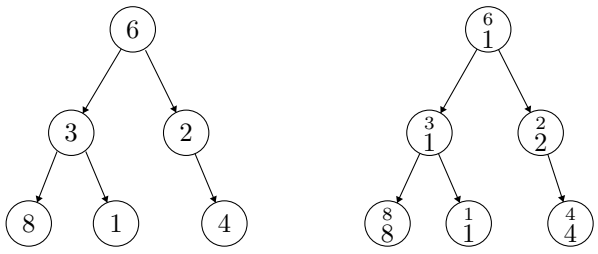
Problem 5.2

Learning from our children

Lizzie realizes that, coincidentally, her pile of silly bandz is such that bandz are weakened when they deviate from their original shape. She assigns each band a weakness value. She also notices that bandz higher on the pile can be damaged by smashing lower bandz. She devises a system to calculate the total weakness of a pile.

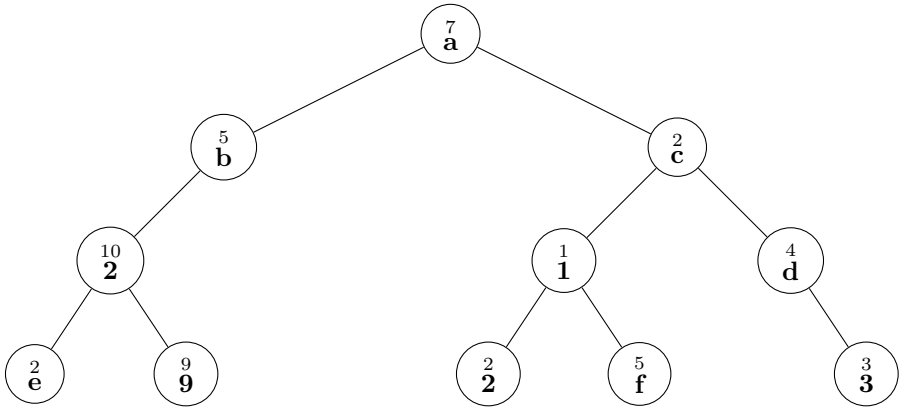
Consider an n-node binary tree T where each node N has an associated integer value, $N.u$. You will design an algorithm that decorates each node in T with another value, $N.z$. The value $N.z$ is the minimum of the u-values among all of N's descendants, where the node N is counted[1] as one of its own descendants. Thus for a leaf (which has only once descendant, itself), $N.z = N.u$. For the root, $N.z$ is the minimum of all the u-values in the whole tree.

A small before-and-after example is shown below. On the left is the original tree. On the right is the decorated tree, with u-values on the top and z-values on the bottom.



(a) The following figure shows a tree in which each node is labeled with its u-value. We've filled in a couple of z-values for you, in bold below the node's u-value. For each letter in the tree below, give the corresponding z-value that would satisfy the above description (e.g., ...**t** = 1,...).

[1] We generally don't count a node as one of its own descendants, but sometimes (as in this problem) it is convenient to do so.



(b) Write pseudocode for an algorithm to **decorate** each node in the tree with its z-value. Assume that each node has variables `u` and `z` that you can access and change as necessary. After it runs, every node's `z` value should have something assigned. Your algorithm should run in $O(n)$ time, where n is the number of nodes in the tree.

2 Python Problems

Problem 5.3

Binary Tree

Using the algorithm Lizzie devised above, she is able to begin organizing her collection of silly bandz. Recognizing the vast quantity of silly bandz she owns, she decides to construct a tree out of them. Help her create a data structure, a linked binary tree, to keep track of her silly bandz.

Your assignment is to implement, in Python, the linked binary tree data structure.

Remember that when doing object-oriented programming in Python, there's an obligatory `self` argument for every function. But this is only when you're defining the method signature! When you *call* the method, the parameter is implicit: you don't need to pass it, Python handles it for you.

Requirements

Functionality

Fill in all of the methods in `bintree.py`. Make sure methods run as efficiently as possible. Remember that the height of an empty tree is undefined and the height of a one-node tree is zero. If the user calls the `height()` method when the height of the tree is undefined, you should throw an exception.

Note: Do not name variables the same name as a method declaration. You will get a TypeError saying that the "object is not callable."

Runtime Requirements

Each of the methods you are writing should run in $O(1)$ worst-case time. We have emphasized this in the stencil where it is quite possible to implement it otherwise.

Testing

As always, you will need to make and hand in your own test cases, but you knew that. You should write all of your test functions in the provided file `bt_test.py`. Don't forget to comment each test so it's clear what you're testing for.

Stencil Code and Hints

As usual, we have provided you with stencil code. To succeed on this problem, and to make the most of our stencil, you may find it helpful to read through the following hints.

The stencil contains 4 classes:

1. `EmptyBinTreeException`: This class defines an exception that you should throw when methods are called on an empty binary tree. The docstrings provided in the stencil should indicate when this exception should be thrown.

2. `InvalidInputException`: This class defines an exception that you should throw when methods are called with invalid parameters. The docstrings provided in the stencil should indicate when this exception should be thrown.

3. `Node`: This class defines a `Node`. Your tree will contain many of these. You are responsible for filling out any method in this clas marked with `TODO`. *It is very important that you read the note below regarding the `self` argument passed to each method in this class.*

4. `BinTree`: This class defines the Linked Binary Tree data structure. You are responsible for filling out any method in this class marked with `TODO`. *It is very important that you read the note below regarding the `self` argument passed to each method in this class.*

An important note about the `self` argument: Remember that when doing object -oriented programming in Python, there's an obligatory `self` argument for every function. This argument can be used to access the internal attributes

of a particular class, since these variables cannot be accessed directly. We do this from within a method by calling `self.<attribute>` (without the angle brackets). Also remember, however, that the `self` argument is only used when *defining* a method. When *calling* the method, Python passes this argument for you, and you should leave it out.

Problem 5.4

Hash Set

Lizzie now decides to sort her silly bandz according to their shapes into boxes where each box contains at most one band. She decides to implement a hashset to help her. Unfortunately, she does not know how to code and is therefore having trouble achieving her goal. She asks you for help.

Implement the core methods of a hash set in Python. Think of a hash set as a simpler version of a hash table; rather than storing (key, value) pairs, the item is simply inserted into the set. In other words, the item is not mapped to anything, it's just hashed and stored. However, it retains the constant-time properties of a traditional hash table. The underlying structure of your set will be an array. **Do not** use Python's built-in sets or dictionaries. You may use only an array in your implementation.

There are two parts to this problem. First, you will write a hash function. This function will take a three-character key and transform it into a single integer value. (More on this below.) This integer value will be used as an index, and you will store the item to be hashed in an array at the calculated index.

Requirements

Implement the hash function, the methods specified below, and turn in the tests you believe necessary to verify that **all** methods described below are fully functional. Note that you won't be evaluated on the *number* of tests you turn in, but rather on how well your tests demonstrate that you have thought about what could go wrong with your hash set.

Remember: Check for invalid inputs. You must throw `InvalidInputException`s where appropriate. This is specified in the stencil code.

Methods

The stencil code for this project is located in `hashset.py`. You will need to implement the following methods, being sure to follow the runtime requirements specified in the stencil code:

- `__init__(self, expected_size=256, key_length=3)`

  - This is partially filled in and handles initializing your hash set to the smallest prime greater than or equal to `expected_size` and setting up a variable to track the size of your hashset.
  - Note that we've defined this method with default parameters. This allows the caller to instantiate your hash set simply by calling `HashSet()` or by including the optional arguments: `HashSet(500)`, `HashSet(500, 4)`, or even `HashSet(key_length=4)`. Make sure that you are checking the validity of your inputs before you use them!
  - You may need to add additional variables that will be used for your hash function, as described below.

- `my_hash(self, key)`
  - Takes a key and returns an index into the array. All of the keys must have the same length. The default is 3 ASCII[2] characters.
  - Your hash function will use universal hashing (discussed in class), and you can find an explanation of the specifics in `hashset.py`.
  - To convert a character to its numerical value (which you'll need to do for hashing), use the Python `ord()` function. For example, `ord('c')` returns 100.

- `insert(self, key)`
  - Inserts `key` into the set. If `key` is already present in the set, it is ignored and not added a second time.
  - To deal with collisions in your hash set, you should have a bucket at each entry in your array to ensure that if two different keys have the same hash, neither one of them is accidentally removed.

- `contains(self, key)`
  - Returns `True` if `key` is present in the set and `False` otherwise.

- `remove(self, key)`
  - Removes `key` from the set and returns it. If the key is not present in the set, then `None` is returned.

Implemented Methods

The following methods have been implemented for you, based on the instance variables created in `__init__`. Make sure that you do not remove or change the names of these variables without making appropriate changes in the methods outlined below. You **should** write tests for these methods in your `hashset_test.py` file, as they interact closely with the methods described above.

[2]This means that the characters are represented by 8-bit numbers, so for some character c, $ord(c)$ will be between 0 and 255.

- `get_keys(self)`
  - Returns a list containing all the keys in the hash set.
- `size(self)`
  - Returns the number of items in the hash set.
- `is_empty(self)`
  - Returns `True` if there are no items in the hash set, and `False` otherwise.
- `clear(self)`
  - Removes all items from the hash set.

Details

In your implementation, you'll be storing strings of k digits. As described above, you will use the `ord` function to generate a value for each digit. `ord` converts a letter to a number base 256. Since the set size, n, is larger than the "base" in which the digits are represented, we can simply use the `ord` values as digits. We then pick k random numbers between 0 and n, the size of the set, form the sum $a_1x_1 + \ldots a_kx_k \bmod n$, and we're on our way.

This does require, however, that the set size be larger than the biggest value returned by `ord`, which is why we required that your set will never have a capacity less than 256. If you *do* build a set with capacity smaller than 256, the universal hashing property proved in class no longer necessarily holds.

Testing

As always, testing is part of your grade! You should write all of your test functions in the provided file `hashset_test.py`. In addition to testing the basic functionality of your hash set, you should also consider these special cases:

- Inserting something with a duplicate key.
- Finding/removing keys not in the set.
- Two keys that have the same hash. This can cause problems if your `contains` or `remove` methods are not correctly implemented. To test this you can change the capacity of your set to ensure collisions.

There may be more cases you should test, so don't assume that your hash set works just because you pass the above tests. Make sure you think of all the different cases that need to be tested.