Homework 5
Due Friday, March 3 at 3:00 PM

“You underestimate how much I like a challenge.” - Annalise Keating

Handing In
To hand in a homework, go to the directory where your work is saved and run

\texttt{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 \texttt{cs0160\_install\ hwX}. Please leave room between questions and in the margins on your pdf so that your grader can leave feedback on your work. \textbf{You will lose points if you do not hand in your written work as a .pdf file.}

Silly Premise
We’re almost at the halfway point, and all these dinners and movies have been a bit overwhelming. It’s time to take a breather, throw on some PJs, and enjoy a little bit of Netflix and Chill time. So grab your legal pads and open a bottle of red wine (if legal), it’s time to teach you a little lesson on \textit{How to Get Away with Murder}!

1 Written Problems

Problem 5.1

Binary perfection
Michaela is freaking out about losing her engagement ring while burying a body in the forest, so to calm herself down she decides to start counting leaves on the nearby \textit{perfect binary trees}. After a while, she starts to notice a pattern...

In class, we proved that the total number of nodes in a perfect binary tree of height \(h\) is \(2^{h+1} - 1\).

Our recursive definition of a perfect binary tree is a binary tree \(T\) where either:

1. The root of \(T\) is a leaf, and hence the tree has only one node, or

2. The left and right subtrees of the root of \(T\), where \(T\) is height \(h + 1\), are both 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:** The only assumptions you can make about a perfect binary tree in this proof come from the above definition. Other assumptions, including the ones made in class, will not earn you full credit.

**Problem 5.2**

**Learning from our children**

After another stunning victory in the courtroom, Annalise decides to celebrate by thinking of interesting binary tree questions to ask CS16 students.

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 for all of \( N \)'s descendants, where the node \( N \) is counted as one of its own descendants. Thus for a leaf (which has only one descendant, itself), \( N.z = N.u \). For the root, \( N.z \) is the minimum of all the \( u \)-values in the whole tree.

(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., \( t = 1, \ldots \)).

(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

---

\(^1\) 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.
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

Concerned about her online image, Brown grad Laurel takes a break from the stress of law school to Google herself. Upon finding that her name also describes an evergreen tree native to the Mediterranean with glossy, aromatic leaves, she is inspired to implement a data structure to honor her namesake and calls you, her go-to computer-savvy friend.

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.

Remember: Check for invalid inputs. You must throw InvalidInputExceptions and other exceptions where appropriate. This is specified in the stencil code.
Problem 5.4
Hash Set
Tired of being used for his coding abilities, Oliver tragically breaks up with Connor. To cope with the breakup, Connor decides to implement a Hash Set. Unfortunately, Connor does not know how to code and is therefore having trouble achieving his goal. Luckily, he sees you in the Sun Lab and knows that you are talented enough to 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 key is treated as both the key and the value. The underlying structure of your set will be an array. Do not use Sets (Python’s built-in support for hash sets) or dictionaries (Python’s built-in support for hash tables). You may use only an array in your implementation.

There are two parts to this problem. First, you will write a hash function. Then you will use your function to calculate an index in your array from a key, enabling you to map a key to a location in the array.

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 InvalidInputExceptions 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 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.

- `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)`

---

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.
- Returns \textbf{True} if there are no items in the hash set, and \textbf{False} otherwise.

\begin{itemize}
  \item \texttt{clear(self)}
  \end{itemize}

- Removes all items from the hash set.

\section*{Details}

In your implementation, you’ll be storing strings of \( k \) digits. As described above, you will use the \texttt{ord} function to generate a value for each digit. \texttt{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 \texttt{ord} values as digits. We then pick \( k \) random numbers between 0 and \( n \), the size of the set, form the sum \( a_1x_1 + \ldots a_kx_k \mod n \), and we’re on our way.

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

\section*{Testing}

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

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

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.