Problem 1

Suppose that we represent a BST with a BSTNode class. (We’ll omit any BST “wrapper” class.) As normal, the BSTNode class will have left, val, right fields (we’ll assume they are all public, for simplicity). However, let’s add two new fields, height, count – which are the height and count of the subtree rooted at that node.

Since we can always figure out the height and count from the BST without these fields, these are cached values; as we’ve talked about in class, these are dangerous (you might forget to keep them up-to-date), but they can help performance if you need to check them often.

On the next page, I have several functions for you to fill in; for insert(), assume that it uses the \texttt{x=change(x)} style; that is, it takes a root parameter (which might be \texttt{None}), and returns the new root (which might be different). Make sure that you update the height and count on each call to insert().

Some rules:

- All functions must work properly if sent a \texttt{None} parameter (that is, an empty tree)
- \texttt{bst\_height()} and \texttt{bst\_count()} must run in $O(1)$ time
- \texttt{bst\_insert()} must run in $O(H)$ time, where $H$ is the height of the tree
  - This means you \textbf{cannot} recurse through the entire tree!
  - Make sure to argue why you can \textbf{correctly update the height and count} without recursing through the entire tree.
def bst_height(root):

def bst_count(root):

def bst_insert(root, new_val):
    """Inserts into the BST, and returns the root of the new tree. If new_val is a duplicate, this is a NOP; it returns the original tree."""

Don't forget to answer:

**Why does your implementation of bst_insert() - which runs in O(H) time - generate the correct height and count values, even though it doesn't recurse through the entire tree?**
Problem 2

This one’s a simple utility function: build a perfectly balanced (as close as you can) BST, given a (sorted) list of values as input. First find the middle element in the list, and then split off the left and right pieces; use recursion to build sub-trees for the left and right sides.

For version 1, it’s OK to use slicing - even though that incurs a lot of cost because it copies the value.

For version 2, add some default parameters to the function so that you can pass the original list along (unmodified), with no need for slicing or copying.

Do version one on this page; I’ve put version 2 on the next page.

**VERSION 1 - SLICING OK**

```python
def bst_from_list(root):
```
# note that I’ve left space for default args - but you
# have to choose what they will be!

def bst_from_list(root, }:
Problem 3

Python dictionaries are implemented using a data structure known as a hash table, which you will study in a later class. Hash tables are great for looking up values (it takes $O(1)$ time to set or remove any item) but they make it expensive to perform in-order traversals of the keys. (We don't have time to learn about them now.)

Implement (on the next page) a BST which stores (key,value) pairs like a dictionary. The key will be used to determine where in the tree you will store it; the value is extra data, ignored by the BST - except that the BST will store it in the correct node. (This is known as satellite data.)

I have provided the __init__ function for a BSTDictNode class; notice the fields that it creates in each BSTDictNode object. (As we've been doing in Section, we'll assume that the fields are all public, for easy access.)

You will write two functions. bst_dict_set() is like insert() in an ordinary BST; it will search for the correct location and create a new node. It will perform insertion in the x=change(x) style, like we've been using - but notice that it has two parameters (the key, and the value). Use the key to determine where the data should be stored - ignore the value entirely, until you have found the right location.

If a user calls bst_dict_set() but the key already exists (a duplicate key), then modify the node, to set it to the new value.

In other words, the code

```python
bst_dict_set(root, 10, “asdf”)
```

should work, more or less, like:

```python
root[10] = “asdf”
```

(if root was an ordinary dictionary).

You will also implement bst_dict_get(), which performs a search through the dictionary. This function (like search() in a BST) takes a key as its parameter. If you find a node matching the key, then return the value stored at that node. If you do not find the key, then raise a KeyError.
class BSTDictNode:
    def __init__(self, key, value):
        self.left = None
        self.key = key
        self.value = value
        self.right = None

def bst_dict_set(root, key, value):

def bst_dict_get(root, key):