Preface

Most of this assignment will continue with the binary search trees from the last assignment. Code is provided to you in hw3.py. This is similar to hw2.py, but also includes correct versions of the methods and functions that you added for in homework 2. You must use this code, rather than your own code from homework 2.

Parameter Passing in Python

Question 1: Fun with Lists

For your binary search tree, create a method for Nodes called CreateList. This should not take any arguments other than the self argument. It should return a list with each node’s key in ascending order (the order that you get from walking the tree).

Create a second version of this method, CreateListX, that takes a parameter, which is the list built so far. CreateListX will add each node’s key in ascending order onto the list. This routine will not pass the list back with the return statement. Instead, it will modify the list that was passed in. When it is initially called, it should be passed with a variable whose value is an empty list. After this routine is finished, the value of the variable should contain the entire list.

For CreateListX, you could have made your routine also pass back the list via return. Explain why that might be considered bad style.

Hand in the code for both methods.

Question 2: Passing Arguments

In the second method you created, you made use of how python passes arguments to functions and methods. It certainly does not use call-by-value, as many programming languages use. What is the name for how python passes arguments (feel free to google the answer).

Explain how the following code works. Explain how this illustrates python’s approach for passing arguments, even though no function or method is called.

```python
a = [1,2]
b = a
b.append(3)
print(a)
b = [4]
print(a)
```

Question 3: Permanence of Objects

Examine the code for BuildTrees in hw3.py. The routine calls BuildTree1023 to build 1000 trees. What happens to these trees? Explain how they are removed. Feel free to use any resources on the web for answering this question.
Question 4: Measuring the Effect

We will now determine how the height of a tree changes as inserts and deletes are made to it. Does repeatedly inserting and deleting nodes cause the average height of the tree to grow (get worse) and shrink (improve)?

Create a function called ChangeTree with a single parameter called num. The function should do the following.

1. Call BuildTree1023 to build a tree with 1023 nodes with the keys from 1 to 1023.
2. Do num times:
   (a) Randomly pick a number between 1 and 1023, find it in the tree, and then delete it.
   (b) Insert a new node with the same key that you just deleted back into the tree.

Have the routine output 2 values (as an unmutable list, or tuple): the height of the tree after it is initially built, and after the deletion-insertions.

Hand in your code. Run ChangeTree with the argument of 1000. How does the size of the tree compare before and after the 1000 deletes and inserts?

Why might this test not be very insightful? Hint: see the next question.

Question 5: Average Behavior

Similar to what we did in homework 2, we will create a function called Run500 that will run ChangeTree(1000) 500 times and compute the average size of the tree before the 1000 inserts and deletes, and compute the average size of the trees after the 1000 inserts and deletes.

Hand in a copy of your code, and report the average before and the average after inserting and deleting 1000 times. After inserting and deleting 1000 times, does the average height of the trees degrade or improve?

Question 6: Alternate Version of Delete

In Figure 12.4 of the textbook, the re-arrangement of the tree is quite complex. There is a simpler version. Referring to Figure 12.4 case d where we want to remove node z, and z’s parent is q, and z’s left child is l and right child is r. We could instead move r into z’s and find the minimum node under r, and make l it’s left child (see class slides).

Give the code for this version of delete (call it DeleteX). This is tricky code, so make sure you try out your delete code on a case that exemplifies case d, and print out your answer using str to make sure your delete is correct. You do not need to hand in your test case.

Hand in the code for DeleteX.

Question 7: Alternate as Good as the Original?

You will now determine whether DeleteX is better or worse than Delete.

Create a new function, called ChangeTreeX. It will be similar to ChangeTree except that it will take a second parameter, called useDeleteX which will have a default value of False, which will control which version of delete is used. If you do not know how to use binary variables or default parameter values, look this up on the web.

Also, we will be running ChangeTreeX for a lot more than 1000 iterations to see the effects of a lot of inserts and deletes. So, after every 2000 iterations, have ChangeTreeX print the iteration number (e.g., 2000, 4000, 6000, etc) and the current height of the tree.
Unlike the previous questions, we will not average the result over a lot of runs. Instead, we are doing a single run but for much longer. Let’s do 1,000,000 inserts and deletes on a single tree.

For the original version of Delete, what is happening to the height of the tree for large numbers of inserts and deletes? Does it stabilize at all, and if so, around what value? Is this the same value that you found that you got after 1000 inserts and deletes? How does this compare to the best case and worst case height of a binary search tree with 1023 nodes?

What happens when you run this with DeleteX? Does it stabilize, and if so, around what value? How does this compare to the best case and worst case height of a binary search tree with 1023 nodes?

**Question 8: Runtime**

Determine how long the two versions take to run. There is python module timeit that you should use.

```python
from timeit import default_timer as timer

start = timer()
# do stuff
end = timer()
```

Do you see the difference in execution time that the DeleteX causes? By what factor is it different? Explain how that factor relates to the difference in the height of the trees produced by the two versions of delete.

**Question 9: Why DeleteX is so Much Worse**

Explain why DeleteX is behaving so much worse than Delete? For case d in Figure 12.4, how does the height of the trees that Delete and DeleteX create compare to the original height of the trees (before the node was deleted)?

**Prelude to String Alignment**

**Question 10: Returning the longest path**

This question builds on PrintAllPaths from the previous homework. Make FindLongestPath will traverse the tree. For PrintAllPaths, each time it found a leaf node, it prints out the path. For FindLongestPath, each time if finds a leaf node, it should compare the length of the path that it currently thinks is best to the depth of the current node.

As input, it should take 3 parameters. Since this is a Node method, the first parameter is self. The second parameter should be the node’s ancestors and the third parameter should be the longest path found so far. It should not alter the second or third arguments, but return the longest path found as its return value.