CS532: Homework 3 (version 1)
Spencer Hann

This is version 0 of the homework. I will be adding one extra question on dynamic programming.
Version 1: I rewrote some of the questions to try to make them more understandable. The coding part of question 4 is now moved entirely to question 5.

Question 1: Preface

To facilitate automated testing of your script, do not have any code in your script that runs automatically.
Put any testing code in a function called main. To run main automatically when you are testing your script, include the following lines at the very end of your code.

```
if __name__ == "__main__":
    # execute only if run as a script
    main()
```

Also, make sure you have all of your code in a single python file. 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. Note that there was a bug in the delete method from hw2.py. The 6th line of the method was if y.parent is not None: whereas it should be if y.parent != z:. This is corrected in hw3.py.

Question 2: Fun with Lists

Create a method for Nodes called CreateList. This should not take any arguments other than the self argument. This should return a list with each node's key in ascending order. In creating this method, you will need to be careful for when you are appending to the list (adding a single element) versus concatenating a list onto another list.
Create a second version of this method, CreateListX, that takes a parameter, which is the list built so far. 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. Note that this version does not return any value, it just manipulates the list passed to it. Hand in the code for both methods.

```
def CreateList(self):
     l = []
     if self.left:
         l = self.left.CreateList()
     l.append(self.key)
     if self.right:
         l.extend(self.right.CreateList())
     return l

 def CreateListX(self, l):
     if self.left:
         self.left.CreateListX(l)
     l.append(self.key)
     if self.right:
         self.right.CreateListX(l)
```

Question 3: 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.

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

Python uses "pass by Object." On line 1, a new list is initialized and "a" is bound to it. One line 2, "b" is bound to the same object as "a." On line 3, the object to which "b" refers (which is the same object to which "a" refers) has a 3 appended to it. On line 4, the object that "a" refers to is printed. On line 5, "b" is bound to a new object. On line 6, the object that "a" refers to is printed again.

Question 4: Insert and Deletes

In the previous homework, we built a binary search tree with 1023 unique keys from 1 to 1023. If we delete a node with a certain key, and then insert a node with that exact same key, explain how the structure of the binary search tree (its nodes and leaves) might change.

Mostly the height of the tree will remain the same. If a leaf is deleted and re-inserted it will be put back where it started. If a node with two children is deleted, its successor will replace it in the tree. Moving the successor will move up its own sub-tree, but when re-inserting the original node will most likely go to the left sub tree. If the successor subtree did not contain the longest path, but the removed node goes down the longest path on the left when being re-inserted, this can cause the tree to gain a level. If a node with only one child is deleted, its entire subtree moves up a level. When the node is re-inserted, it may be added to the bottom level, but it also might find a “hole” or an open spot before the bottom level, where it can become a new leaf without adding to the height of the sub-tree. If this happens in a sub-tree that contains the longest path for the whole tree the the height of the whole tree will have been reduced by 1.

Question 5: Insert and Deletes: Part b

This question builds off of the last question in the previous homework. Modify BuildTree1023 so that it does the following:

1. Build a tree with 1023 nodes with the keys from 1 to 1023 (just as in the previous homework).
2. Do 1000 times:
    (a) Randomly pick a number between 1 and 1023, find it in the tree, an d 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 immutable list): the height of the tree after it is initially built, and after the 1000 deletion-insertions.
The BuildTrees routine from the previous homework runs BuildTree1023 1000 times. Change it so that it computes the average height after the initial tree is built, and after the 1000 deletion-insertions.
Hand in a copy of your code, and hand in the average before and after. After inserting and deleting 1000 times, does the average height of the trees degrade (get longer)?

The difference is subtle, but on average, the heights of the trees are reduced by about 1 (from slightly over 22, to just under 21).

```
def BuildTree1023():
    l = [x for x in range(1,1024)]
    t = Tree()
    while l != []:
        i = int(random.random()*len(l))
        t.Insert(Node(l[i]))
        del l[i]
    orig_height = t.root.Height()
    for i in range(1000):
        x = random.randint(1,1023)
        t.Delete(t.root.Search(x))
        t.Insert(Node(x))
    return orig_height, t.root.Height()

def BuildTrees():
    sum1, sum2 = 0, 0
    for i in range(1000):
        a, b = BuildTree1023()
        sum1 += a
        sum2 += b
        #print(sum1, sum2)
    print("original height: " + str(sum1/1000))
    print("post delete height: " + str(sum2/1000))
```

Output:

```
    spencer@spencer-rBS:~/OHSU/532/hw3$ python3 hw3.py
    original height: 22.064
    post delete height: 20.915
```

Question 6: Alternate Version of Delete

In Figure 12.4 of the textbook, the re-arrangement of the tree is quite complex. A simpler version for case d is to move z's left child to be y's left child (it was none) and to replace z with r. 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.

```
    def DeleteX(self,z):
    if z.left is None:
          self.Transplant(z,z.right)
     elif z.right is None:
          self.Transplant(z,z.left)
     else:
          y = z.right.Min()
          y.left = z.left
          z.left.parent = y
          self.Transplant(z,z.right)
```

Question 7: Alternate Version of Delete: Part b

With this new version of DeleteX, create a version of BuildTree1023 (called BuildTreeX) that after building a tree with 1023 nodes, randomly picks a key, and deletes it from the tree, and then inserts it back it. Rather than doing this for a 1000 times, do this for 100,000 times. After every 1000 iterations of deleting and inserting, print out the iteration number (e.g., 1000, 2000, 3000, etc) and the current height of the tree.
Make BuildTreeX take a single parameter that controls if it runs the original version of Delete or the altered version, DeleteX.
Note, unlike question 5, you will not be averaging your results over 1000 differ runs. Instead, we are doing a single run but for much longer.

```
def BuildTreeX(use_DeleteX=False):
     l = [x for x in range(1,1024)]
     t = Tree()
     while l != []:
          i = int(random.random()*len(l))
          t.Insert(Node(l[i]))
          del l[i]
     orig_height = t.root.Height()

     if use_DeleteX: # determine which function will be used
          delete = t.DeleteX
     else:
          delete = t.Delete

     for i in range(100000):
          If i % 1000 == 0:
               print("Iteration " + str(i),
                     ", Height: " + str(t.root.Height()))
          k = random.randint(1,1023)
          delete(t.root.Search(k))
```

```
        t.Insert(Node(k))

    post_height = t.root.Height()
    print("original height: " + str(orig_height))
    print("post deletion height: " + str(post_height))
    return orig_height, post_height
```

For the original version of Delete, what is happening to the height of the tree? 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?

The height of the tree seems to fluctuate around 17-21, depending on the test. This is very good compared to the worst case of 1023.

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?

The height rises rapidly at first, and then seems to plateau in the upper 300s. Result vary all the way from 360 to 400. Obviously this is still better than the worst case height, but also quite bad compared to the other deletion algorithm

Also comment on how long your code takes to run. Do you see the difference in execution time that the DeleteX causes?

On my laptop, the DeleteX test runs in about 7.6 seconds and the normal Delete function runs in about 0.9 seconds. This is definitely a non-trivial difference.

Question 8: Alternate Version of Delete: Part c

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)?

With Delete, if the subtree that holds the deleted node’s successor is on the longest path through the tree then the trees height will be decreased when that subtree is moved up. If it is not on the longest path the the overall height stays the same. However, with DeleteX, if the z’s left subtree had a greater height than its right, then when it is moved down in the tree, the tree’s overall height increases. It otherwise stays the same. So Delete will sometimes decrease the height of the tree, whereas DeleteX with regularly increase the height of the tree.