CS532 Homework 2
Archana Machireddy

Question 2

```
def CreateList(self):
    keys = []
    if self.left is not None:
        keys += self.left.CreateList()
    keys.append((self.key))
    if self.right is not None:
        keys += self.right.CreateList()
    return keys

def CreateListX(self,x):
    if self.left is not None:
        self.left.CreateListX(x)
    x.append((self.key))
    if self.right is not None:
        self.right.CreateListX(x)
```

Question 3

Python uses call-by-object or call-by-object-reference while passing arguments to functions and methods. From an online blog (https://jeffknupp.com/blog/2012/11/13/is-python-callbyvalue-or-callbyreference-neither/):

“In Python, (almost) everything is an object. What we commonly refer to as "variables" in Python are more properly called names. Likewise, "assignment" is really the binding of a name to an object. Each binding has a scope that defines its visibility, usually the block in which the name originates.”

In python the object reference i.e the name it is bound to, is passed as the function parameter. When immutable arguments like integers, strings and tuples are passed, it acts like call-by-value. They are not changed by the function, as they are immutable. But when as mutable object like lists are passed, they can be changed inside the function. As only the name bound to the list object is passed into the function, changes made within the function change the list object. If a new list is assigned to the name within the function, the list in the caller’s scope does not change.

```
a = [1,2]
```

```
b = a
b.append(3)
print(a)
b = [4]
print(a)
```

Line 1: Creates a binding between name ‘a’ and a list object [1,2]

Line 2: A new name ‘b’ is created and assigned to name ‘a’. Assignment between names does not create a new object. Now both values ‘a’ and ‘b’ point to the same list object [1,2]. It does not create a copy.

Line 3: Append method is called on the list object bound to name ‘b’. This appends 3 to the list object making it [1,2,3].

Line 4: As there is only one list to which both names ‘a’ and ‘b’ are bound to, changes made in the list will be visible through both names ‘a’ and ‘b’. This line prints the list : [1,2,3]

Line 5: The name ‘b’ is now bound to a new list object [4]. This removes its previous binding to list object [1,2,3]

Line 6: Name ‘a’ is still bound to list object [1,2,3]. So it prints list : [1,2,3]

Question 4

During deletion the successor replaces the deleted node. The successor will the minimum in the sub-tree of the node to be deleted. So during deletion operation a leaf node or a node at lower levels gets moved up. During insertion, the node always gets inserted as a leaf. So, the node, which was high up in the tree before deletion, will end up at the lower levels of the tree as a leaf. Repeated insertions and deletions can change the structure of the binary search tree quite a bit.

Question 5

```
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]
    h1 = t.root.Height()
    for i in range(1000):
```

```
        x = random.randint(1,1023)
        t.Delete(t.root.Search(x))
        t.Insert(Node(x))
    h2 = t.root.Height()
    return (h1,h2)

def BuildTrees():
    sum1 = 0
    sum2 = 0
    for i in range(1000):
        (a,b) = BuildTree1023()
        sum1 = sum1 + a
        sum2 = sum2 + b
#        print(sum)
    print(sum1/1000, sum2/1000)
```

The average height before was 22 and after was 21. The average height of the tree gets shorter after deleting and inserting 1000 times.

Question 6

```
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()
        if y.parent == z:
            self.Transplant(z,y)
            y.left = z.left
            y.left.parent = y
        else:
            y.left = z.left
            y.left.parent = y
            self.Transplant(z,z.right)
```

```
Archanas-MBP:HW3 archana$ python hw3.py
((3)5(10))15((18(19))20(25))
Delete node 15 using Delete
((3)5(10))18((19)20(25))
Archanas-MBP:HW3 archana$ python hw3.py
((3)5(10))15((18(19))20(25))
Delete node 15 using DeleteX
(((3)5(10))18(19))20(25)
Archanas-MBP:HW3 archana$
```

Question 7

```
def BuildTreeX(ver):
    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]
    h1 = t.root.Height()
    for i in range(100000):
        x = random.randint(1,1023)
        if ver == 0:
            t.Delete(t.root.Search(x))
        else:
            t.DeleteX(t.root.Search(x))
        t.Insert(Node(x))
        if i >0 and i % 1000 == 0:
            print(i,t.root.Height())
```

The height of tree initially before deletions and insertions is about 22. For the original version of Delete, the height of the tree decreases and fluctuates at a value between 17 and 22 after 100000 iterations on different runs. But the value is almost always less than the initial height of the tree. It is closer to best-case height (10), than the originally created tree. It is still much better than the worst-case scenario where the tree would have been 1023 long.

With DeleteX the size of the tree increases. It ends up with a height between 360 and 400 in different runs. This is about 40 times the best case. But it is still better than the worst-case 1023 long tree.

100000 iteration with Delete take about 1.7 seconds, while with DeleteX take 12.4 seconds. This is because during deletion and insertion the time taken to traverse the tree to search for a key, increases as the height of the tree is getting larger and larger.

Question 8

For this discussion let the node to be deleted be node z. In Delete, the successor of the node z replaces it. This reduces the size of the tree by 1, as the successor takes the place of node z, and the successor.right goes into the position of successor.

But in DeleteX, instead of the successor, we are replacing the node z with z.right, and moving z.left as the left child to the successor of node z. The successor to node z will be the minimum value in its subtree, and will be located in the lower levels of the tree, nearer to the leaves if not a leaf. If the tree is pretty balanced (which seems to be the case in our tree as the height of the tree is just twice the optimal), deleting a node which is high in the tree would result in its left branch being moved as the left child to a node in the lower levels of the right sub-tree. This would almost double the height of the tree. This is why the height increases while using DeleteX.

In figure 12.4, if the height of the original tree was h, the height of the tree after Delete is h-1. While using DeleteX, if height of z.left is equal to or lesser than the height of y.right, the height of the tree after DeleteX will be h-1. If height of z.left is k more than y.right, then the height of the tree after using DeleteX will be h+k.