CS 260 Lab 8
Binary search tree removals

Imagine we have a Python program that reads a list of values, inserts each value into a binary search tree, and then removes each value from the binary search tree. After each insert and remove operation, the program also invokes a method that displays a parenthesized in-order traversal of the tree. The parentheses help to show the structure of the binary search tree. Such a program would behave as shown in the following two examples:

Example 1

```
Enter a Python list: [5,3,7,2,4,6,8,0,9,1]
Insert 5: (5)
Insert 3: ((3) 5)
Insert 7: ((3) 5 (7))
Insert 2: (((2) 3) 5 (7))
Insert 4: (((2) 3 (4)) 5 (7))
Insert 6: (((2) 3 (4)) 5 ((6) 7))
Insert 8: (((2) 3 (4)) 5 ((6) 7 (8)))
Insert 0: ((((0) 2) 3 (4)) 5 ((6) 7 (8)))
Insert 9: ((((0) 2) 3 (4)) 5 ((6) 7 (8 (9))))
Insert 1: ((((0 (1)) 2) 3 (4)) 5 ((6) 7 (8 (9))))
Remove 5: ((((0 (1)) 2) 3 (4)) 6 (7 (8 (9))))
Remove 3: ((((0 (1)) 2) 4) 6 (7 (8 (9))))
Remove 7: ((((0 (1)) 2) 4) 6 (8 (9)))
Remove 2: (((0 (1)) 4) 6 (8 (9)))
Remove 4: ((0 (1)) 6 (8 (9)))
Remove 6: ((0 (1)) 8 (9))
Remove 8: ((0 (1)) 9)
Remove 0: ((1) 9)
Remove 9: (1)
Remove 1:
```

Example 2

```
Enter a Python list: [5,7,3,8,6,4,1,10,0,9,2]
Insert 5: (5)
Insert 7: (5 (7))
Insert 3: ((3) 5 (7))
Insert 8: ((3) 5 (7 (8)))
Insert 6: ((3) 5 ((6) 7 (8)))
Insert 4: ((3 (4)) 5 ((6) 7 (8)))
Insert 1: (((1) 3 (4)) 5 ((6) 7 (8)))
Insert 10: (((1) 3 (4)) 5 ((6) 7 (8 (10))))
Insert 0: ((((0) 1) 3 (4)) 5 ((6) 7 (8 (10))))
Insert 9: ((((0) 1) 3 (4)) 5 ((6) 7 (8 ((9) 10))))
Insert 2: ((((0) 1 (2)) 3 (4)) 5 ((6) 7 (8 ((9) 10))))
```

```
Remove 5:
Remove 7:
Remove 3:
Remove 8:
Remove 6:
Remove 4:
Remove 1:
Remove 10:
Remove 0:
Remove 9:
Remove 2:
```

Suppose the BinarySearchTree class contains a method remove(x) that operates as follows. First, remove(x) locates the node p that contains data item x. Next it considers four main cases:

(1) If p is a leaf node, then prune node p from the tree.
(2) If p has a left child but no right child, then replace p by a link directly to p's left child.
(3) If p has a right child but no left child, then replace p by a link directly to p's right child.
(4) If p has two children, then:
    - Locate p's successor node q. (So q is the node that follows p in an in-order traversal.)
    - Swap the data in nodes p and q.
    - Recursively remove x from p's right subtree. (This removes node q using case 1, 2, or 3.)

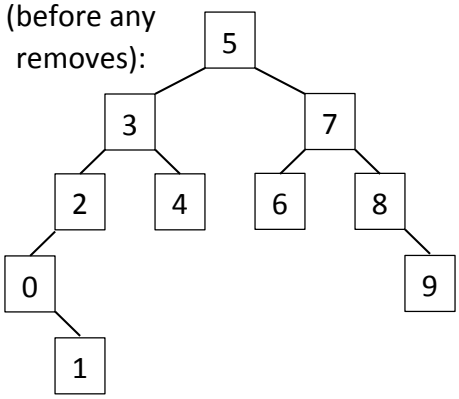
The sequence of remove operations in Example 1 above are illustrated by the sequence of binary search trees that are shown on the subsequent page. First examine these pictures and make sure you understand how each case of the remove operation modifies the binary search tree.

Next, write the following on paper, and turn in your paper by the end of today's lab at 9:50:

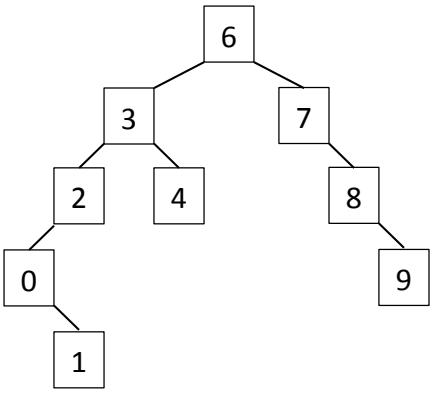
- Draw the sequence of binary search trees that illustrate the remove operations in Example 2 above, using input list `[5,7,3,8,6,4,1,10,0,9,2]`. Use the same format that we've shown for Example 1, and identify the cases that are applied during each remove operation.
- Complete the remaining lines of output to show the parenthesized in-order traversals after each remove operation.

Finally, there is no Python coding required for today's lab, but it is recommended that you should think about how you could implement such a remove(x) function in Python.

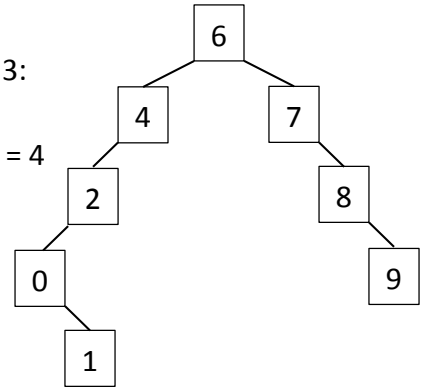
After inserts (before any removes):



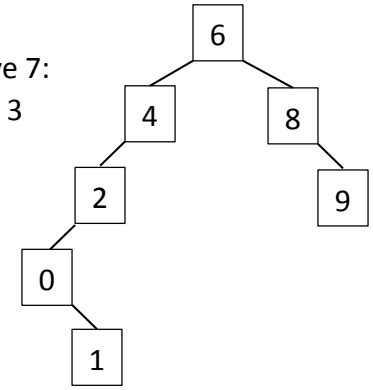
Remove 5:
Case 4
Successor = 6
Case 1



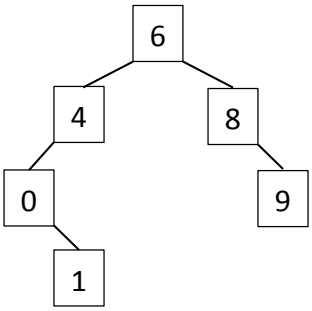
Remove 3:
Case 4
Successor = 4
Case 1



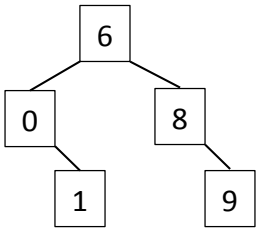
Remove 7:
Case 3



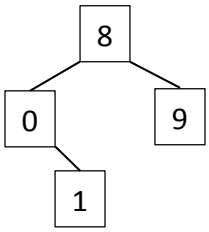
Remove 2:
Case 2



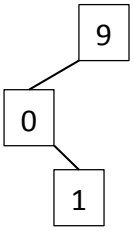
Remove 4:
Case 2



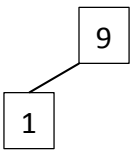
Remove 6:
Case 4
Successor = 8
Case 3



Remove 8:
Case 4
Successor = 9
Case 1



Remove 0:
Case 3



Remove 9:
Case 2

Remove 1:
Case 1