1. Consider the Binary Search Tree (BST):

   ![BST Diagram]

   a. What would need to be done to delete 32 from the BST?

   Set parent's right child to None (30)

   b. What would need to be done to delete 9 from the BST?

   Change parent's left child pointer and 18's parent pointer

   c. What would be the result of deleting 50 from the BST? Hint: One technique when programming is to convert a hard problem into a simpler problem. Deleting a BST node that contains two children is a hard problem. Since we know how to delete a BST node with none or one child, we can convert "deleting a node with two children" problem into a simpler problem by overwriting 50 with another node's value. Which nodes can be used to overwrite 50 and still maintain the BST ordering? 53 - smallest value in right subtree or 47 - largest in left

   d. Which node would the TreeNode's findSuccessor method return for succ if we are deleting 50 from the BST?

   53

2. When the findSuccessor method is called how many children does the self node have?

3. How could we improve the findSuccessor method? Eliminate "dead code" which never runs since self node is known to have two children

4. When the spliceOut method is called from remove how many children could the self node have? 0 or 1 which would be a right child

5. How could we improve the spliceOut method?

   Eliminate "dead code" (see "X"ed out code)
6. The shape of a BST depends on the order in which values are added (and deleted).
   a) What would be the shape of a BST if we start with an empty BST and insert the sequence of values:
      
      70, 90, 80, 5, 30, 110, 95, 40, 100

   b) If a BST contains n nodes and we start searching at the root, what would be the worst-case big-oh \( O() \) notation for a successful search? (Draw the shape of the BST leading to the worst-case search)

7. We could store a BST in an array like we did for a binary heap, e.g. root at index 1, node at index \( i \) having left child at index \( 2 \times i \), and right child at index \( 2 \times i + 1 \).
   a) Draw the above BST (after inserting: 70, 90, 80, 5, 30, 110, 95, 40, 100) stored in an array (leave blank unused slots)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
<th>21</th>
</tr>
</thead>
<tbody>
<tr>
<td>70</td>
<td>90</td>
<td>80</td>
<td>5</td>
<td>30</td>
<td>110</td>
<td>95</td>
<td>40</td>
<td>95</td>
<td>100</td>
<td>190</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

   b) What would be the worst-case storage needed for a BST with \( n \) nodes?

8. a) If a BST contains \( n \) nodes, draw the shape of the BST leading to best, successful search in the worst case.

   

   b) What is the worst-case big-oh \( O() \) notation for a successful search in this "best" shape BST?
2. More partial TreeNode class and partial BinarySearchTree class.

class BinarySearchTree:
...

def delete(self, key):
    if self.size > 1:
        nodeToRemove = self._get(key, self.root)
        if nodeToRemove is not None:
            self.remove(nodeToRemove)
            self.size -= 1
        else:
            raise KeyError('Error, key not in tree')
    elif self.size == 1 and self.root.key == key:
        self.root = None
        self.size -= 1
    else:
        raise KeyError('Error, key not in tree')

def _delitem_(self, key):
    self.delete(key)

def remove(self, currentNode):
    if currentNode.isLeaf(): # leaf
        if currentNode == currentNode.parent.leftChild:
            currentNode.parent.leftChild = None
        else:
            currentNode.parent.rightChild = None
    elif currentNode.hasBothChildren(): # interior
        succ = currentNode.findSuccessor()
        succ.spliceOut()
        currentNode.key = succ.key
        currentNode.payload = succ.payload

    else: # this node has one child
        if currentNode.hasLeftChild():
            if currentNode.leftChild.isLeftChild():
                currentNode.leftChild.parent = currentNode.parent
                currentNode.parent.leftChild = currentNode.leftChild
            elif currentNode.leftChild.isRightChild():
                currentNode.leftChild.parent = currentNode.parent
                currentNode.parent.rightChild = currentNode.leftChild
            else:
                currentNode.replaceNodeData(currentNode.leftChild.key,
                                             currentNode.leftChild.payload,
                                             currentNode.leftChild.leftChild,
                                             currentNode.leftChild.rightChild)
        else:
            if currentNode.rightChild.isLeftChild():
                currentNode.rightChild.parent = currentNode.parent
                currentNode.parent.leftChild = currentNode.rightChild
            elif currentNode.rightChild.isRightChild():
                currentNode.rightChild.parent = currentNode.parent
                currentNode.parent.rightChild = currentNode.rightChild
            else:
                currentNode.replaceNodeData(currentNode.rightChild.key,
                                             currentNode.rightChild.payload,
                                             currentNode.rightChild.leftChild,
                                             currentNode.rightChild.rightChild)
3. Yet even more partial TreeNode class and partial BinarySearchTree class.

```python
class TreeNode:
    ...
    def findSuccessor(self):
        if self.hasRightChild():
            succ = self.rightChild.findMin()
        else:
            if self.parent:
                if self.isLeftChild():
                    succ = self.parent
                else:
                    self.parent.rightChild = None
                    succ = self.parent.findSuccessor()
                    self.parent.rightChild = self
        return succ

def findMin(self):
    current = self
    while current.hasLeftChild():
        current = current.leftChild
    return current

def spliceOut(self):
    if self.isLeaf():
        if self.isLeftChild():
            self.parent.leftChild = None
        else:
            self.parent.rightChild = None
    elif self.hasAnyChildren():
        if self.isLeftChild():
            self.parent.leftChild = self.leftChild
        else:
            self.parent.rightChild = self.leftChild
            self.leftChild.parent = self.parent
            if self.isLeftChild():
                self.parent.leftChild = self.rightChild
            else:
                self.parent.rightChild = self.rightChild
                self.rightChild.parent = self.parent
```
Lecture 23 - AVL tree

BST that is height balanced

balance factor = \frac{\text{height of left sub-tree}}{\text{height of right sub-tree}} - 1

70, 90, 80, 5, 30, 110, 95, 40, 100

1.44 x log₂ n