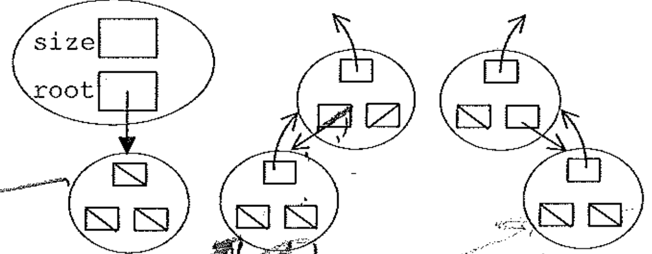
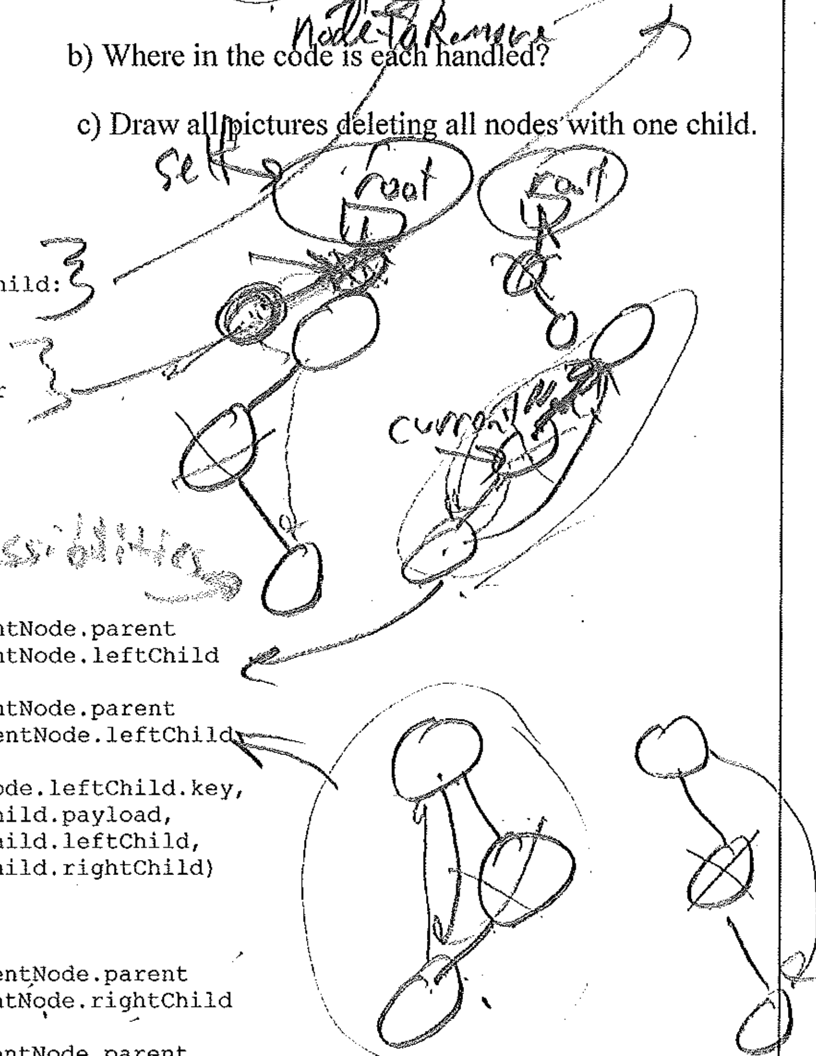
2. More partial TreeNode class and partial BinarySearchTree class.

a) Update picture where we delete a leaf.



b) Where in the code is each handled?

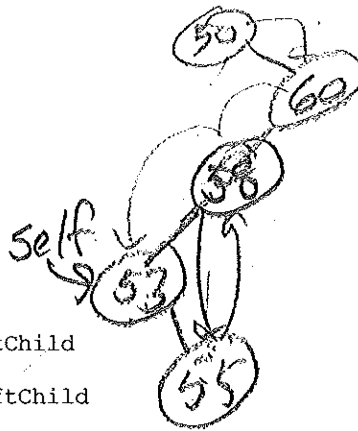
c) Draw all pictures deleting all nodes with one child.



```
class BinarySearchTree:
   . . .
  def delete(self,key):
    if self.size > 1:
        nodeToRemove = self._get(key,self.root)
        if nodeToRemove:
            self.remove(nodeToRemove)
            self.size = 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 = 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.isLeftChild():
            currentNode.leftChild.parent = currentNode.parent
            currentNode.parent.leftChild = currentNode.leftChild
        elif currentNode.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.isLeftChild():
            currentNode.rightChild.parent = currentNode.parent
            currentNode.parent.leftChild = currentNode.rightChild
        elif currentNode.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.

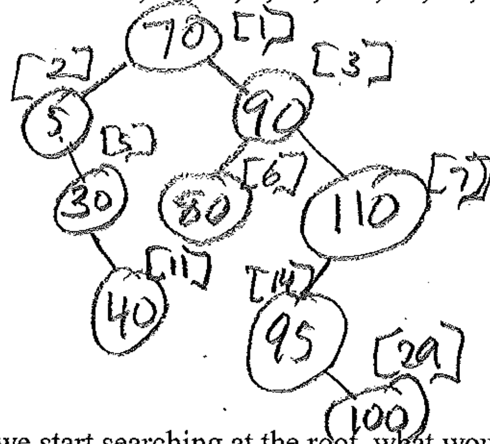
```
class TreeNode:
  ...
    def findSuccessor(self):
        succ = None
        if self.hasRightChild():
            succ = self.rightChild.findMin()
        else:
            if self.parent:
                if self.isLeftChild():
                    succ = self.parent
                else:
     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



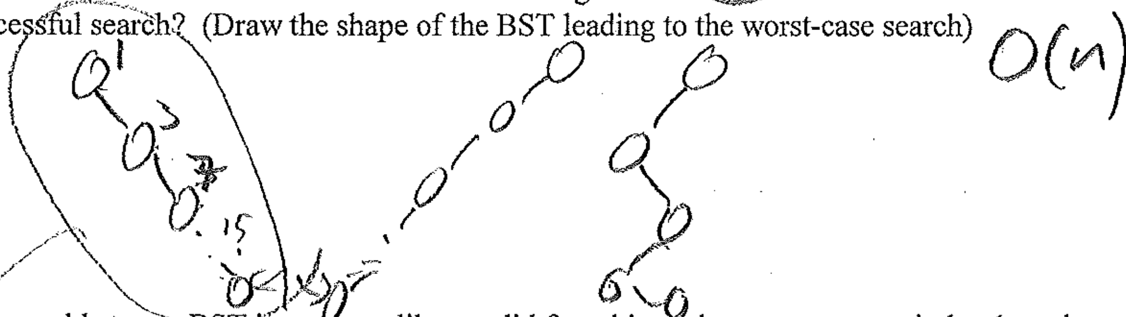
6. The shape of a BST depends on the order in which values are added (and deleted).

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

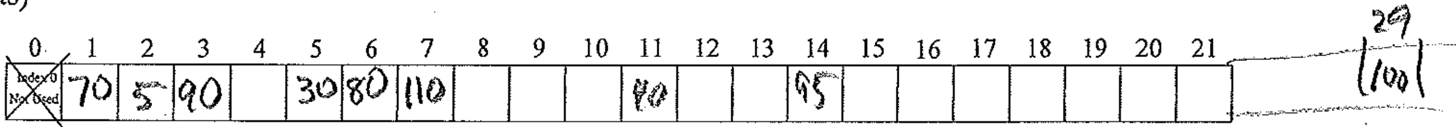


b) If a BST contains n nodes and we start searching at the root, what would be the worst-case big-oh $O()$ notation for a successful search? (Draw the shape of the BST leading to the worst-case search)



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 * i, and right child at index 2 * i + 1.

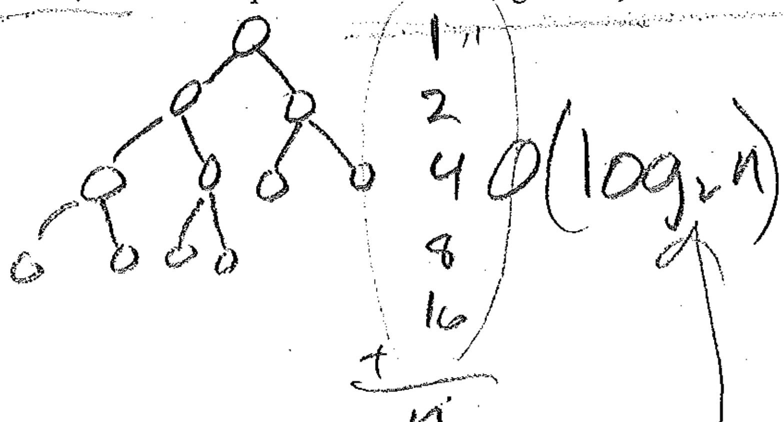
a) Draw the above BST (after inserting: 70, 90, 80, 5, 30, 110, 95, 40, 100) stored in an array (leave blank unused slots)



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
Index 0 Not Used	70	5	90		30	80	110				40			95							

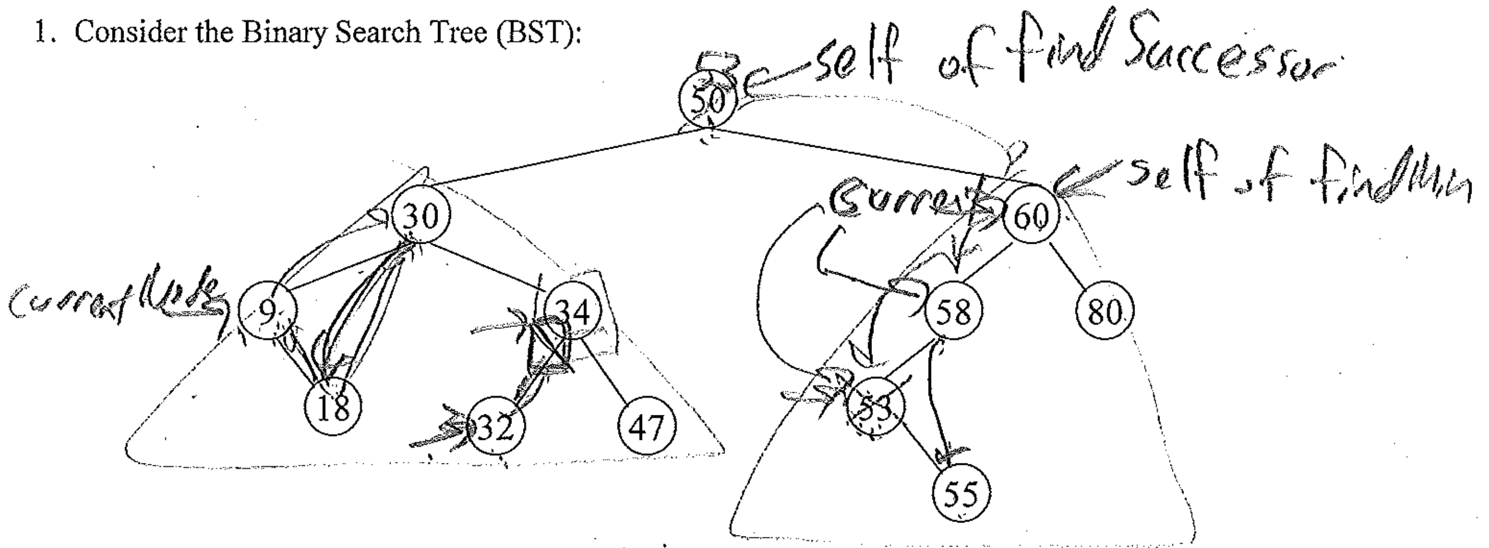
b) What would be the worst-case storage needed for a BST with n nodes?

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?

1. Consider the Binary Search Tree (BST):



a. What would need to be done to delete 32 from the BST? change it's parent's leftChild or rightChild pointer to None

b. What would need to be done to delete 9 from the BST? link around it by changing it's parent and child's pointers

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 or 47. Textbook's code picks 53 -- largest value in deleted nodes right subtree

d. Which node would the `TreeNode's findSuccessor` method return for `succ` if we are deleting 50 from the BST? 53

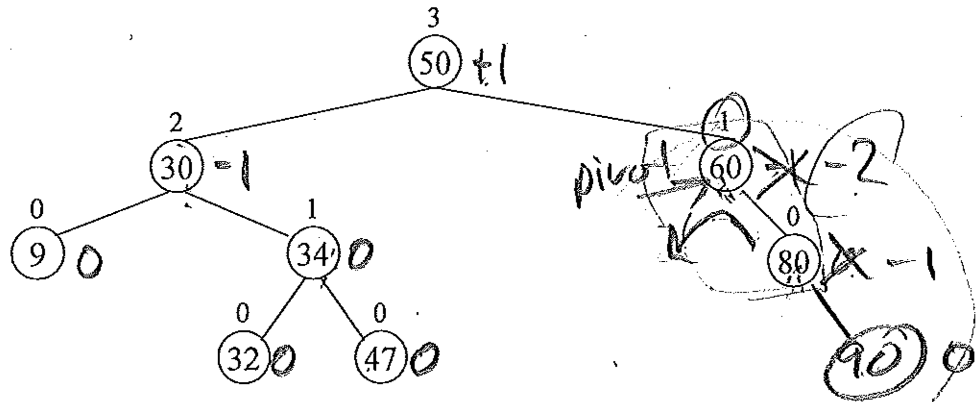
2. When the `findSuccessor` method is called how many children does the `self` node have? 2

3. How could we improve the `findSuccessor` method? - eliminate "dead code" -- code that can never run. (see page 4 of lecture 19)

4. When the `spliceOut` method is called from `remove` how many children could the `self` node have? At most a rightChild since its the smallest node in a subtree.

5. How could we improve the `spliceOut` method? eliminate dead code (see page 4 of lecture 19)

1. An *AVL Tree* is a special type of Binary Search Tree (BST) that it is *height balanced*. By height balanced I mean that the height of every node's left and right subtrees differ by at most one. This is enough to guarantee that a AVL tree with n nodes has a height no worst than $O(1.44 \log_2 n)$. Therefore, insertions, deletions, and search are worst case $O(\log_2 n)$. An example of an AVL tree with integer keys is shown below. The height of each node is shown.



Each AVL-tree node usually stores a *balance factor* in addition to its key and payload. The balance factor keeps track of the relative height difference between its left and right subtrees, i.e., height(left subtree) - height(right subtree).

a) Label each node in the above AVL tree with one of the following *balance factors*:

- 0 if its left and right subtrees are the same height
- 1 if its left subtree is one taller than its right subtree
- -1 if its right subtree is one taller than its left subtree

b) We start a `put` operation by adding the new item into the AVL as a leaf just like we did for Binary Search Trees (BSTs). Add the key 90 to the above tree.

c) Identify the node "closest up the tree" from the inserted node (90) that no longer satisfies the height-balanced property of an AVL tree. This node is called the *pivot node*. Label the pivot node above.

d) Consider the subtree whose root is the pivot node. How could we rearrange this subtree to restore the AVL height balanced property? (Draw the rearranged tree below)