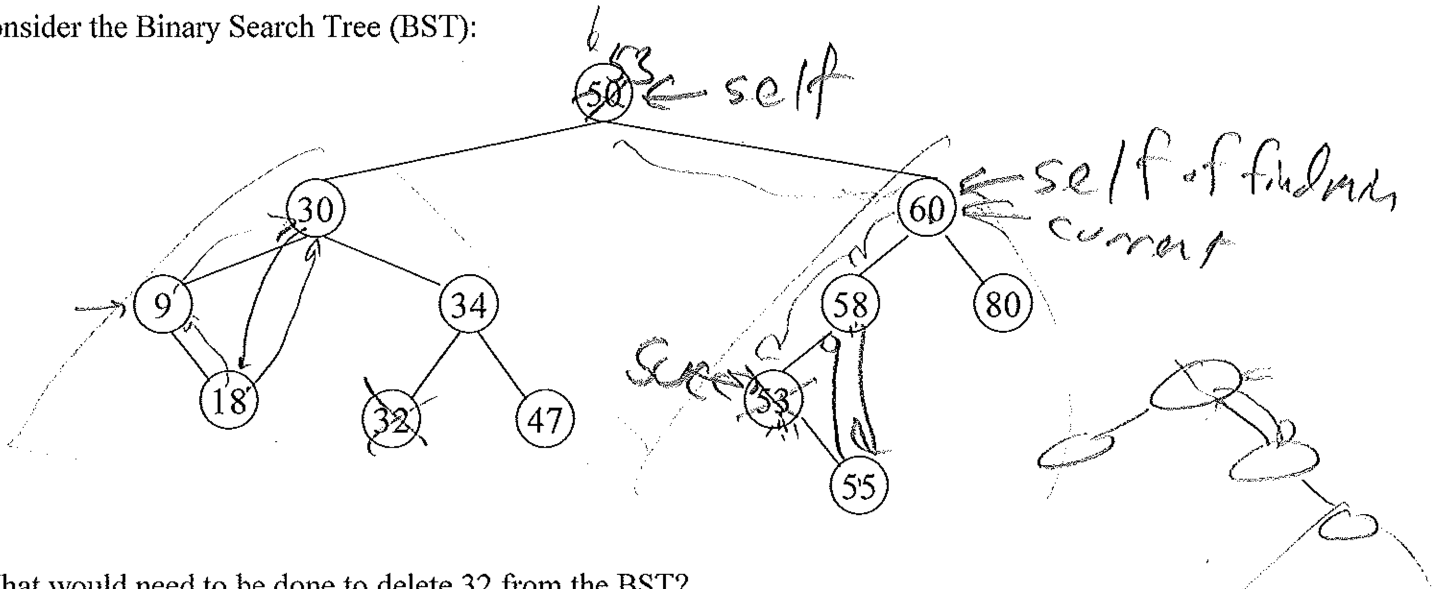
Data Structures (CS 1520) Lecture 20 Name:__________________

1. Consider the Binary Search Tree (BST):



a. What would need to be done to delete 32 from the BST?
Set parent's (34) rightChild to None

b. What would need to be done to delete 9 from the BST? change parent's (30) leftChild 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 " " 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.

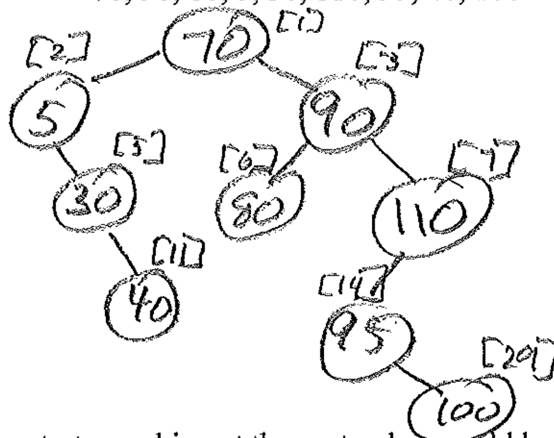
5. How could we improve the `spliceOut` method?

Eliminate "dead code" (see "X"ed out code)

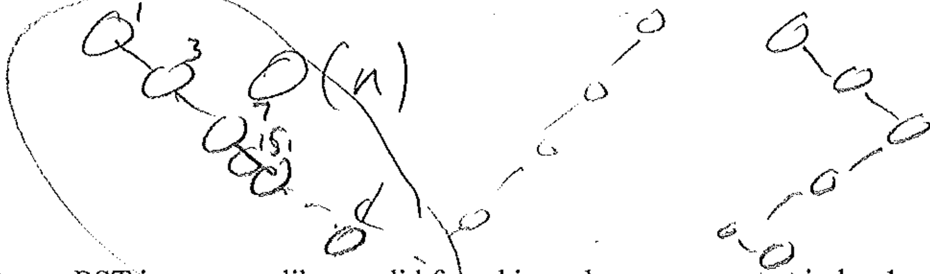
6. The shape of a BST depends on the order in which values are added (and deleted).

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 * 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	...	29
Index 0 Not Used	70	5	90		30	80	110				40			95									100

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

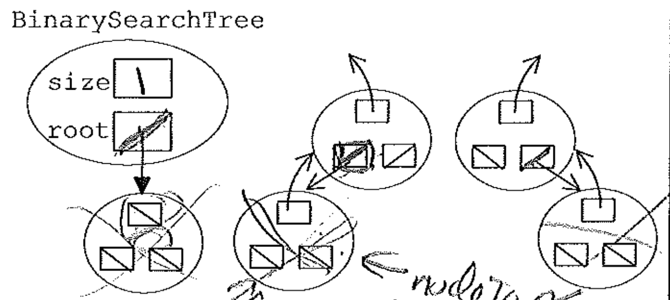
2^n

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.

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.

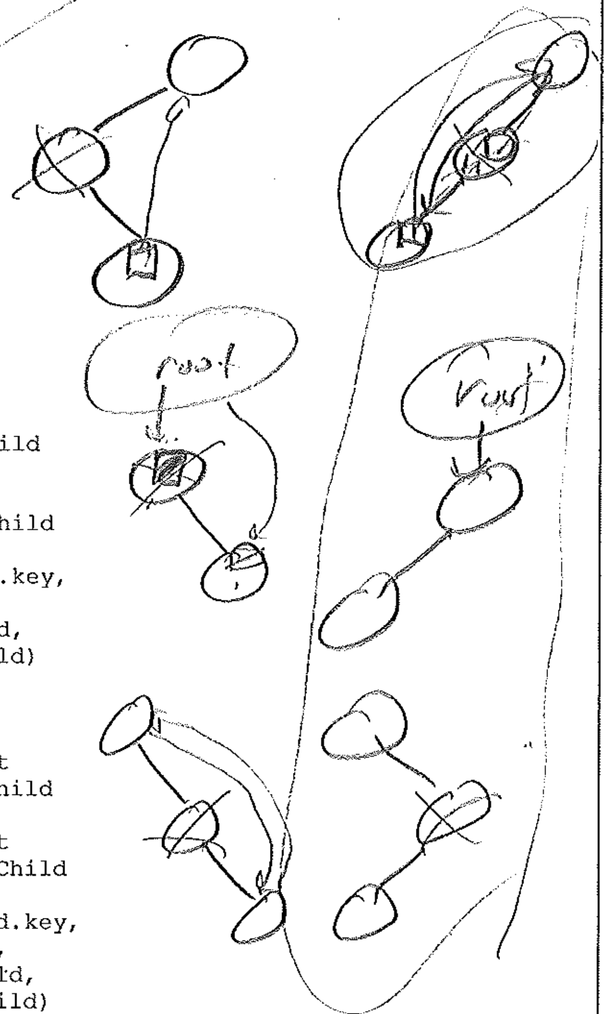
```
class BinarySearchTree:
   . . .
  def delete(self,key):
    if self.size > 1:
        nodeToRemove = self._get(key,self.root)
        if nodeToRemove:
            self.remove(nodeToRemove)
            self.size = self.size-1
        else:
            raise KeyError('Error, key not in tree')
    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.

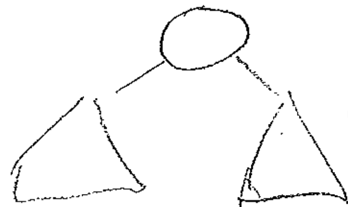
```
class TreeNode:
    . . .
      def findSuccessor(self):
         succ = None
         if self.hasRightChild():
             succ = self.rightChild.findMin()
         else:
             if self.parent:
                 if self.isLeftChild():
                     succ = self.parent
              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

Lecture 23 - AVL-tree

BST that's height balanced



← height of left-subtree and right-subtree differ by at most 1

"**balance factor**" = (height left subtree) − (height right subtree)

-1, 0, +1

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