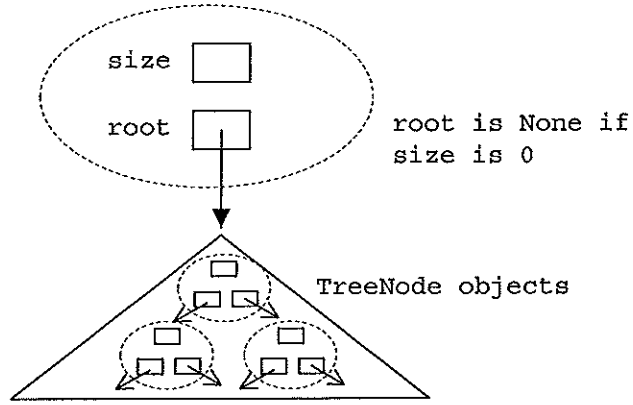
1. Consider the partial TreeNode class and partial BinarySearchTree class.

```
class TreeNode:
   def __init__(self,key,val,left=None,right=None,
                                          parent=None):
        self.key = key
        self.payload = val
        self.leftChild = left
        self.rightChild = right
        self.parent = parent

    def hasLeftChild(self):
        return self.leftChild

    def hasRightChild(self):
        return self.rightChild

    def isLeftChild(self):
        return self.parent and \
               self.parent.leftChild == self

    def isRightChild(self):
        return self.parent and \
               self.parent.leftChild == self

    def isRoot(self):
        return not self.parent

    def isLeaf(self):
      return not (self.rightChild or self.leftChild)

    def hasAnyChildren(self):
        return self.rightChild or self.leftChild

    def hasBothChildren(self):
        return self.rightChild and self.leftChild

    def replaceNodeData(self,key,value,lc,rc):
        self.key = key
        self.payload = value
        self.leftChild = lc
        self.rightChild = rc
        if self.hasLeftChild():
            self.leftChild.parent = self
        if self.hasRightChild():
            self.rightChild.parent = self

    def __iter__(self):
       if self:
          if self.hasLeftChild():
               for elem in self.leftChild:
                  yield elem
          yield self.key
          if self.hasRightChild():
               for elem in self.rightChild:
                  yield elem
```

self

Self

A BinarySearchTree object




```
class BinarySearchTree:
  def __init__(self):
      self.root = None
      self.size = 0

  def length(self):
      return self.size

  def __len__(self):
      return self.size

  def __iter__(self):
      return self.root.__iter__()

  def __str__(self):
   """Returns a string representation of the tree
      rotated 90 degrees counter-clockwise"""

     def strHelper(root, level):
        resultStr = ""
        if root:
           resultStr += strHelper(root.rightChild,
                                  level+1)
           resultStr += "| " * level
           resultStr += str(root.key) + "\n"
           resultStr += strHelper(root.leftChild,
                                  level+1)

        return resultStr

    return strHelper(self.root, 0)
```

recursive calling __iter__

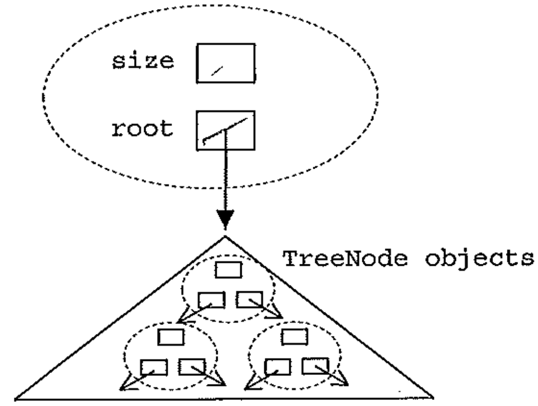
a) How do the BinarySearchTree __iter__ and __str__ methods work?

__iter__ calls itself recursively using the "for elem in .." to do an inorder traversal

More partial TreeNode class and partial BinarySearchTree class.

```
class BinarySearchTree:              if k in bst:
    ...
    def __contains__(self,key):
        if self._get(key,self.root):
            return True
        else:
            return False

    def get(self,key):
        if self.root:
            res = self._get(key,self.root)
            if res:
                return res.payload
            else:
                return None
        else:
            return None

    def _get(self,key,currentNode):
        if not currentNode:
            return None
        elif currentNode.key == key:
            return currentNode
        elif key < currentNode.key:
            return self._get(key,currentNode.leftChild)
        else:
            return self._get(key,currentNode.rightChild)

    def __getitem__(self,key):
        return self.get(key)

    def __setitem__(self,k,v):
        self.put(k,v)

    def put(self,key,val):
        if self.root:
            self._put(key,val,self.root)
        else:
            self.root = TreeNode(key,val)
        self.size = self.size + 1

    def _put(self,key,val,currentNode):
        if key < currentNode.key:
            if currentNode.hasLeftChild():
                self._put(key,val,currentNode.leftChild)
            else:
                currentNode.leftChild = TreeNode(key,val)

        elif key > currentNode.key:
            if currentNode.hasRightChild():
                self._put(key,val,currentNode.rightChild)
            else:
                currentNode.rightChild = TreeNode(key,val)

        else: # key ==
            currentNode.payload = val
            self.size -= 1
```

A BinarySearchTree object



b) The `_get` method is the "work horse" of BST search. It recursively walks `currentNode` down the tree until it finds `key` or becomes `None`

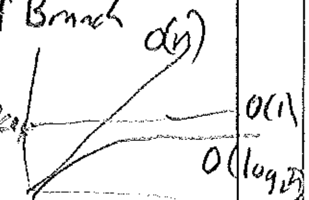
In English, what are the base and recursive cases?

Base cases: (1) walk currentNode off bottom of Branch

(2) found matching key

Recursive: (1) key < current key search left subtree

(2) search right subtree

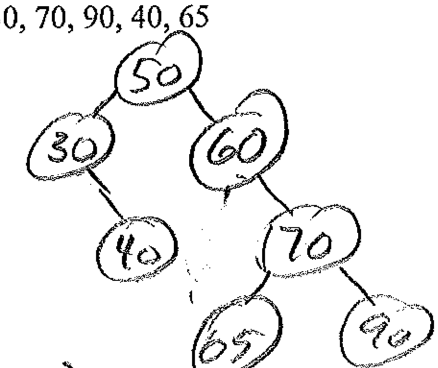


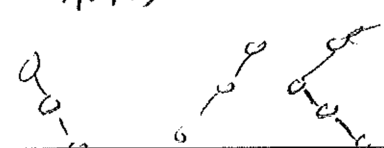
c) What is the `put` method doing?

Handles adding first node to BST and calls _put with pointer to root TreeNode

d) Complete the recursive `_put` method.

e) Draw the "shape" of the BST after `put`s of: 50, 60, 30, 70, 90, 40, 65

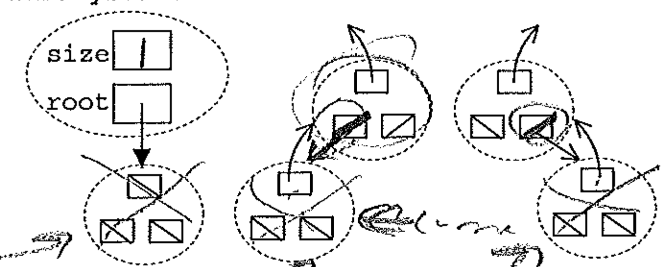




f) If "n" items are in the BST, what is `put`'s: Best-case O(1)? Worst-case O(n)? Average-case O(log2 n)?

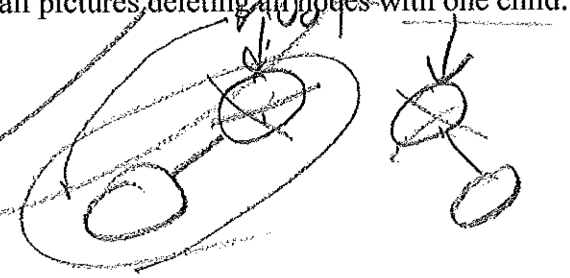
2. More partial TreeNode class and partial BinarySearchTree class.

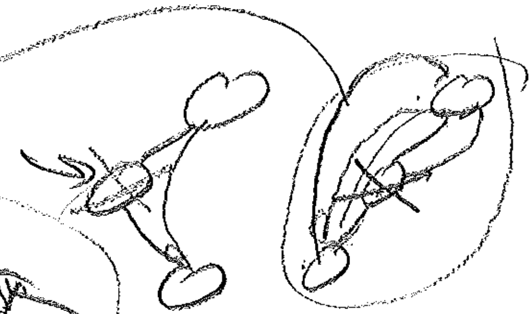
a) Update picture where we delete a leaf.

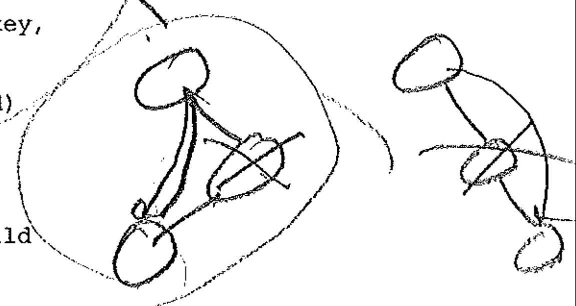


b) Where in the code is each handled?

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





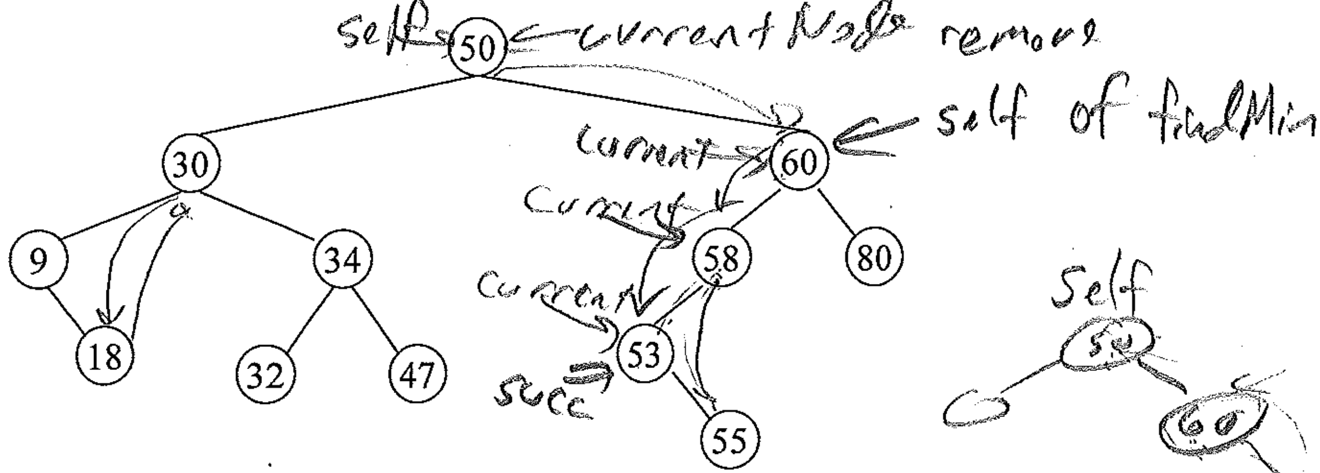


```
class BinarySearchTree:
  . . .
 def delete(self,key):
   if self.size > 1:
      nodeToRemove = self._get(key,self.root)
      if nodeToRemove:
          self.remove(nodeToRemove)
          self.size = self.size-1
      else:
          raise KeyError('Error, key not in tree')
   elif self.size ==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

1. Consider the Binary Search Tree (BST):



a. What would need to be done to delete 32 from the BST?
Change parent's point to Node to None

b. What would need to be done to delete 9 from the BST?
Change parent's point to deleted nodes child and set new child's parent

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? Smallest node in right subtree (53)
(alternatively ; largest node in left subtree (47))

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

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

3. How could we improve the `findSuccessor` method? eliminate "dead code" that never runs.

4. When the `spliceOut` method is called from `remove` how many children could the `self` node have?
at most a left child

5. How could we improve the `spliceOut` method?
eliminate "dead 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
Index 0 Not Used																					

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?