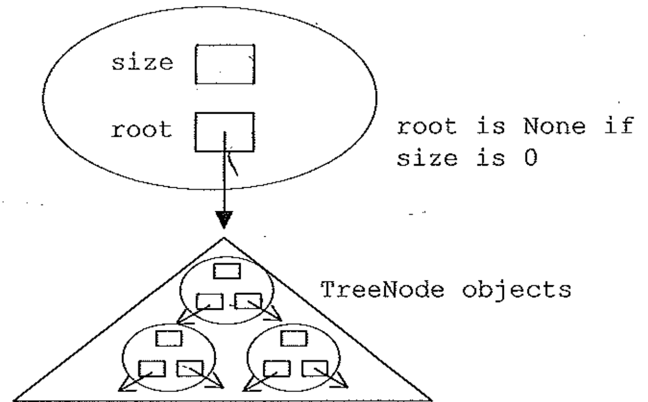
Data Structures (CS 1520) Lecture 19 Name:_________________

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
                          rightChild

    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
```

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)
```

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

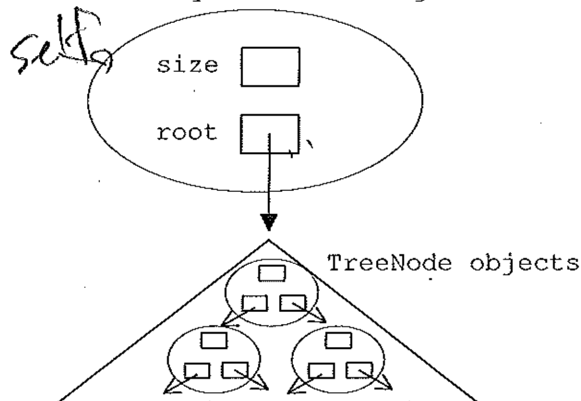
__iter__ : recursion to do an inorder traversal

strHelper: does a non-standard traversal; right subtree, root
left subtree

More partial TreeNode class and partial BinarySearchTree class.

```
class BinarySearchTree:
    ...
    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,parent=currentNode)
                self.size += 1
        elif key > currentNode.key:
            if currentNode.hasRightChild():
                self._put(key,val,currentNode.rightChild)
            else:
                currentNode.rightChild = TreeNode(key,val,parent=currentNode)
                self.size += 1
        else:
            currentNode.payload = val
```

(Handwritten annotations: "base cases" pointing to the first two branches of `_get`; "recursive" beside the last two; "redundant" pointing to `self.size = self.size + 1` in `put`.)



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?

currentNode walks off bottom of tree branch

Found match for key

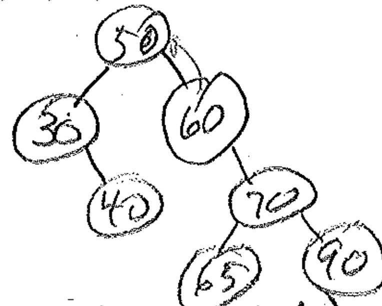
Recursive cases check either left or right subtree

c) What is the `put` method doing?

Handles special case of adding first TreeNode and in

d) Complete the recursive `_put` method.

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



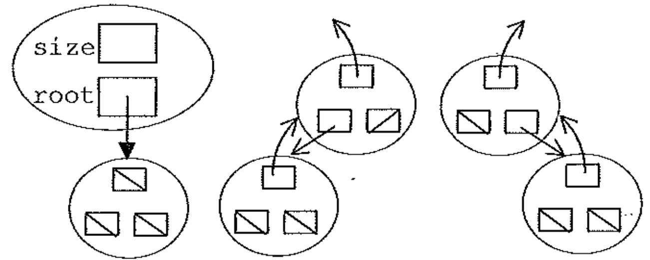
f) If "n" items are in the BST, what is `put`'s: Best-case $O(1)$? Worst-case $O(n)$? Average-case $O(\log_2 n)$

shape of tree determines performance

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:
            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)
```

a) Update picture where we delete a leaf.



b) Where in the code is each handled?

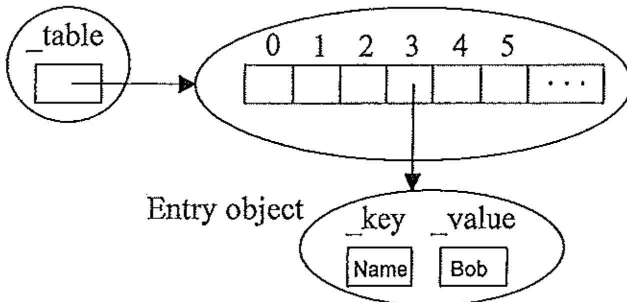
c) Draw all pictures deleting all nodes with one child.

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
```

1. The Map/Dictionary abstract data type (ADT) stores key-value pairs. The key is used to look up the data value.

Method call	Class Name	Description
`d = ListDict()`	`__init__(self)`	Constructs an empty dictionary
`d["Name"] = "Bob"`	`__setitem__(self,key,value)`	Inserts a key-value entry if `key` does not exist or replaces the old value with `value` if `key` exists.
`temp = d["Name"]`	`__getitem__(self,key)`	Given a `key` return it value or `None` if `key` is not in the dictionary
`del d["Name"]`	`__delitem__(self,key)`	Removes the entry associated with `key`
`if "Name" in d:`	`__contains__(self,key)`	Return `True` if `key` is in the dictionary; return `False` otherwise
`for k in d:`	`__iter__(self)`	Iterates over the keys in the dictionary
`len(d)`	`__len__(self)`	Returns the number of items in the dictionary
`str(d)`	`__str__(self)`	Returns a string representation of the dictionary




```
class Entry(object):
    """A key/value pair."""

    def __init__(self, key, value):
        self._key = key
        self._value = value

    def getKey(self):
        return self._key

    def getValue(self):
        return self._value

    def setValue(self, newValue):
        self._value = newValue

    def __eq__(self, other):
        if not isinstance(other, Entry):
            return False
        return self._key == other._key

    def __str__(self):
        return str(self._key) + ":" + str(self._value)
```

```
from entry import Entry

class ListDict(object):
    """Dictionary implemented with a Python list."""

    def __init__(self):
        self._table = []

    def __getitem__(self, key):
        """Returns the value associated with key or
        returns None if key does not exist."""
        entry = Entry(key, None)
        try:
            index = self._table.index(entry)
            return self._table[index].getValue()
        except:
            return None

    def __delitem__(self, key):
        """Removes the entry associated with key."""
        entry = Entry(key, None)
        try:    # NOTE: Python list index method
                # errors on unsuccessful search
            index = self._table.index(entry)
            self._table.pop(index)
        except:
            return

    def __str__(self):
        """Returns string repr. of the dictionary"""
        resultStr = "{"
        for item in self._table:
            resultStr = resultStr + " " + str(item)
        return resultStr + " }"

    def __iter__(self):
        """Iterates over keys of the dictionary"""
        for item in self._table:
            yield item.getKey()
        raise StopIteration
```

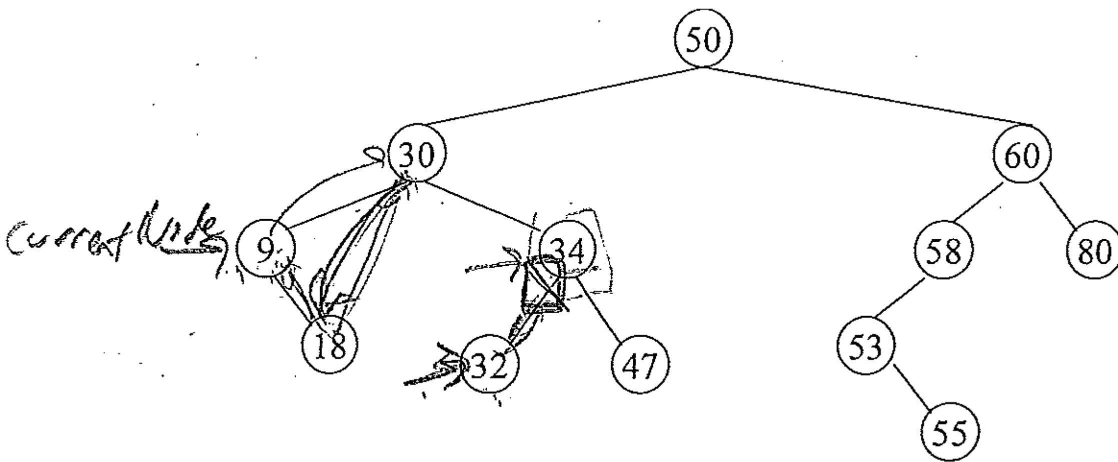
a) Complete the code for the `__contains__` method.

```
def __contains__(self, key):
```

b) Complete the code for the `__setitem__` method.

```
def __setitem__(self, key, value):
```

1. Consider the Binary Search Tree (BST):



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

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

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?

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

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

3. How could we improve the `findSuccessor` method?

4. When the `spliceOut` method is called from `remove` how many children could the `self` node have?

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