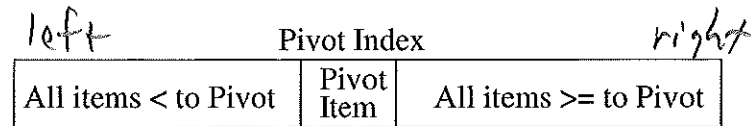


5. Quick sort general idea is as follows.

- Select a "random" item in the unsorted part as the *pivot*
- Rearrange (*partitioning*) the unsorted items such that:
 - Quick sort the unsorted part to the left of the pivot
 - Quick sort the unsorted part to the right of the pivot



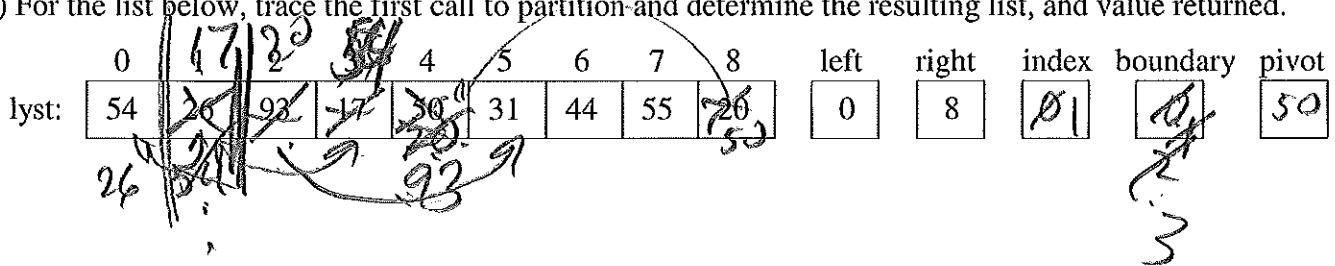
a) Given the following partition function which returns the index of the pivot after this rearrangement, complete the recursive quicksortHelper function.

```
def partition(lyst, left, right):
    # Find the pivot and exchange it with the last item
    middle = (left + right) // 2
    pivot = lyst[middle]
    lyst[middle] = lyst[right]
    lyst[right] = pivot
    # Set boundary point to first position
    boundary = left
    # Move items less than pivot to the left
    for index in range(left, right):
        if lyst[index] < pivot:
            temp = lyst[index]
            lyst[index] = lyst[boundary]
            lyst[boundary] = temp
            boundary += 1
    # Exchange the pivot item and the boundary item
    temp = lyst[boundary]
    lyst[boundary] = lyst[right]
    lyst[right] = temp
    return boundary
```

```
def quicksort(lyst):
    quicksortHelper(lyst, 0, len(lyst) - 1)
```

```
def quicksortHelper(lyst, left, right):
    if left < right:
        pivotIndex = partition(lyst, left, right)
        quicksortHelper(lyst, left, pivotIndex - 1)
        quicksortHelper(lyst, pivotIndex + 1, right)
```

b) For the list below, trace the first call to partition and determine the resulting list, and value returned.



b) What initial arrangement of the list would cause partition to perform the most amount of work?

Handwritten answer: pivot is largest item that's initial at middle

c) Let "n" be the number of items between left and right. What is the worst-case $O(\)$ for partition?

Handwritten answer: $O(n)$

d) What would be the overall, worst-case $O(\)$ for Quick Sort?

Handwritten answer: $O(n^2)$ ← see attached.

e) Ideally, the pivot item splits the list into two equal size problems. What would be the big-oh for Quick Sort in the best case?

always splits
in middle



⋮
⋮
⋮

n
 n
 $\log_2 n$ levels
best case
 $O(n \log_2 n)$ best case

f) What would be the big-oh for Quick Sort in the average case?

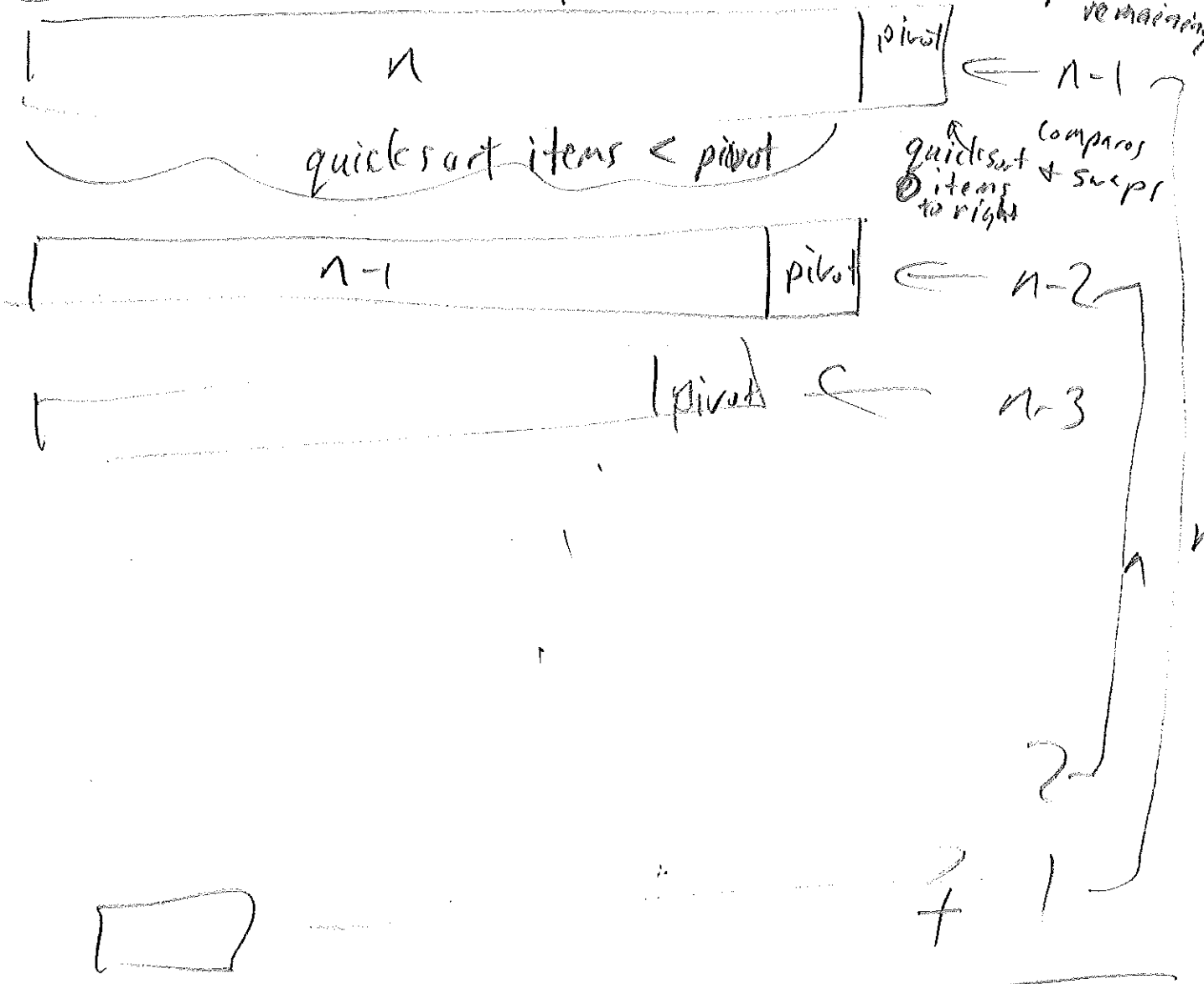
Average case is $O(n \log_2 n)$, but it is hard to show. Intuitively, it makes sense since the "randomly" selected pivot would on "average" be about in the middle after partition.

g) The textbook's partition code (Listing 5.15 on page 225) selects the first item in the list as the pivot item. However, the above partition code selects the middle item of the list to be the pivot. What advantage does selecting the middle item as the pivot have over selecting the first item as the pivot?

Sometime we sort data that's partially or mostly sorted just to be sure its sorted.

By picking the middle item, if the data was initial sort, then we get close to "best" case performance.

Worst-case Quicksort occurs if pivot selected is always largest item. remaining

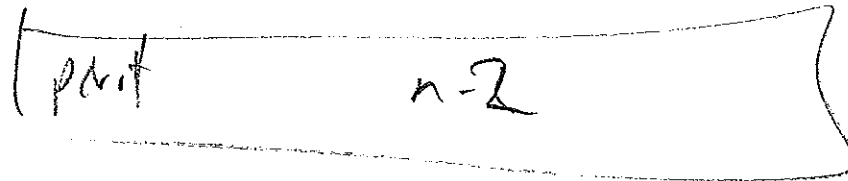
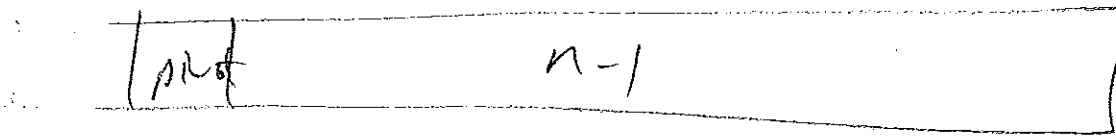
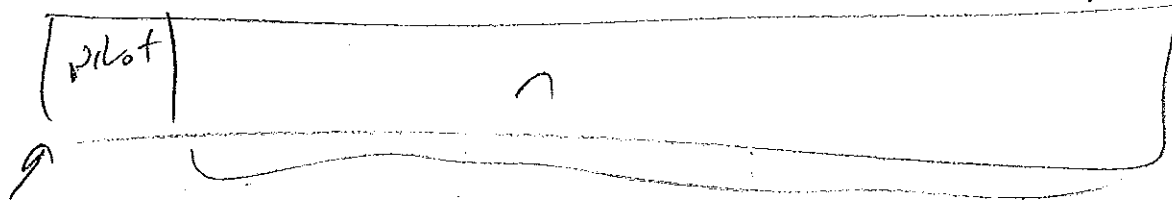


$$= n \frac{(n-1)}{2}$$

$$O(n^2)$$

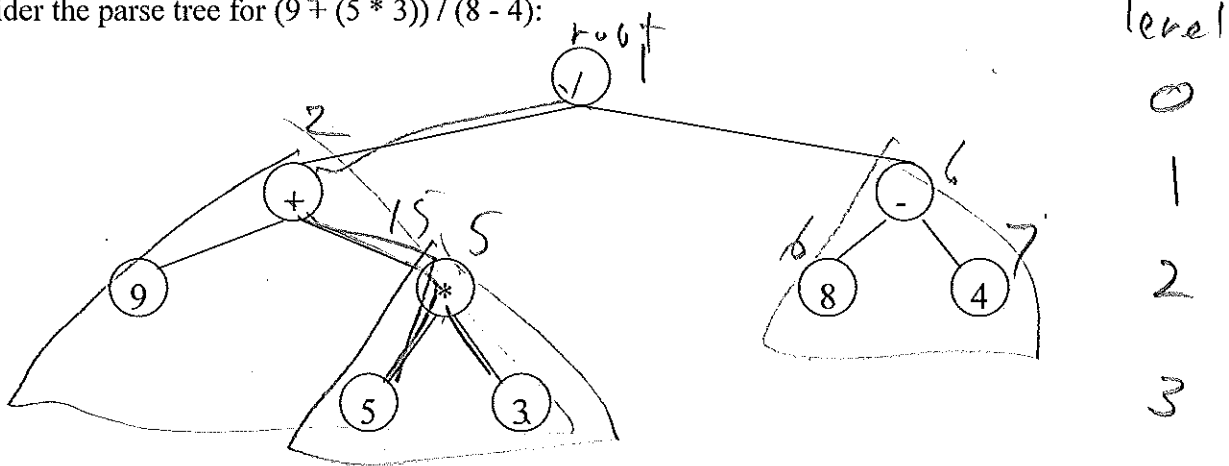
worst case

ascending initial + pick 1st item pivot



$O(n^2)$ compares
(no swaps)

1. Consider the parse tree for $(9 + (5 * 3)) / (8 - 4)$:



a) Identify the following items in the above tree:

- node containing "*" 9
- edge from node containing "-" to node containing "8"
- root node
- children of the node containing "+" 9, *
- parent of the node containing "3" *
- siblings of the node containing "*" 9
- leaf nodes of the tree 9, 5, 3, 8, 4
- subtree who's root is node contains "+"
- path from node containing "+" to node containing "5"
- branch from root node to "3"

b) Mark the levels of the tree (level is the number of edges on the path from the root)

c) What is the height (max. level) of the tree? 3

Note: tree does not need to be binary

2. In Python an easy way to implement a tree is as a list of lists where a tree look like:

["node value", remaining items are subtrees for the node each implemented as a list of lists]

Complete the list-of-lists representation look like for the above parse tree:

['/', ['+', [9], ['*', [5], [3]]], ['-', [8], [4]]]

3. Consider a "linked" representations of a BinaryTree. For the expression $((4 + 5) * 7)$, the binary tree would be:

```

class BinaryTree:
    def __init__(self, rootObj):
        self.key = rootObj
        self.leftChild = None
        self.rightChild = None
    
```

```

import operator
class BinaryTree:
    def __init__(self, rootObj):
        self.key = rootObj
        self.leftChild = None
        self.rightChild = None

    def insertLeft(self, newNode):
        if self.leftChild == None:
            self.leftChild = BinaryTree(newNode)
        else:
            t = BinaryTree(newNode)
            t.left = self.leftChild
            self.leftChild = t

    def insertRight(self, newNode):
        if self.rightChild == None:
            self.rightChild = BinaryTree(newNode)
        else:
            t = BinaryTree(newNode)
            t.right = self.rightChild
            self.rightChild = t

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

    def getRightChild(self):
        return self.rightChild

    def getLeftChild(self):
        return self.leftChild

    def setRootVal(self, obj):
        self.key = obj

    def getRootVal(self,):
        return self.key

    def inorder(self):
        if self.leftChild:
            self.leftChild.inorder()
        print(self.key)
        if self.rightChild:
            self.rightChild.inorder()

    def postorder(self):
        if self.leftChild:
            self.leftChild.postorder()
        if self.rightChild:
            self.rightChild.postorder()
        print(self.key)

```

a) Fix the insertLeft and insertRight code:
(Listing 6.6 and 6.7 are wrong in the text on pp. 242-3)

"left" should be "leftChild"

"right" should be "rightChild"

```

def preorder(self):
    print(self.key)
    if self.leftChild:
        self.leftChild.preorder()
    if self.rightChild:
        self.rightChild.preorder()

def printexp(self):
    if self.leftChild:
        print('(', end=' ')
        self.leftChild.printexp()
    print(self.key, end=' ')
    if self.rightChild:
        self.rightChild.printexp()
    print(')', end=' ')

def postordereval(self):
    ops = {'+':operator.add, '-':operator.sub,
          '*':operator.mul, '/':operator.truediv}
    res1 = None
    res2 = None
    if self.leftChild:
        res1 = self.leftChild.postordereval()
    if self.rightChild:
        res2 = self.rightChild.postordereval()
    if res1 and res2:
        return ops[self.key](res1, res2)
    else:
        return self.key

```

Note:
None evaluates
as False
Any non-None
value as True

Some corresponding external (non-class) functions:

```

def inorder(tree):
    if tree != None:
        inorder(tree.getLeftChild())
        print(tree.getRootVal())
        inorder(tree.getRightChild())

def printexp(tree):
    if tree.leftChild:
        print('(', end=' ')
        printexp(tree.getLeftChild())
    print(tree.getRootVal(), end=' ')
    if tree.rightChild:
        printexp(tree.getRightChild())
    print(')', end=' ')

def height(tree):
    if tree == None:
        return -1
    else:
        return 1 +
            max(height(tree.leftChild),
                height(tree.rightChild))

```

```

def printexp(tree):
    sVal = ""
    if tree:
        sVal = '(' + printexp(tree.getLeftChild())
        sVal = sVal + str(tree.getRootVal())
        sVal = sVal + printexp(tree.getRightChild()) + ')'
    return sVal

def postordereval(tree):
    ops = {'+':operator.add, '-':operator.sub,
          '*':operator.mul, '/':operator.truediv}
    res1 = None
    res2 = None
    if tree:
        res1 = postordereval(tree.getLeftChild())
        res2 = postordereval(tree.getRightChild())
        if res1 and res2:
            return ops[tree.getRootVal()](res1, res2)
    else:
        return tree.getRootVal()

```

b) If myTree is the BinaryTree object for the expression: $((4 + 5) * 7)$, what gets printed by a call to:

| myTree.inorder() | myTree.preorder() | myTree.postorder() | inorder(myTree) |
|------------------|-------------------|--------------------|-----------------|
| 4 | * | 4 | 4 |
| + | + | 5 | + |
| 5 | 4 | + | 5 |
| * | 5 | 7 | * |
| 7 | 7 | * | 7 |

c) If myTree is the BinaryTree object for the expression: $((4 + 5) * 7)$, what gets printed by a call to myTree.printexp()? $((4+5)*7)$ fully parenthesized expression

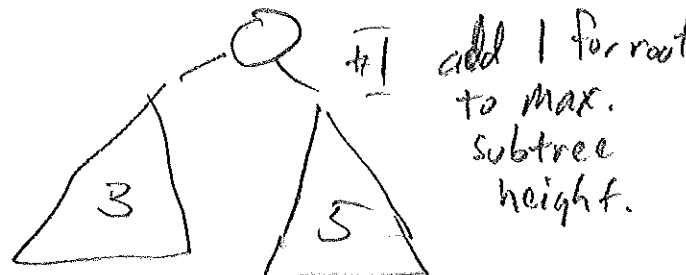
d) If myTree is the BinaryTree object for the expression: $((4 + 5) * 7)$, what gets printed by a call to myTree.postordereval()? 63

e) From an class/Abstract Data Type (ADT) point of view, why are the external versions of the methods "better"?

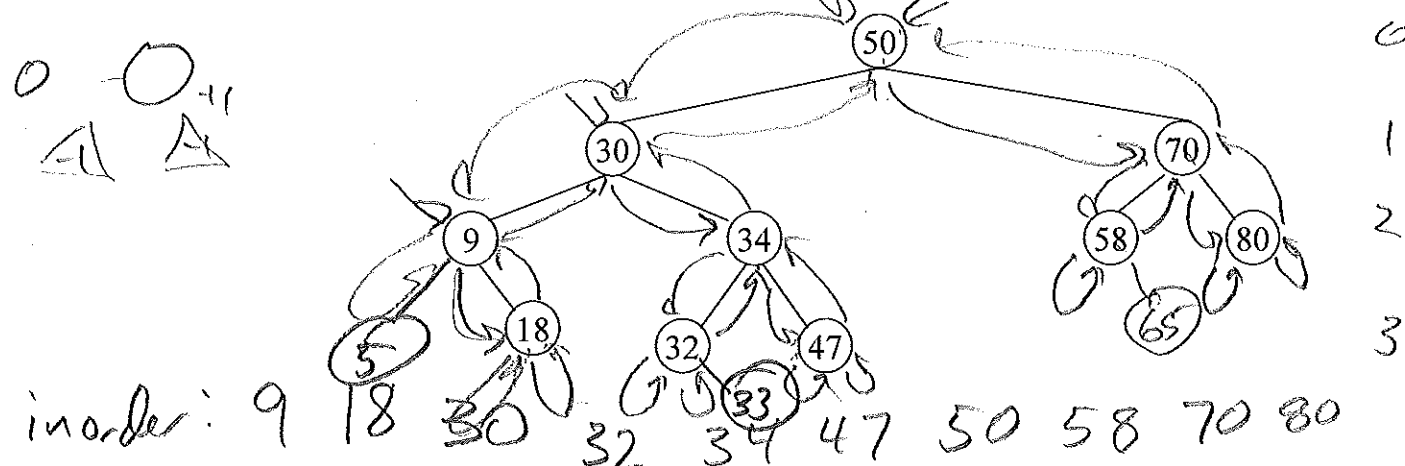
The BinaryTree class methods; printexp and postordereval only work if the BinaryTree objects are used for a parse tree. Or other uses they are useless.

f) Write the height method for the BinaryTree class.

```
def height(self):
    if self.isLeaf():
        return 0
    else:
        return 1 + max(self.leftChild.height(), self.rightChild.height())
```



4. Consider the Binary Search Tree (BST). For each node, all values in the left-subtree are < the node and all values in the right-subtree are > the node.



- a. Starting at the root, how would you find the node containing "32"?
- b. Starting at the root, when would you discover that "65" is not in the BST?
- c. Starting at the root, where would be the "easiest" place to add "65"?
- d. Where would we add "5" and "33"?

postorder: 18 9 32 47 34 30 58 80 70 50