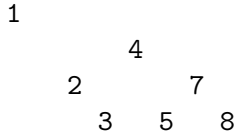
Question 1. [10 MARKS]

This question has you draw trees. We don't want you to draw a memory model; just draw circles with values inside them and lines connecting the circles (like we've been doing in lecture).

Part (a) [3 MARKS]

This subquestion is about **binary search trees**. Draw the tree that results from inserting the following values in the order shown:

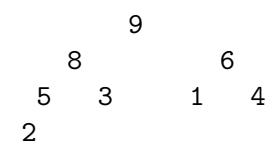
```
1 4 2 7 5 3 8
```



Part (b) [3 MARKS]

This subquestion is about **max heaps**. Draw the tree that results from inserting the following values into a max heap in the order shown.

```
2 6 4 3 5 1 9 8
```



Part (c) [1 MARK]

Draw the largest tree (the one with the most values) that you can think of that is both a binary search tree and a max heap.

```
    4
2
```

There can't be a right subtree because the heap property and BST property conflict on that side. 2 can't have a child because of the shape property.

Part (d) [1 MARK]

Draw the largest tree (the one with the most values) that you can think of that is both a binary search tree and a min heap.

```
4
```

There can't be a left subtree because the min heap property and the BST property conflict on that side. There can't be a right child because of the shape property.

Part (e) [2 MARKS]

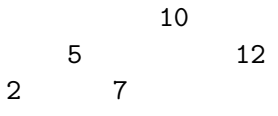
Programmers usually implement heaps using a list. Explain why programmers don't usually implement binary search trees using a list.

Because there is no shape property for a BST there might be a lot of `None` entries, using way more memory than necessary.

Question 2. [5 MARKS]

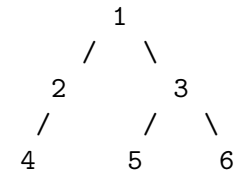
A file storing the pre-order, post-order and in-order traversals of a **binary search tree** has been corrupted and most values have been lost. Below, these "lost" values are replaced with the letter `x`. Draw the binary search tree that produced the file. (If you think more than one tree matches these traversals, you can draw any of them.)

```
  pre-order:  x,  x,  x,  x,  7, 12
 post-order:  x,  2,  x,  5,  x, 10
    in-order:  x,  4,  x,  x,  x,  x
```



Question 3. [8 MARKS]

In the tree below, 2 is the left child of 1, 4 is the left child of 2, and 5 is the left child of 3. There are 3 left children.



Complete the following function.

```
def count_left_children(root):
    '''(Node) -> int
    Return the number of left children in the tree rooted at root.'''

    if not root:
        return 0

    count = count_left_children(root.left) + count_left_children(root.right)
    if root.left:
        count += 1

    return count
```

Question 4. [8 MARKS]

A *ternary search tree* allows three subtrees. As with binary search trees, the left subtree contains values less than the key in the root and the right subtree contains values greater than the key in the root. The third "middle" branch contains values equal to the key in the root.

Here is a ternary node class:

```
class TernaryNode(object):
    def __init__(self, v):
        '''(TernaryNode, int) -> NoneType
        A ternary node containing v with no children.'''
        self.key = v
        self.left = None
        self.middle = None
        self.right = None
```

Here is code that builds a ternary search tree and calls `contains_count`, which is described on the opposite page.

```
if __name__ == '__main__':
    r = TernaryNode(7)
    r.left = TernaryNode(3)
    r.left.right = TernaryNode(5)
    r.left.right.middle = TernaryNode(5)
    r.left.right.middle.middle = TernaryNode(5)
    print contains_count(r, 5, 3) # True
    print contains_count(r, 5, 2) # False
    print contains_count(r, 5, 4) # False
    print contains_count(r, 3, 1) # True
    print contains_count(r, 3, 2) # False
    print contains_count(r, 12, 0) # True
```

Complete the following recursive function without using a helper method. Hint: in the situation when you recurse down the middle subtree, subtract 1 from `k`.

```
def contains_count(root, v, k):
    '''(Node, int, int) -> bool
    Return whether the ternary tree rooted at root contains exactly k copies
    of value v.'''

    if not root:
        return k == 0
    else:
        if v < root.key:
            return contains_count(root.left, v, k)
        elif v > root.key:
            return contains_count(root.right, v, k)
        else: # k == r.key
            return contains_count(root.middle, v, k - 1)
```