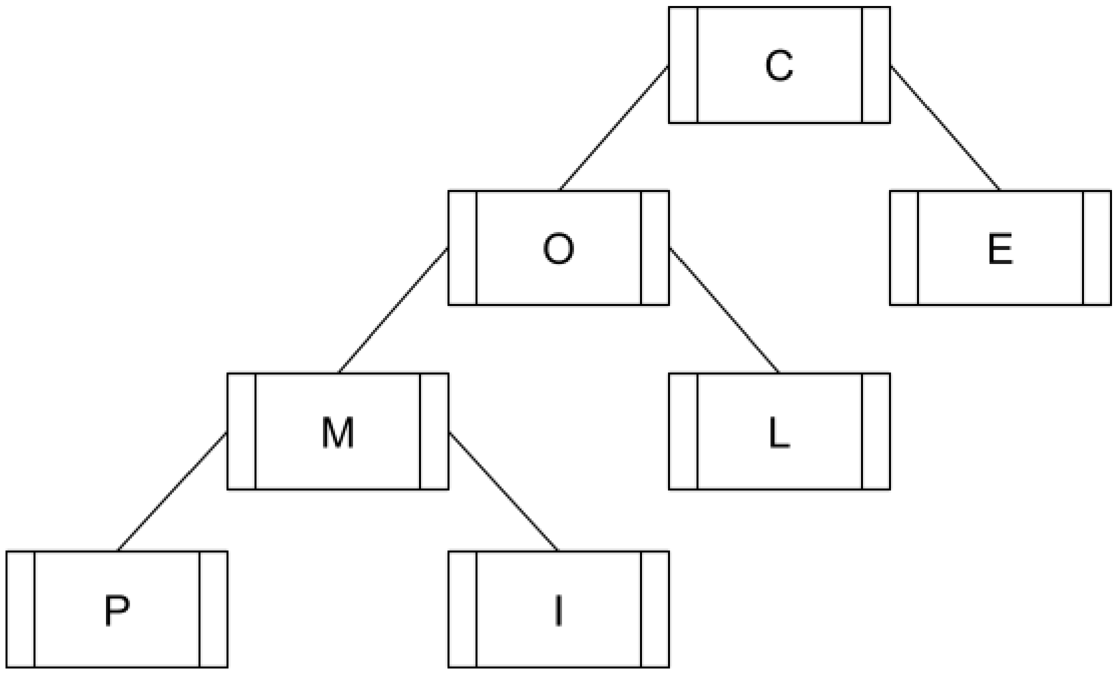
Problem #1 **Name:** Alexander Michelbrink



Preorder: COMPILE

Inorder: PMIOLCE

Problem #2

Since we know that the depth of the root is 0, we can work from the top-down to calculate and set the depths of every node within the tree. This involves using a pre-order traversal and calculating the depth of the node as the "visit" step. To begin the algorithm, the root of tree ***T*** must be placed into the argument ***v***.

Pseudocode:

```
Algorithm computeDepth(v)
    if ( isRoot(v) )
        setDepth(v,0)
    else
        setDepth(v, getDepth(parent(v)) + 1)
    for each child w of v
        computeDepth(w)
```

All operations used within the computeDepth(***v***) algorithm operate in O(1) time. Therefore, when analyzing runtime, we can focus on the amount of times the algorithm is recursively called for any tree ***T*** with ***n*** nodes. There are ***n-1*** children total within the tree, meaning that the "for each child" section of the algorithm makes ***n-1*** recursive calls total. Therefore, the computeDepth(***v***) function operates in O(***n***) time.

Problem #3

There are two main cases to consider when finding the next node in an in-order traversal of a binary tree: ***v*** is an internal node or ***v*** is an external node.

In the case that ***v*** is an internal node, the next node will always be the leftmost child of the right child of ***v***. There are two possibilities if ***v*** is an external node. If ***v*** is the left child of its parent, then the next node would be the parent itself. If ***v*** is the right child of its parent, then the next node would be the next of ***v***'s ancestors to have ***v*** or ***v***'s ancestors as its' left child.

We also have to consider the case where ***v*** is the final node of the tree. In the case of an inorder traversal, this would mean the node is the right child of the root. Therefore, we can simply throw an exception if this occurs.

Pseudocode:

```
function inorderNext(v)
    next <- v
    if ( isInternal(next) )
        next <- rightChild(next)
        while ( leftChild(next) )
            next <- leftChild(next)
    else
        while (leftChild(parent(next))!= next)
            next <- parent(v)
            if (isRoot(next))
                throw finalNodeException
        next <- parent(v)
```

Since each operation used in the algorithm operates in O(1) time, we can look at how many times the while-loops could execute in order to discover the worst-case running time. The most amount of times the while-loop would execute is the height of the tree since the longest path would be from the deepest external to the root or vice versa. Therefore, if the height of the tree is ***h***, then its worst case running time would be O(***h***).

Since $h \leq \frac{1}{2}(n-1)$, in the worst-case of $h = \frac{1}{2}(n-1)$, we could also write the above case as running in time O(***n***).

Problem #4

Upon using the website listed below for research, I discovered that an efficient way to carry out level-order traversal would be through using a queue as an auxiliary data structure that holds the nodes of the tree as elements. Using this idea, I built the following algorithm with queue ***Q***…

Pseudocode:

```
Algorithm levelOrder(v)
     if ( leftChild(v) )
          Q.enqueue(leftChild(v))
     if ( rightChild(v) )
          Q.enqueue(rightChild(v))
     visit(v)
     if ( !Q.isEmpty() )
          levelOrder(Q.dequeue())
```

The algorithm works by enqueueing the children of the current node ***v*** into ***Q*** before visiting ***v***. Then, if ***Q*** is not empty, the algorithm will recursively call itself on the node given using the dequeue() method on ***Q***. Since the algorithm enqueues all of the nodes of one layer before enqueuing the nodes of any lower layer, the dequeue will always return elements layer by layer. Enqueuing the left child before the right also ensures that the traversal travels from left-to-right across layers.

Since each operation used runs in O(1) time, we can look at how many times the algorithm recursively calls itself to analyze the runtime. Since every child will be enqueued before being dequeued and becoming the argument of the recursive call, the algorithm will undergo ***n-1*** recursive calls. This confirms a worst-case runtime of O(***n***).

Sources

GeeksforGeeks. *Level Order Traversal (Breadth First Search or BFS) of Binary Tree*. https://www.geeksforgeeks.org/level-order-tree-traversal/

Problem #5

A.) ____

1. Initial S: **21 14 32 10 44 8 2 11 20 26**

2. First, we remove each first element from the initial Sequence S and insert it into the end of the unsorted Priority Queue. This is repeated until S is empty.

   PQ: **21 14 32 10 44 8 2 11 20 26**

3. Next, each time we remove an element from the Priority Queue, we scan the whole unsorted PQ sequence to find and remove the minimum element. This element is then inserted back into the end of the original sequence S. This is repeated until the PQ is empty.

   Final S: **2**
   2 8
   2 8 10
   2 8 10 11
   2 8 10 11 14
   2 8 10 11 14
   2 8 10 11 14 20
   2 8 10 11 14 20 21
   2 8 10 11 14 20 21 26
   2 8 10 11 14 20 21 26 32
   2 8 10 11 14 20 21 26 32 44

B.) ____

1. Initial S: **21 14 32 10 44 8 2 11 20 26**

2. First, we remove each first element from the initial sequence S and insert it into the sorted Priority Queue. Each insert, we have to scan the elements currently within the PQ to find out where to insert the new element according to order. This is repeated until S is empty.

   PQ: **21**
   14 21
   14 21 32
   10 14 21 32
   10 14 21 32 44
   8 10 14 21 32 44
   2 8 10 14 21 32 44
   2 8 10 11 14 21 32 44
   2 8 10 11 14 20 21 32 44
   2 8 10 11 14 20 21 26 32 44

3. Next, we remove each first element from the sorted Priority Queue and insert it back into the original sequence S. This repeats until the PQ is empty.

   Final S: **2 8 10 11 14 20 21 26 32 44**

Problem #6

A.) ____

The following assumes:

- A function key(***pair***) that returns the key of a given pair and runs in O(1) time.
- All keys are numbers.

Pseudocode

```
Algorithm countSort(S)
     for all currentPair in S
          c ← 0
          for all pair in S
               if ( key(currentPair) > key(pair) )
                    c = c + 1
          sNew.insertAtRank(c, currentPair)
     return sNew
```

B.) ____

Since every key (the first for all loop) is compared against every key (the second for all loop), there are a total of n^2 comparisons for n elements in S.

All operations, except for insertAtRank(), run in O(1) time. Therefore, since the insertAtRank() operation is only inside the first for-each loop, the running time of the algorithm is O(n^2).

C.) ____

If the keys have the potential to have the same value, then the algorithm can be modified to handle this by changing the comparison that occurs between the keys of currentPair and all other pairs in S. The greater-than operator used in the pseudocode above would have to become a greater-than-or-equal-to operator. Additionally, a check would need to be included to ensure that the comparison of a key with itself is not included in the final count of c.

This will result in sNew having any pairs with the same key next to each other. These pairs' order in sNew will correlate with their order in the original sequence S.