Exercise 3.2

Algorithm: recursiveSearch (node, key)

1.

2. If tree is empty
   return null
   end if
   If node.key = key
   return node
   else if key < node.key
   If node.left != null
   return search (node.left, key)
   end if
   else
   if node.right != null
   return search (node.right, key)
   end if
   end else

Algorithm: iterativeSearch (Array[], int data)

1. left = 0
2. right = size - 1
   while left <= right
   mid = left + right - 1
   if data = Array[mid]
   return A[mid]
   end if
   else if data > A[mid]
   left = mid + 1
   end
   else
   right = mid - 1
   end
   end while

Exercise 3.6

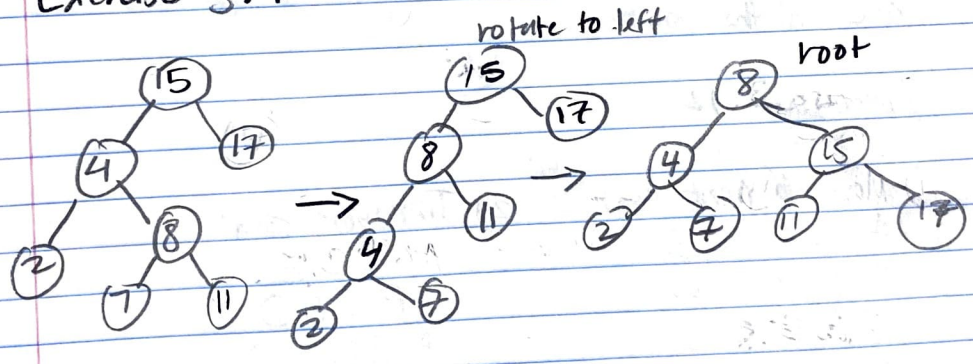
If $h = 0$ $\log_2(1) = 0$
If $h = 1$ $n = 3$ $\log_2 3 \approx 1$
if $h = 2$ $n = 7$ $\log_2 7 = 2$
If $h = 3$ $n = 15$ $\log_2 15 \approx 3$

} Binary

If $h = 1$ $n = 4$ $\log_3 4 \approx 1$
If $h = 2$ $n = 13$ $\log_3 13 \approx 2$

A balanced ternary would be $O(\log_3 n)$ because at each height the tree is being divided by 3.

Exercise 3.7



Exercise 3.8

1) This is true because that is the max that each node can have so if each node has $m-1$ keys then we know that the tree is full and therefore its max height can then be obtained

2) max height is $\log_m n$

3) $\log_{10}(10^9) = 9$

Exercise 3.10

The number of black nodes counted are the same for all leaves because the black alternates in the node. Each leave has an ~~sd~~ number of black keys. Every red nodes has a black parent because they are on the edge of the node with the black one being in the middle.

Exercise 3.11

The optimal order would be based on which letters are accessed the most so the ones with the higher frequencies are at the front

Exercise 3.12

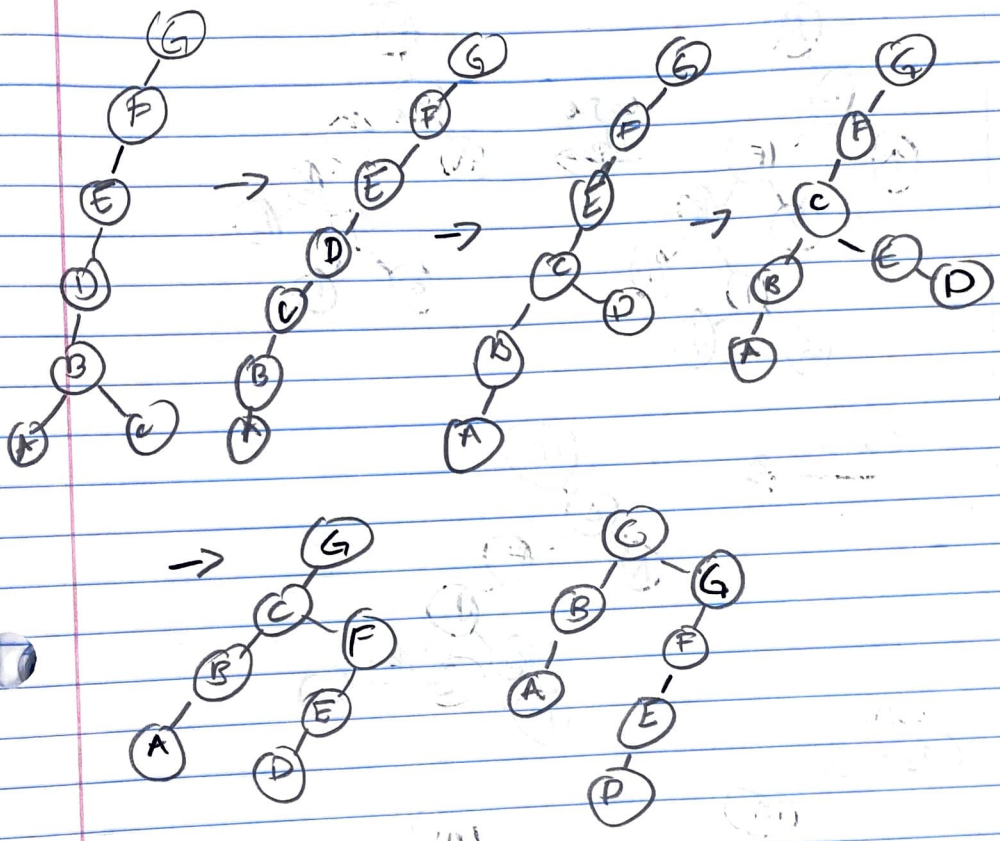
1) Move ~~to~~ front strat

A B C D E
E A B C D
E A B C D
D E A B C
E D A B C
B E D A C
E B D A C

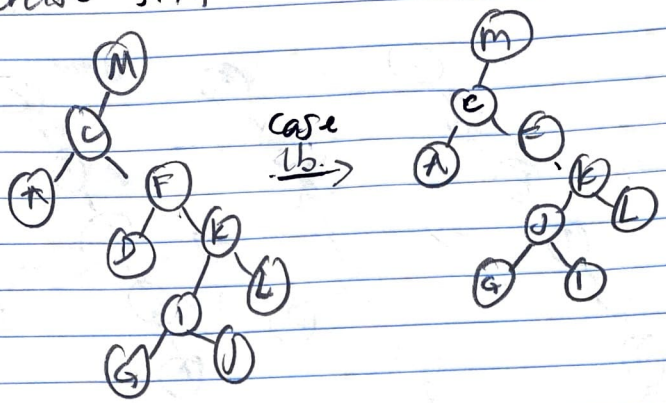
2) Transpose strat

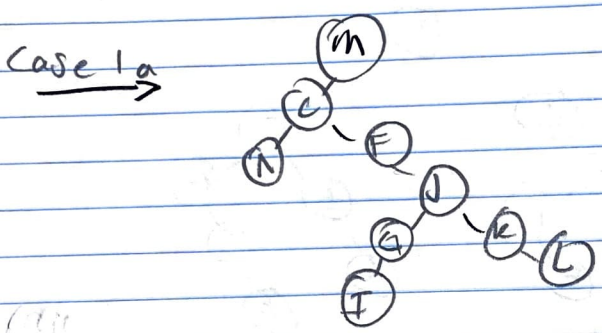
A B C D E
A B C E D
A B E C D
A B E D C
A E B D C
E A B D C
E B A D C
E B A D C

Exercise 3.13

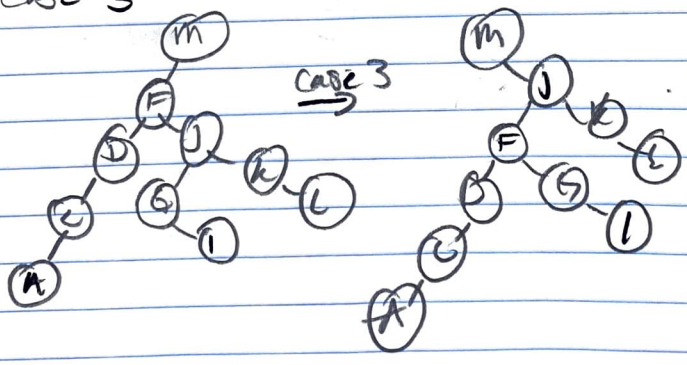


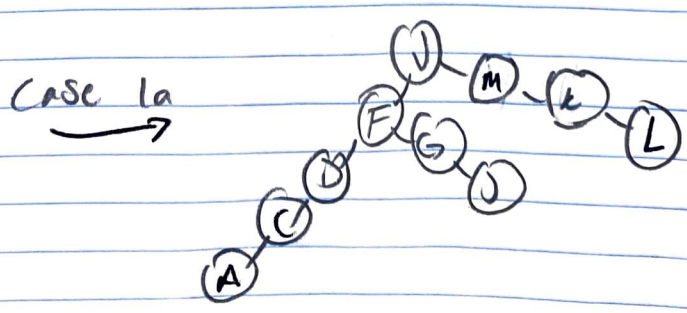
Exercise 3.14



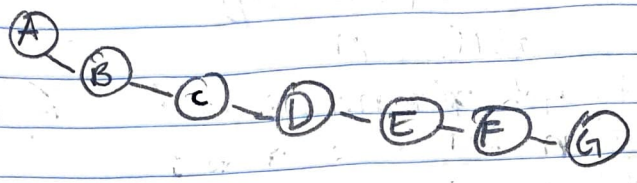


Case 3





Exercise 3.3



Exercise 3.4

```
Algorithm : IterativeMax (node n)
        n = root
        while n.right != null
              node = node.right
        end while
        return node
```

```
Algorithm : RecursiveMax (node n)
      base case  if (n.right = null)
                    return n
                 end
                    recursiveMax(n.right)
```

Exercise 3.5

Since the successor is all the way to the right its only child can be to the left