CS-2383 A3

1) Algorithm inOrderTraversal (Node rnode)

```
Stack stack;
b ← true;
while (b = true)
    while (rnode != null)
        stack.push(root);
        root = root.left;
    if(stack.isEmpty())
        return;
    rnode = stack.pop;
    visit(rnode);
    rnode = rnode.right;
```

2) a) visitAllNodes (level, root)

```
~~K~~;h
if(level =1)
    Order (level, root);
    return level+1;
goDown ← visitAllNodes (~~tet~~ level - 1, root);
Order (goDown, root);
return goDown + 1;
```

```
Algorithm Order (degree, rNode)
if (rNode = null)
    return;
if (degree = 1)
    visit (rNode)

else if (degree > 1)

    Order(degree - 1, rNode.left);
    Order(rNode.right, degree - 1);
```

$O(n^2)$

2)2)

```
Algorithm InOrderT (level, rNode)
Queue queue
if(rNod = null)
    return
queue.enqueue(rNode)
s <---- queue.size
while (s > 0)
    current <- queue.dequeue
    visit (current)
    if(current.hasleft())
        queue.enqueue(current.left)
    if(current.hasRight())
        queue.enqueue(current.right)
    s--
```

O(n)

3) Algorithm getLCA(root, v, w)
input: Nodes root, v, w,
if (root = null)
    return null;
if (root = v OR root = w)
    return root;
leftNode ← getLCA(root.left, v, w)

rightNode ← getLCA(root.right, v, w)
if(leftNode != null AND rightNode != null)
    return root;
if (rightNode != null)
    return rightNode;
else
    return leftNode;

O(n)

4) If element is the root node:
if root is the last element, then remove it.
if it is not (there are other elements.
change root to the last element and remove it
downheap Order from the root to the end.

If element is an intenal node
look for the element and save its index. Then substitute the last element with that element. Finally, when the element we are looking for is the last, remove it. Then find the depth of the removed element with downheap order starting from the depth of the (now substitute) element.

If the element is the external node:

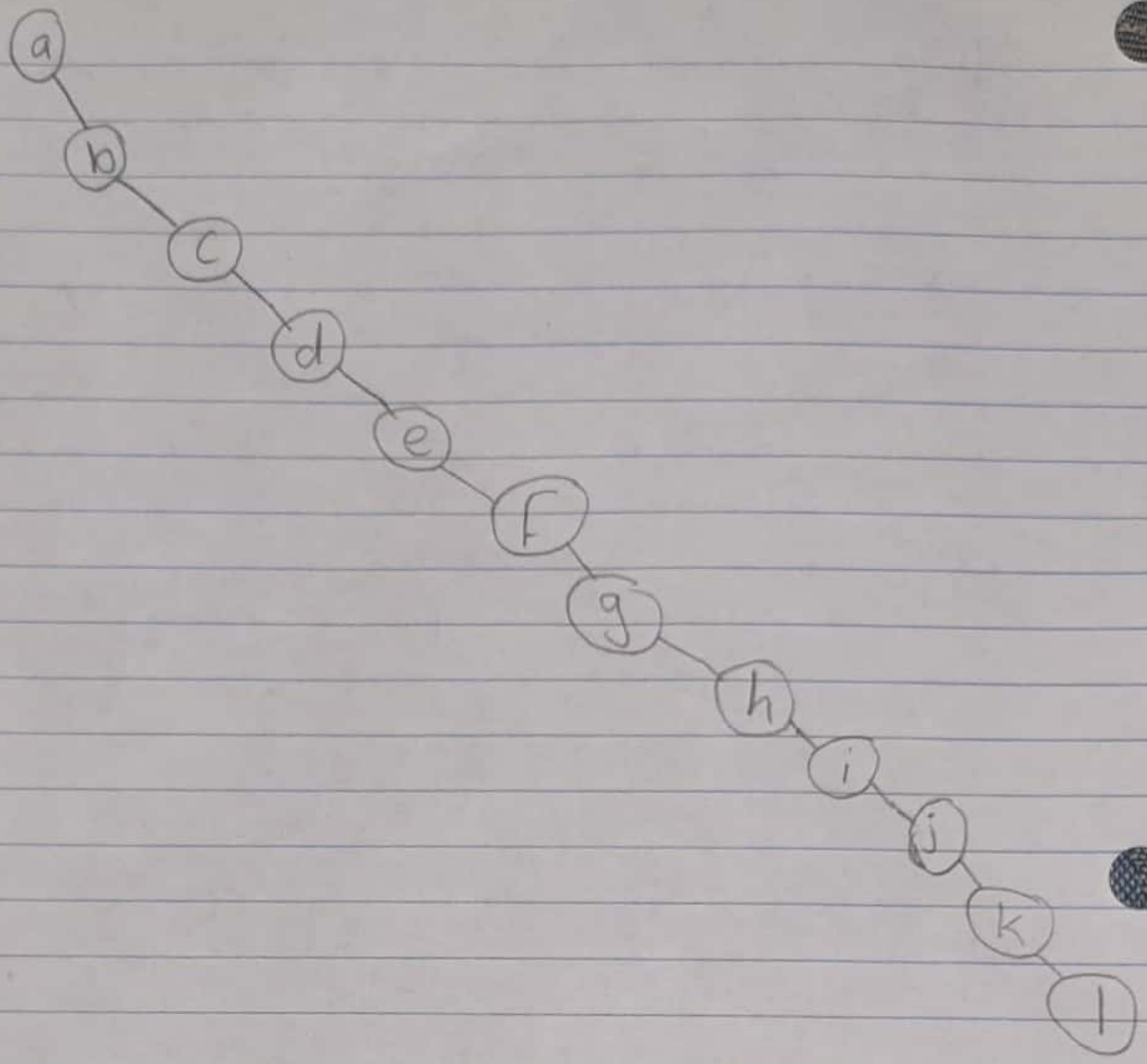
remove the arbitrary element (external node). If that external node is the last node: remove. Else, substitute that node with the last node. Then compare the substituted node with its parent. If child < parent then upheap order. If not, keep ~~the same it th~~ it as is.

5)

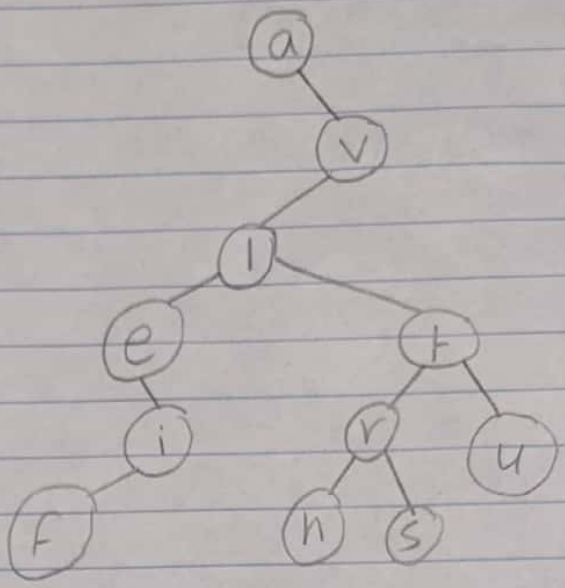
```
Algorithm compare (rNode, k)
if(rNode.key≤k OR rNode != null)
    print&rNode.key&;
    compare(rNode.left, k)
    compare (rNode.right, k)
```

O(k)

6) a)



6) b)



>> Algorithm findAll(k)

```
while (root != null)
    if (root = k)
        insertinGroup (k)
        root ← root.right
    elseif (root < k)
        root ← root.right
    else
        root ← root.left
return group
```

After putting the keys in a group, when we need to search for them, we only have to iterate to the correct node(s). Since this is done s times, and height is h, run time will be determined by h+s

$\therefore O(h+s)$