Name: Paul Zeng
Email: paulmiaozeng@gmail.com

1. a: a-b-c-d-e-f
1. b: a-b-d-h-e-i
1. c: a-a-b-c-a-b-d-e

2. a: best case is O(1), which is when the key is found at the root of the entire tree

worst case (balanced): O(n), where n is the number of nodes in the tree. Even though the tree is balanced and the depth of the tree is log(n), our algorithm still may need to look at all of the nodes, therefore its time complexity is O(n).

worst case (not balanced): O(n), where n is the number of nodes in the tree. In the worst case in which the tree is not balanced, the depth of the tree equals n-1, and our algorithm may need to traverse every node in the tree therefore its time complexity is O(n)..

2. b:

```
    private static int depthInTree(int key, Node root) {
        if (key == root.key)
            return 0;
        if (key < root.key && root.left != null) {
            int depthInLeft = depthInTree(key, root.left);
            if (depthInLeft != -1)
                return depthInLeft + 1;
        }
        if (key > root.key && root.right != null) {
            int depthInRight = depthInTree(key, root.right);
            if (depthInRight != -1)
                return depthInRight + 1;
        }
        return -1;
    }
```

2. c: best case is O(1), which is when the key is found at the root of the entire tree

worst case (balanced): O(log(n)), where n is the number of nodes in the tree. When the tree is balanced, whenever our function makes a recursive call, half of the subtree is thrown away. Our algorithm ends up looking at O(log(n)) nodes.

worst case (not balanced): O(n), where n is the number of nodes in the tree. When the tree is not balanced, in the worst case, the depth of the tree equals n-1, and our algorithm may traverse every node in the tree.

3. a:

```
          P
        /   \
       S     R
       \    / \
        B  J   M
       /  /   / \
      K  C   D   E
```

3. b:

```
           L
      /        \
     O          S
      \        / \
       G      H   N
      / \    /
     I   B  K
           /
          Z
```

4. a: '|_|' represents internal nodes.

```
          |_|
        0/   \1
        E   |_|
           0/ \1
         |_|   |_|
        0/ \1 0/ \1
        F  C  O  20
```

4. b:110 100 100 111 101 0 (space are inserted to help human reading only)

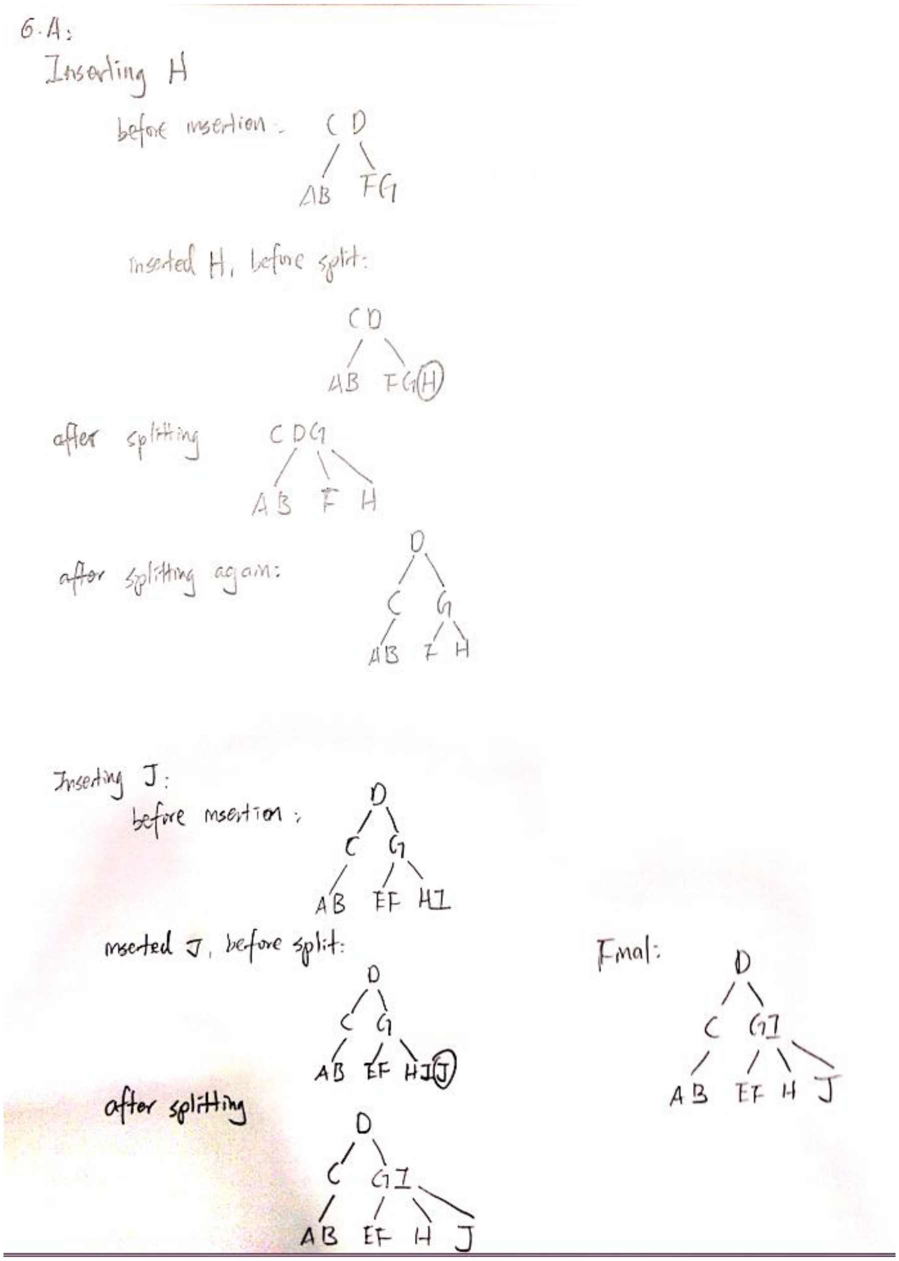
5. a: 44-35-23-28-53-48-62-57-80
5. b: 28-23-35-48-57-80-62-53-44
5. c:

```
            44
          /    \
        35      53
        /      /  \
       23   48   62
         \   \   / \
         28  51 57 80
         /
        25
```

5. d:

```
           44
          /  \
        23   48
          \   \
          28  62
              / \
            57 80
```

5. e: Yes, this tree is balanced because for each node, its subtrees either have the same height or their heights differ by 1.



6.A:

Inserting H

before insertion:

inserted H, before split:

after splitting

after splitting again:

Inserting J:

before insertion:

inserted J, before split:

after splitting

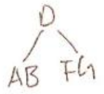
Final:

6.B. Inserting F:

before insertion: ABDG

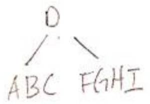
inserted F, before split: ABDFG

after split:

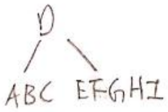


Inserting E:

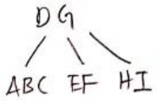
before insertion:



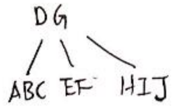
inserted E, before split:



after split:



Final


DG
ABC EF HIJ