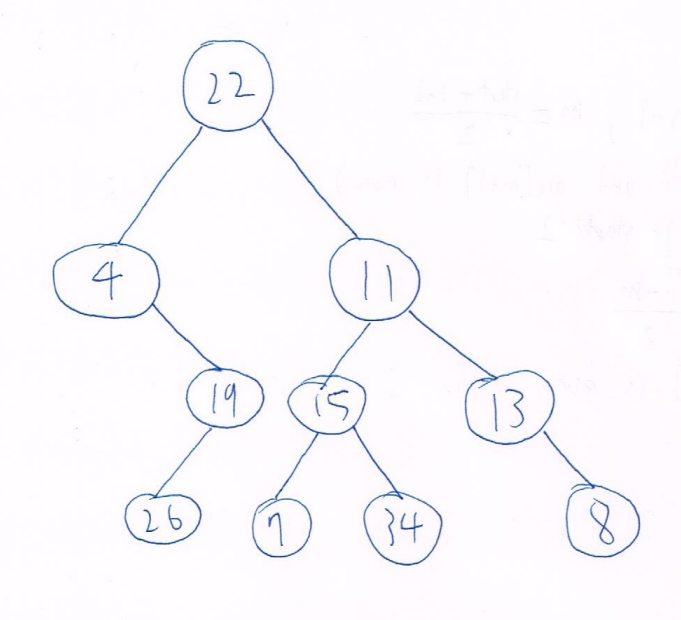
DSA HW-2

Problem 0:

1. 李沛宸 B10902032
2. 李沛宸 B10902032
3. 李沛宸 B10902032
4. 李沛宸 B10902032
5. 李沛宸 B10902032

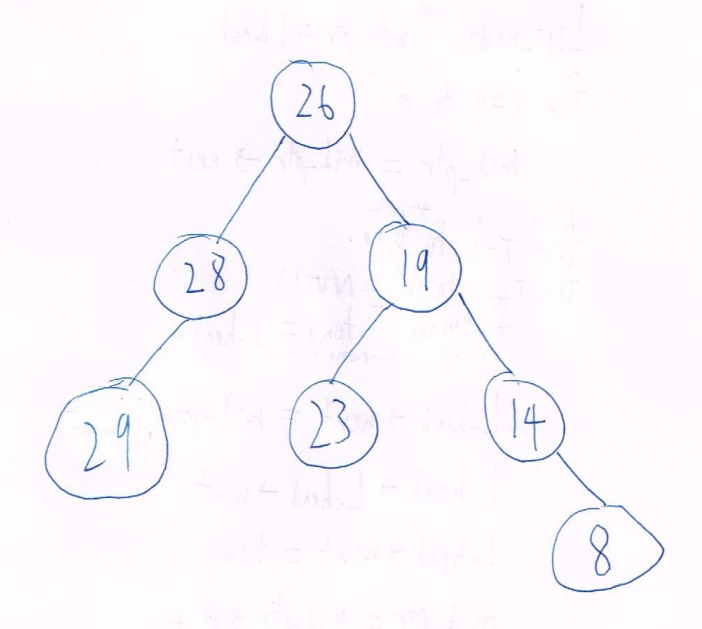
Problem 1:

1. Tree:



   Human algorithm: In a postorder traversal it's last node in that subtree will be it's root, so we can cut the tree in half by looking at the inorder traversal, cause the nodes at the left side of the root are in the left subtree and right side in the right subtree. By doing this we can reconstruct the tree.

2. Tree:



3. Algorithm:

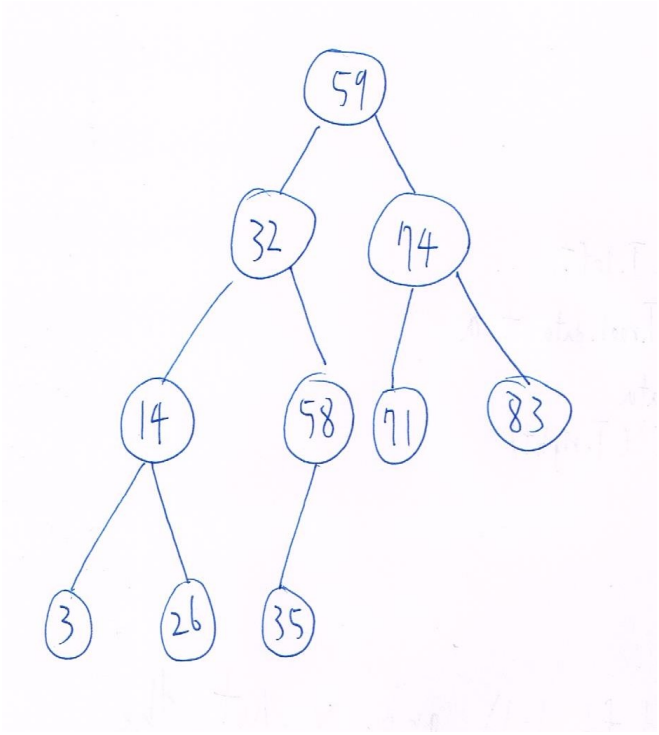
```
int a = 0
modify_T(root):
    if root is nullptr:
        return
    else:
        modify_T(root->right)
        root->data += a
        a = root->data
        modify_T(root->left)
```

The time complexity is O(n) cause it traverse through n node with O(1) operation to the node.
The algorithm works by traversing from the largest node to the smallest node, and adjust the data in the node by adding the sum of the value of every node that has been traversed, which is the nodes that are larger than it.

4. Prove:

    1. Let x be a leaf node and y's left child, assume z is a node smaller than y then z must be in y's left subtree, however x is the only node in y's left subtree, so by contradict y must be the smallest node among all nodes larger than x
    2. Let x be a leaf node and y's right child, assume z is a node larger than y then z must be in y's right subtree, however x is the only node in y's right subtree, so by contradict must be the largest node among all nodes smaller than x
    3. By the 2 fact above we can prove that y is either the smallest node among all nodes larger than x, or the largest node among all nodes smaller than x, when x is a leaf node and y is its parent.

5. Tree:



6. Algorithm:

```
height(root):
    if root is nullptr:
        return 0
    else
        return 1 + max(height(root->left), height(root->left))
wasted_positions_num = pow(2, height(root)) - 1 - n
```

The time complexity is O(n) cause the recursive fuction traverse through n nodes with O(1) operations during the traverse
The algorithm works cause we know that the number of nodes in a complete binary tree are 2^height - 1, so the wated positions in an array will be 2^height - 1 - n

Problem 2:

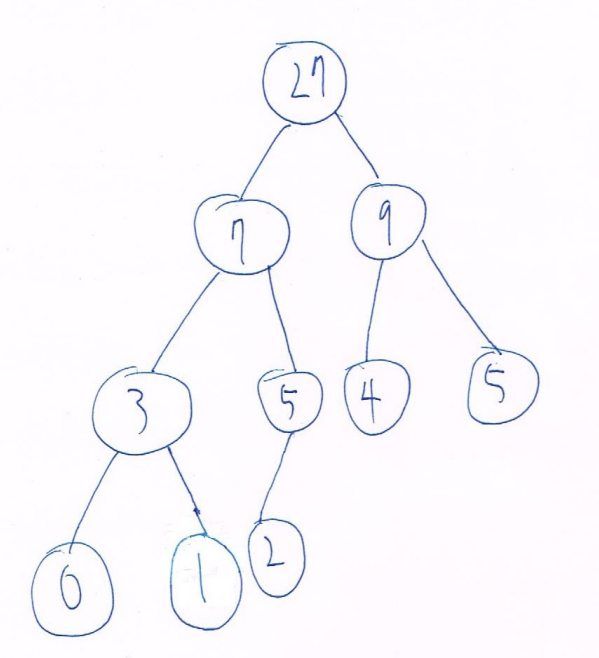
1. Algoritm:

```
find_max_min():
    stack = empty stack
    push(stack, 1)
    push(stack, 2)
    for i = 3 to i = n:
        push(stack, i)
        a = pop(stack), b = pop(stack), c = pop(stack)
        if query_attitude_value(a, b, c) == True:
            push(stack, a)
            push(stack, c)
        else if query_attitude_value(a, c, b) == True
            push(stack, a)
            push(stack, b)
        else
            push(stack, b)
            push(stack, c)
    max_min = {pop(stack), pop(stack)}
    return max_min
```

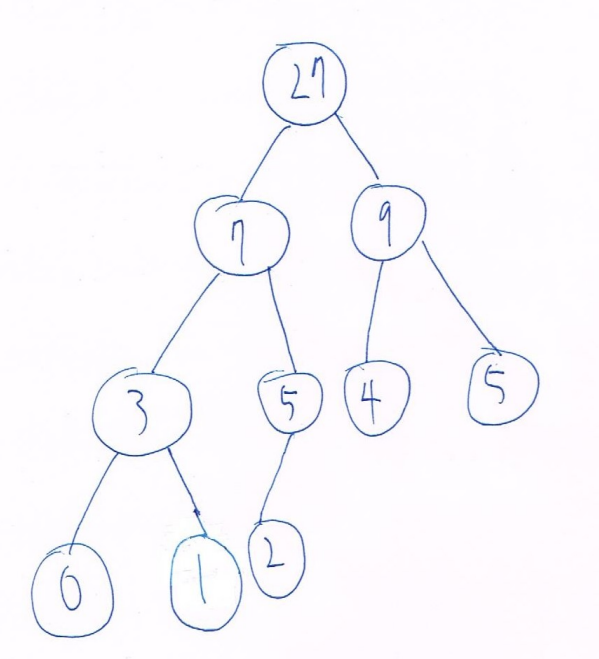
The time complexity is O(n) query complexity, cause the for loop interations are n-2, and each iteration needs O(1) query complexty

Prolem 3:

1. Tree:



2. Tree:



3. Algorithm:

```
find_max_index(heights, a, b):
    max = 0
    index = 0
    for i = a to b:
        if heights[i] > max
            max = heights[i]
            index = i
    return index
build_cartesian_tree(root, heights, a, b):
    if a == b:
        return
```

```
        else:
            index = find_max_index(heights, a, b)
            root->data = heights[index]
            build_cartesian_tree(root->left, heights, a, index-1)
            build_cartesian_tree(root->right, heights, index+1, b)
```

The time complexity is O(n * h) because to complete every floor of the tree it will take (n-c) times for searching the max index
Example for the worst case: