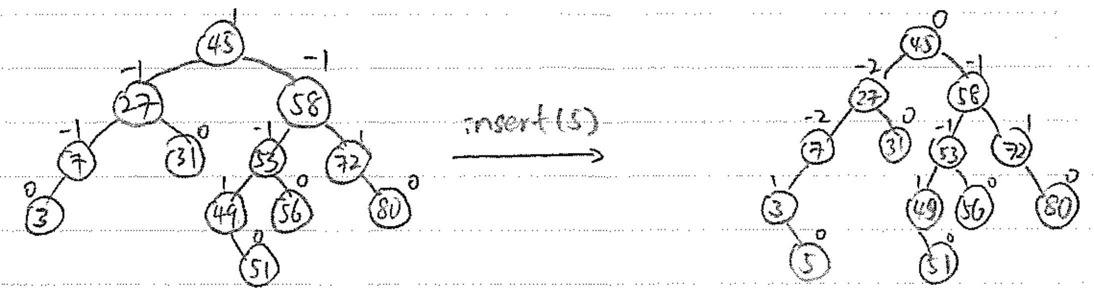
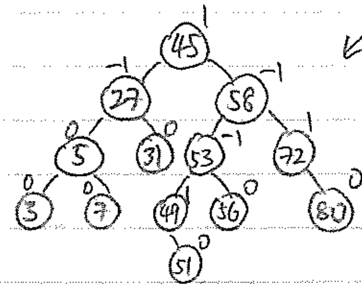
A3

Jianan Luo
20523403

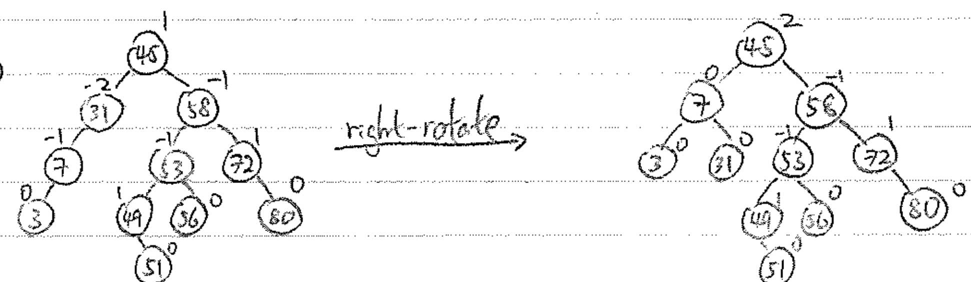
1. a)

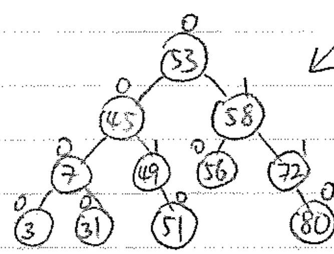




Final tree (double right rotate)

b)





Final tree (double left rotate)

c) Let N_h represent the least number of node needed to build a too tree has height h. Since the height of left and right subtree diff by at most 2. So the least number of node N_h can be write as $N_{h-1}+1+N_{h-3}$. Where N_{h-1} represent # of nodes of the higher subtree, and 1 represent the root and N_{h-3} represent the # of node of the lower subtree in the worst case. Then just by following the prove on the right, we could get $h \in O(\log n)$

$N_h = N_{h-1} + 1 + N_{h-3}$ ①

$N_{h-1} = N_{h-2} + 1 + N_{h-4}$ ②

$N_{h-3} = N_{h-4} + 1 + N_{h-6}$ ③

subsitute ② and ③ to ① we get.

$N_h = N_{h-2}+1+N_{h-4}+1+N_{h-4}+1+N_{h-6}$

$= N_{h-2} + 2N_{h-4} + N_{h-6} + 3$

$\because N_{h-2} + N_6 + 3 > 0$

$\therefore N_h > 2N_{h-4} > 2 \cdot 2N_{h-8} > \underbrace{2 \cdot 2 \cdots 2}_{\frac{h}{4}} N_{h-4n}$, where $N_{h-4n} = 1$, (just a leaf)

$N_h > 2^{\frac{h}{4}} \cdot 1$

$\log(N_h) > \frac{h}{4}$

$h < 4\log(N_h) \Rightarrow h \in O(\log n)$

d)

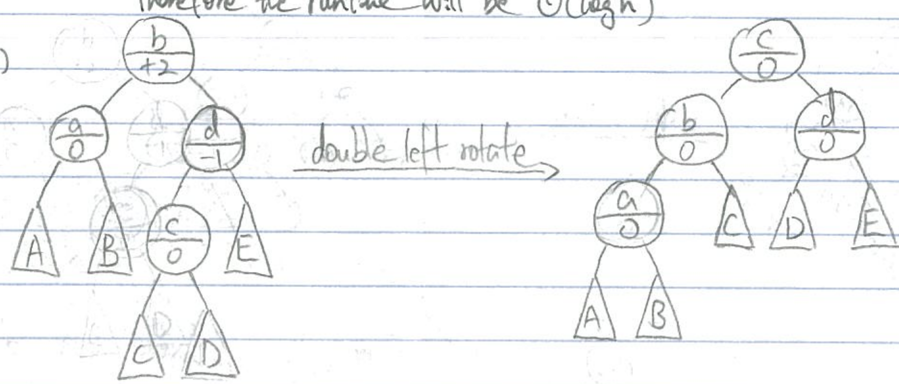
```
heightAVL (AVLtree T)
    if (T.left == nil && T.right == nil)
        return 0
    if (T.balance == -1)
        return (1 + heightAVL (T.left))
    else
        return (1 + heightAVL (T.right))
```

tyee.

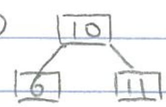
Prove: So everytime we check the balance factor, if leaf, return 0, ($O(1)$), else, we call recursion, and the node go down floor and ignore other half nodes ($\frac{n}{2}$). Therefore the runtime will be $O(\log n)$

So this algorithm measure the height of the AVL tree run in time $O(\log n)$. It checks if the current ~~not~~ node is a leaf, if it is, return ~~[illegible]~~ else, we need to check the balance factor, -1 means T.left is higher, 1 means T.right is hight, 0 means same height. So we just return 1 + heigher ~~sub~~ sub-tree. So finally, we get the height of the AVL tree by runtime $O(\log n)$

2. a)

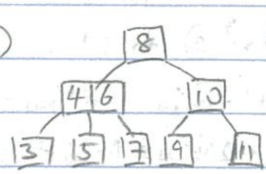


3. a)



b)

c)



d) $h = \lfloor \log_2(n+1) \rfloor - 1$

Proof: When we insert $1, 2, \ldots, n$. Every time ~~the~~ the 2 3 tree's height increase, it will split and become to a binary search tree BST,

So the ~~#~~ of node n, before height increase must follow

$$\begin{cases} n < 2^{h+2} - 1 \ ① \\ n \geq 2^{h+1} - 1 \ ② \end{cases}$$

①: $n < 2^{h+2} - 1$

$n+1 < 2^{h+2}$

$\log_2(n+1) < h+2$

$h > \log_2(n+1) - 2$

②: $n \geq 2^{h+1} - 1$

$n+1 \geq 2^{h+1}$

$\log_2(n+1) \geq h+1$

$h \leq \log_2(n+1) - 1$

So we get: $\log_2(n+1) - 2 < h \leq \log_2(n+1) - 1$

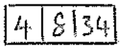
$\because h$ is an integer

$\therefore h = \log_2(n+1) - 1$

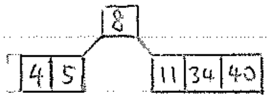
Therefore proved.

A3

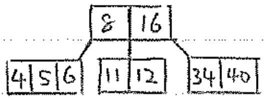
4. a) insert 34, 4, 8



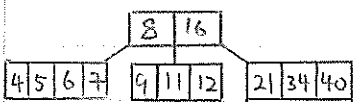
insert 5, 40, 11



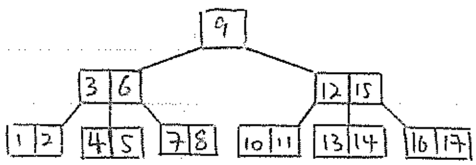
insert 6, 12, 16



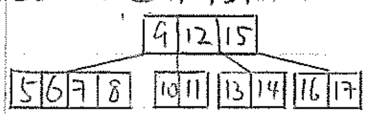
insert 21, 7, 9



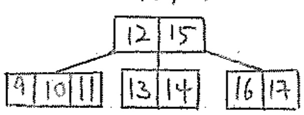
b)



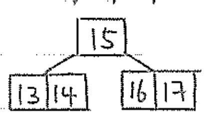
c) remove 1, 2, 3, 4



remove 5, 6, 7, 8



remove 9, 10, 11, 12



remove 13, 14, 15, 16

remove 17

[]

5. preprocessing algorithm:

```
B[k], C[n]
for i from 0 to n-1
    B[A[i]] += 1
for j from 0 to k-1
    C[j] = C[j-1] + B[j]  // when j=0, C[j-1]=(
```

algorithm:

```
return (C[b] - C[a-1])  // when a=0, C[a-1]=(
```

★ Runtime of preprocessing algorithm: $O(n+k)$
Runtime of algorithm: $O(1)$

// Preprocessing algorithm

This algorithm is similar to counting sort. So first we allocate two arrays, B[k] and C[n]. Then we count each occurrence, A[i] and add to its relevant position which is B[A[i]]. For example, if number 3 occurs 4 time, B[3] should equal to 4. Then we calculate the number of elements less or equal to j from 0 to k-1, and store the number to C[j] by add C[j-1] and B[j]. (C[j-1] is the # of elements less than k, B[j] is the # of elements equal to k.)

// Algorithm

When we try to find the # of the integers in the range [a, b], which means $a \le int \le b$. So we just need to subtract C[a-1] from C[b]. (C[a-1] is the # of elements less than a, C[b] is the # of elements less or equals to b.)

Justification for runtime:

- from the pseudocode, we can see that runtime for first "for loop" is $O(n)$. And the runtime for second "for loop" is $O(k)$. So the preprocessing algorithm is $O(n+k)$.
- from the pseudocode, we can see that the runtime for return is just $O(1)$.

∴ Proved.