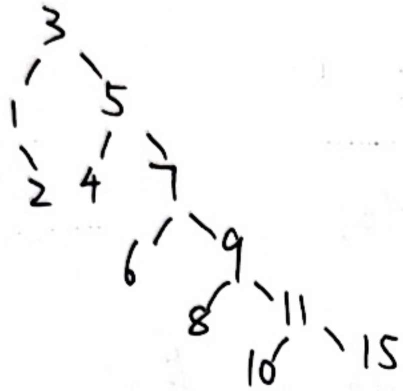
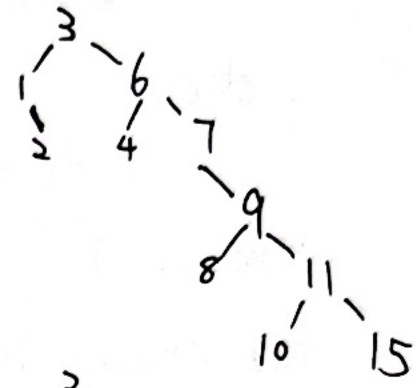
Assignment 3 Hongdao Meng hm3424

Q1a. 3, 1, 5, 4, 7, 9, 11, 15, 2, 6, 8, 10.

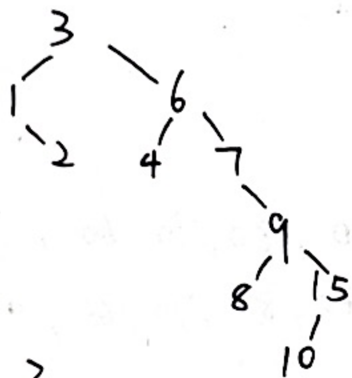
BST :



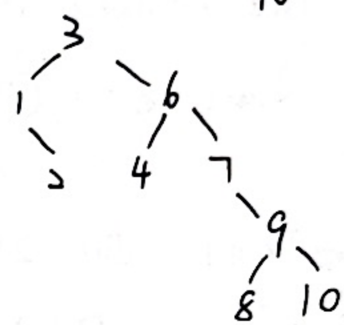
Delete 5 :



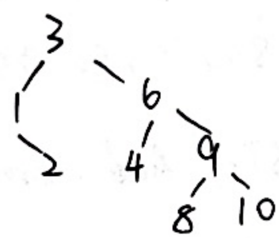
Delete 11 :



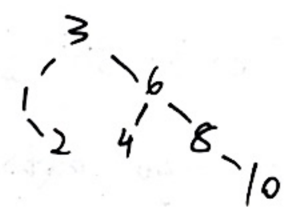
Delete 15 :



Delete 7 :

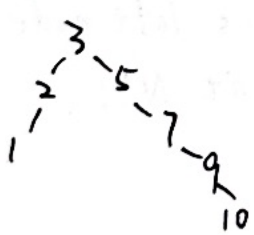


Delete 9 :



Q2b. Insert order : 3, 5, 2, 7, 1, 9, 10.

BST :



Insert	comparasons
3	0
5	1 (with3)
2	1 (with3)
7	2 (3→5→7)
1	2 (3→2→1)
9	3 (3→5→7→9)
10	4 (3→5→7→9→10)

total = 0+1+1+2+2+3+4=13

QuickSort:

Pivot=3 left [2, 1] right [5, 7, 9, 10]

comparisons = 6

[2, 1] Pivot=2 left [1] right [] comparison=1

[5, 7, 9, 10] Pivot=5

left [ ] right [7, 9, 10] compare=3

Pivot=7 left [ ] right [9, 10] C=2

Pivot=9 left [ ] right [10] C=1

total = 6+1+3+2+1 = 13

The total comparison times of BST and QuickSort are both 13, and the process of the two correspond one by one.

Q1c.

```
Delete-Min(T):
    if T is NIL:
        return NIL
    elif T.left is NIL:
        return T.right
    else:
        T.left = Delete-Min(T.left)
    return T.
```

The Minkey is the left most node of ~~B~~BST.

Runtime : O(h)

Q1d.

PrintTree1 : 50, 25, 12, 6, 35, 30, 40, 80, 70, 60, 95

PrintTree2 : 50, 25, 12, 6, 35, 30, 40, 80, 70, 60, 95

PrintTree3 : 50, 25, 12, 6, 35, 30, 40, 80, 70, 60, 95

All three Algo output same on all BSTs. And three procedures are equivalent.

Because they follow the same print conditions and cover the same node path (dfs)

print as long as left node isn't NIL.

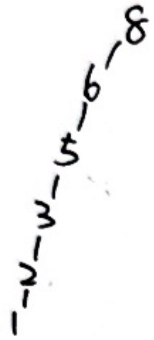
Q2a.

Inorder : left → root → right

postorder : left → right → root

If the BST without right , the traversal output will be same.

BST :



Q2b.

```
Find Depth (x):
    If x.parent is NIL:
        return 0
    else:
        return 1+ Find Depth (x. parent)
```

Runtime : $O(h)$

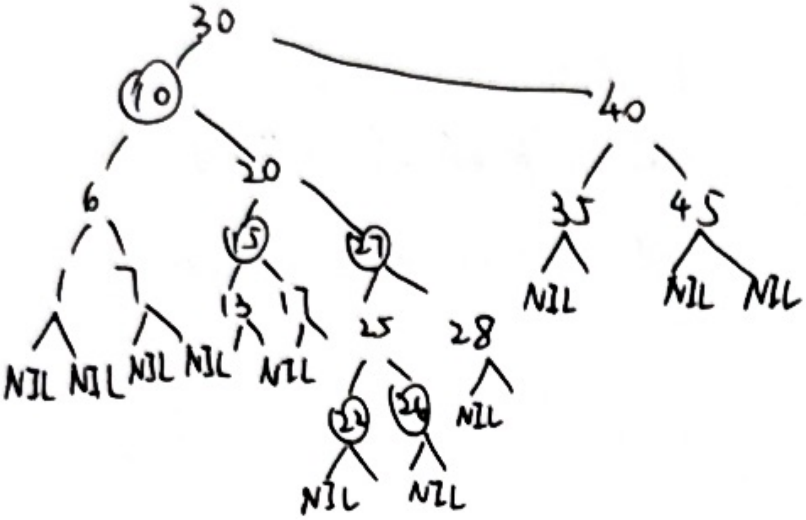
Q2C.

```
Find Ancestor (T, x, y):
    current_val = T. key
    x_val       = x. key
    y_val       = y. key

    if x_val < current_val and y_val < current_val:
        return Find Ancestor(T. left, x, y)
    elif x_val > current_val and y_val > current_val:
        return Find Ancestor(T. right, x, y)

    else:
        return T
```

Q2d: pre order: root -> left -> right

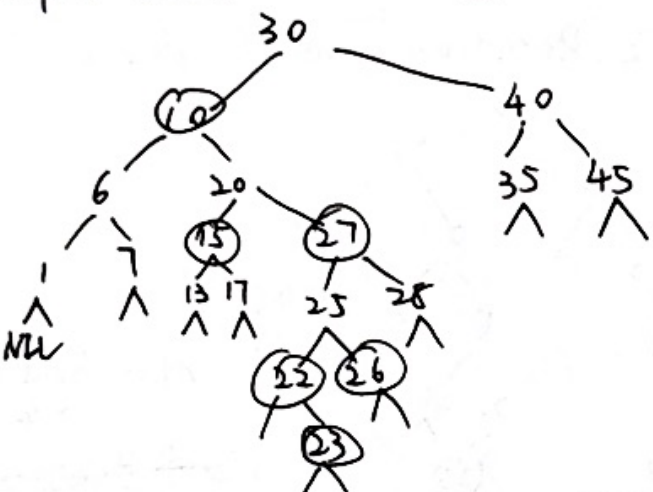
```
RecreateTree(S):
    If S.isEmpty():
        return NIL
    val = S.pop()
    If val == '0':
        return NIL
    node = newNode()
    node.left = RecreateTree(S)
    node.right = RecreateTree(S)
    return node
```

Q3a. max black-height = 3



Insert 23:

step 1. Insert as red 23 initial insertion:



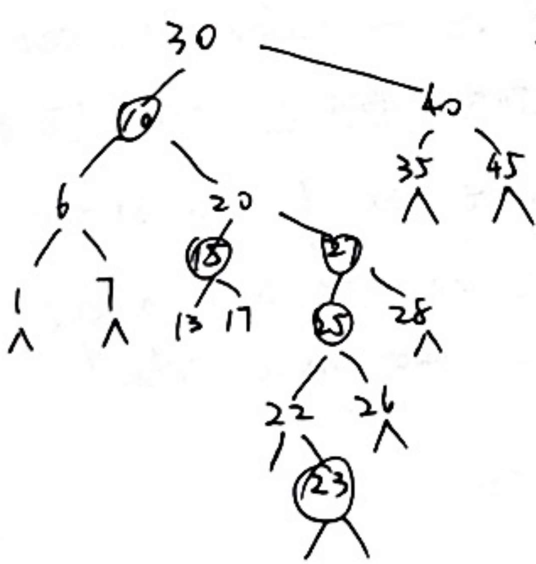
step 2. RB-repair case: parent is Red and uncle is Red too.

22 26

∴ Recolor: Uncle and parent to Black

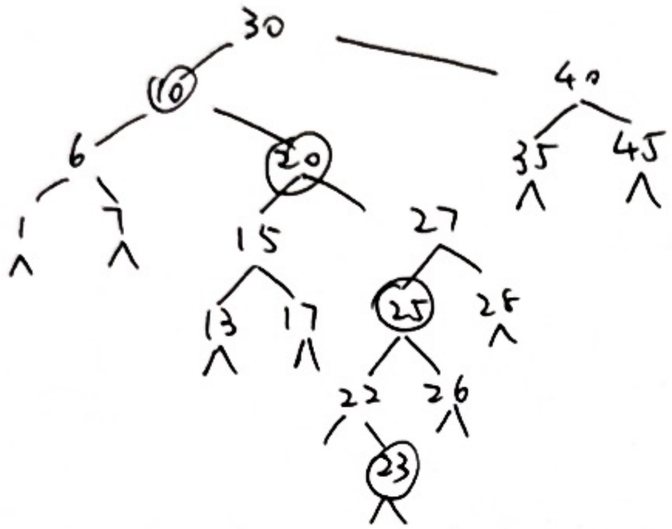
Grandparent to Red

25



Step 3. Call RB-repair for node Grandparent 25

Case : parent and uncle both red → Recolor



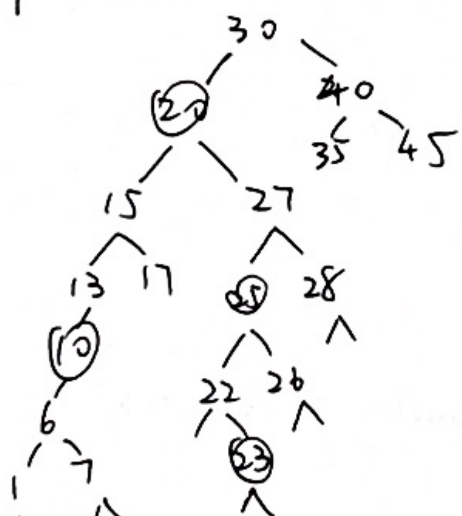
Grandpa 20 → Red

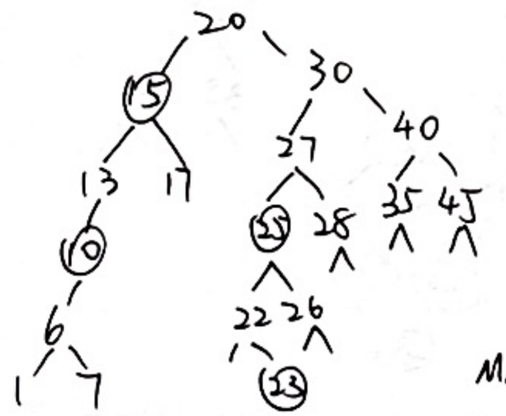
uncle 15 parent 27 → Black

Call RB-repair for 20.

Step 4. RB-repair for 20   Case : parent 10 is RED, Uncle 6 is Black

Step 4.1 Bent to straight type.   Step 4.2 Rotation and Recolor





Max Black Height = 4

~~Max B-Height = 4~~

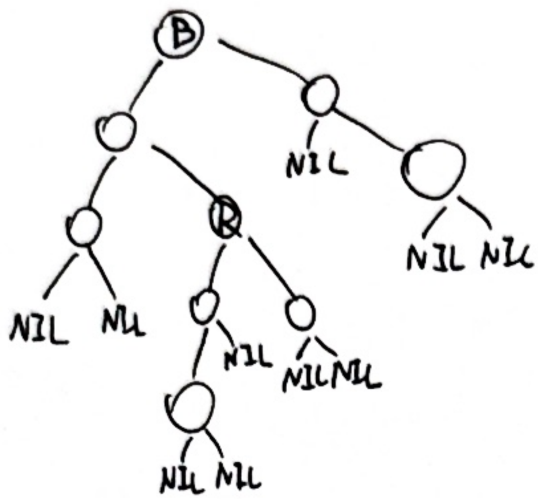
~~Doesn't change~~.

~~After insert 23, although the process of rotation and recolor, these operation only adjust the position and don't affect total number of black nodes.~~

black height changed, because 23 insert at the leaf of tree, which cause red-red conflicts propagate upward and affect the root node. And some new black ~~root~~ node added.

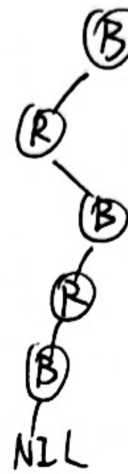
another node like 24 can do that same,

Q3b. Black-height (root) = max (3)



The max black height from root is 3

But



this path has at lease 3 black nod

root.left = Red will cause the max BH to be 2.

∵ This tree has a significantly unbalanced structure.

∴ It can't be correctly recolor

Q3c.

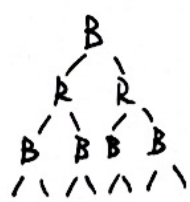
1. $2^{(bh*2)} - 1 = 2^{(4*2)} - 1 = 255$ the maximun num is 255

2. Impossible.
   If all black, tree must be completed Binary tree.
   num of nodes $= 2^n - 1 \,!= 60$

3. max num of nodes : $2^{(bh*2)} - 1 = 2^6 - 1 = 63 < 71$

   ∴ Impossible

4. bh=2 :



total 15 nodes

bh=3 All black completed tree $2^4 - 1 = 15$

bh=4 impossible at least need $2^5 - 1 = 31$ nodes.

black height can be 2 or 3.

Q4a.

FindLast(T):

~~Cur = T~~

If T == NIL:

return NIL

If T.right != NIL and T.right.max = T.max:

return FindLast(T.right)

If T.end == T.max:

return T

return FindLast(T.left)

Q4b. It's impossible.

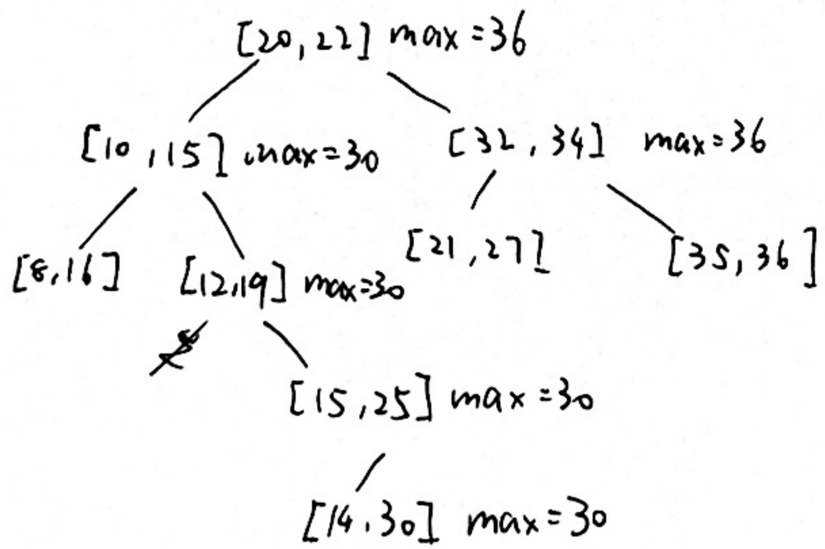
If interval i need to return [21,27], interval i's start > 30.

If i start < 30, it'll return [14,30]. Bcs root.left.max = 30.

And i can't overlap with [32,34].

If i start with a > 30, i can't overlap with [21,27].

∴ It's impossible for Interval_seach(i) returns node [21,27].



the start of i < 30.

the root.left is searche

start of i > 30.

Can't return [21,27]

Q4C. Maxright(T, k):

```
if T is NIL:
    return -infinity

if T.start >= k:
    left_max = Maxright (T.left, k)
    right_max = Maxright (T.right, k)
    return max (T.end, left_max, right_max)
else:
    return Maxright (T.right, k)
```

Runtime:

Because each node is accessed at most once, and recursion is linear with the number of nodes.

$\therefore O(n)$

Q5a.

```
CostAfter(T, k):
    if T is NIL:
        return 0
    if T.start > k:
        return T.budget + (T.right.btotal if T.right else 0)
                 + CostAfter(T.left, k)
    else:
        return CostAfter(T.right, k)
```

Each recursion selects only left or right subtree. The path length and tree height is linear.

Each operation of node are $O(1)$, ∴ Runtime: $O(h)$.

Q5b.

```
TotalBudget(T, k)      inorder
    if T is NIL or k<=1:
        return 0
    left_size = T.left.size if T.left else 0
    RankofT = left_size + 1

    if RankofT == k:
        return T.left.btotal if T.left else 0
    elif RankofT > k:
        return TotalBudget(T.left, k)
    else:
        sumLeft = T.left.btotal if T.left else 0
        remain = k - RankofT - 1
        return sumLeft + T.budget + TotalBudget(T.right, remain
```

Each recursion moves one layer in the direction of the tree height, and path length doesn't exceed tree's ~~length~~ height.

Each operation is $O(1)$ ∴ Runtime : $O(h)$