Home Work - 6

Name: Sai Rohit Kalyan Gandham.
StudentID: ~~1002070024~~ 1002070724
EmailID: sxg0724@mavs.uta.edu

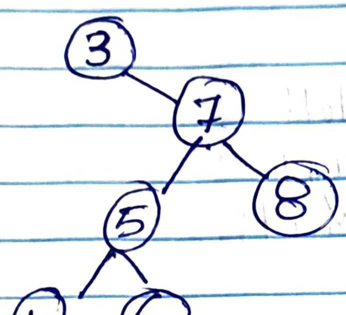
12.2-3 pesudo Code for the Tree-predecessor procedure.
Sol) C-code

```
Struct tree_t {
        Struct tree_t * left;
        Struct tree_t * right;
        Struct tree_t * parent;
         int key; };
typedef Struct treet tree_t;

 tree_t *maximum (tree_t *tree) {
   While (tree → right) tree = tree → right
   return tree; }
tree_t * predecessor (tree_t * tree) {
   if (tree → left) {
         return maximum(tree-left); }
   While ( parent && parent → left = tree) {
        tree = tree → parent;
        parent = tree → parent; }

   return parent; }
```

12.2.4 Given is BST (Binary Search Tree)
Sol) * Suppose that the Search for keys "k" in a Binary Search Tree ends up in a leaf. Consider three sets: A keys to the left of the Search path; B, the keys on the Search path; and C, the keys to the right of

the Search path.

* Here, we assume that the Search ended at 4,
So, A = {4}, B = {3, 7, 5, 6} and
C = {8}



12.31
Sol)

```c
#include <stdlib.h>

Struct node_t {
    Struct node_t *parent;
    Struct node_t *left;
    Struct node_t *right;
    int key; };

typedef Struct node_t node_t;

typedef Struct {
    node_t *root
  } tree_t;

tree_t *make_tree() {
    tree_t *tree = malloc(Sizeof(tree_t));
```

```
    tree -> root = NULL;
    return tree; }
node_t *insert_node (node_t *node, int key) {
    if (node -> key < key)
        if ( node -> right) {
         return   insert_node (node -> right, key); }
        Else {
            node_t *new = make_node(key);
            new -> parent = node;
            node -> right = new;
            return new; }
    } Else {
            node_t *new = make_node (key);
              new -> parent = node;
              node -> left = new;
                return new; } } }
    node_t *Search (tree_t *tree, int key) {
        node_t *node = tree -> root;
    While (node) {
            if (node -> key = key) {
                return node; }
        Else if (node -> key < key) {
                node = node -> right; }
        Else {
                node = node -> left; } }
        return Null; } }
```

12-1 (a) * When inserting items with identical keys the Boolean Calue at the line 5 of Tree insert is always false and so the right child will always be Chosen, Because Boolean Clause at line 11 is also false new node will be inserted as a right child of right most node.

* after inserting (n) nodes to the tree, the height of the tree will be n and no node will have a left child

* Inserting (n) items into an initailly empty binary search tree will cost because the height of tree increases at every insertion and new element is inserted as a new leaf node.

$$\sum_{i=1}^{n} i \in \Theta(n^2)$$

* This is asymptotic performance of inserting n items initially empty binary search tree.

b) * Building binary tree like with height $\Theta(\log n)$

* This can be seen when start to insert elements into an intially empty BST.

* let's tal, boolean flag values 0 set "x" to left[x] and value 1 set x to right[x]

* Boolean flag of node will be changed after visiting that node.

* 4^{th} element will go as a left child of right child of root node, etc.

* inserting 7 items into an intaially empty BST

* This strategy will result in each of two childeren subtree having a difference in size at most one

$$\sum_{i=1}^{n} \log n \in \Theta(n \log(n))$$

c) This will only take linear time since the tree itself will be height 0, and a single insertion into a list can be done in constant time.

d) Worst Case:

Every random choice is to the right (or all the left) & this will result in the same behaviour as in the first part of this problem, $O(n^2)$

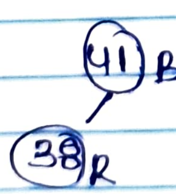
Expected Running time:

* Note that when randomly choosing, we will pick left roughly half the time, so the tree will be roughly balanced, that's why we have that depth is roughly $\log(n)$ $O(n\log n)$

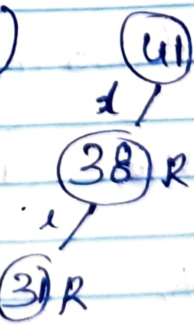
13.3.2 Red-Black Tree: 41, 38, 31, 12, 19, 8 * Here Suffix (B) = Black
Suffix (R) : Red.

Step ① insert "41" 41 B

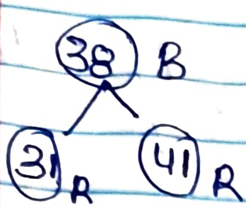
②



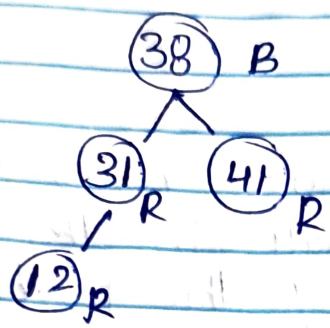
③



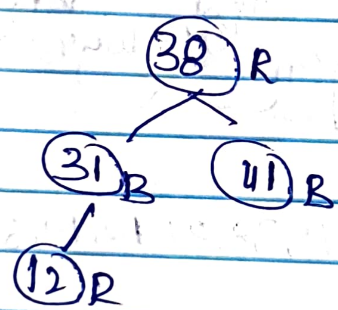
in step ③ : Node and parent node both are red and parent of the parent is root doing and need to do $l \to$ right rotation and change color.



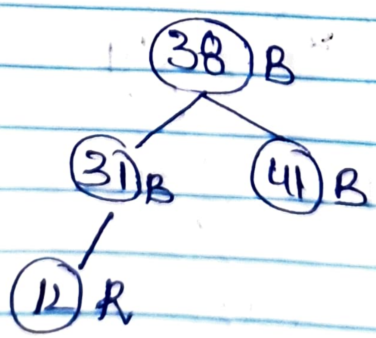
Step (4):



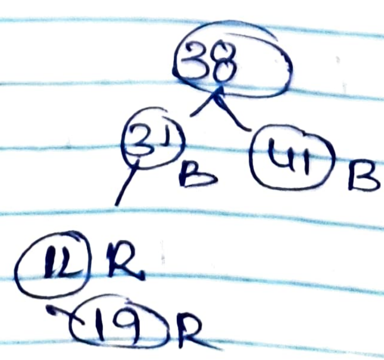
- again two adjecent red nodes and parent of parent node is root node do rotation and change color and check again.



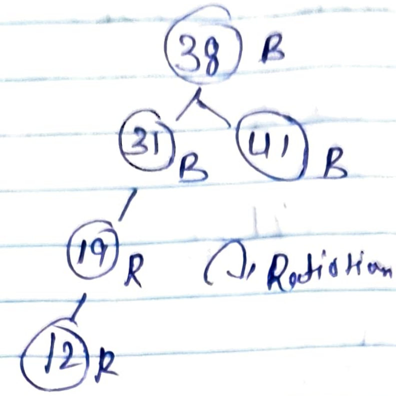
- Here root should be Black change again



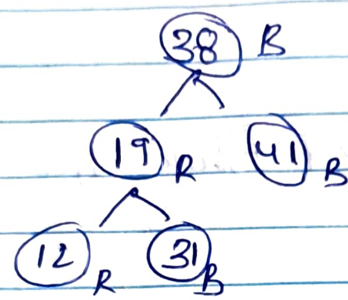
Step (5)



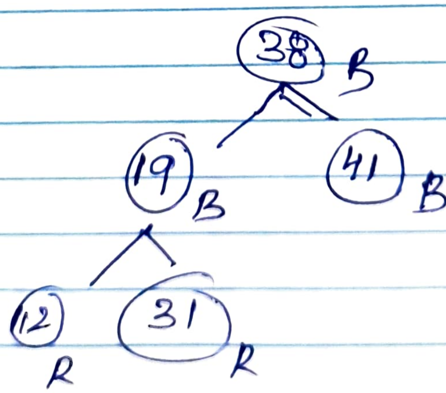
* again two adjacent red node's So do rotation first



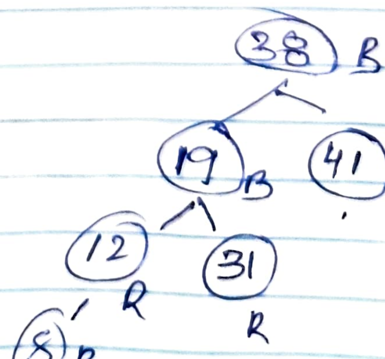
=>



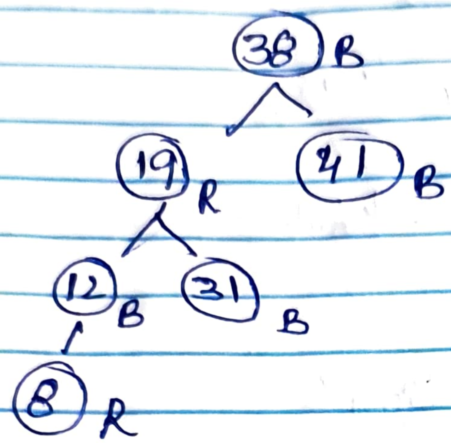
* Change color then finaly it will be



Step 6 : insert 8



* again two adjacent red node and parent of parent node is not root node so change the parent node color's



* final Tree of Red-Black Tree is following.