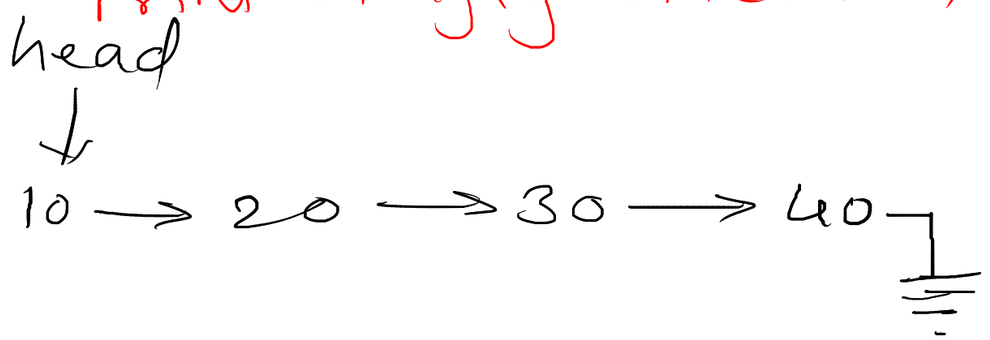
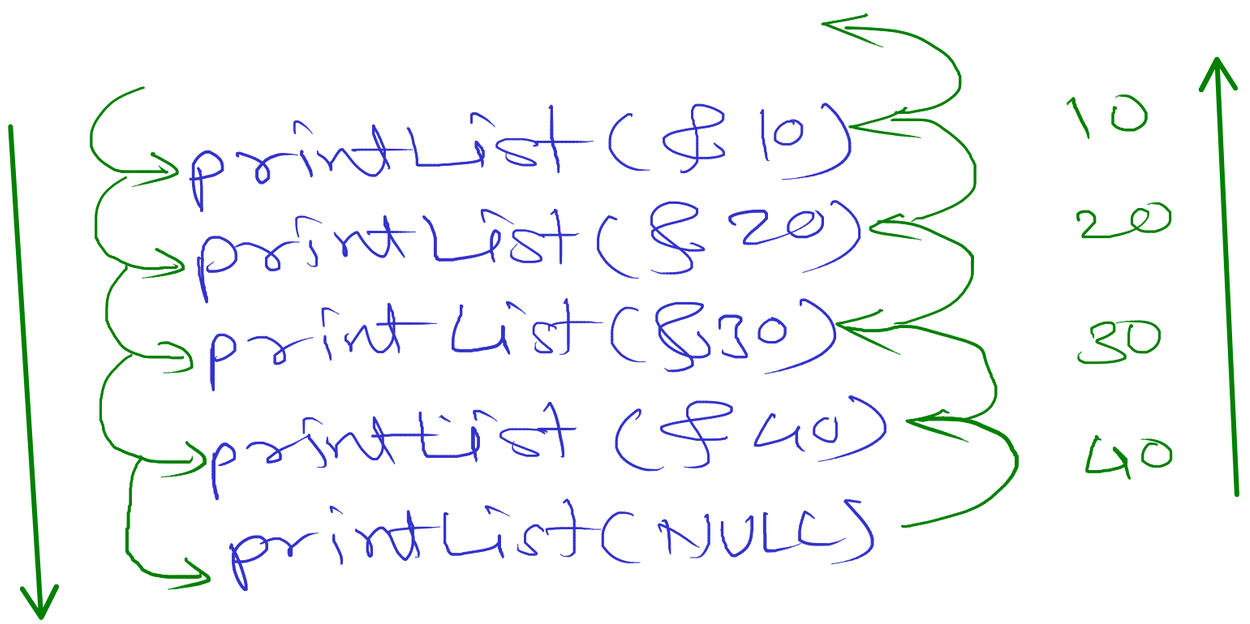
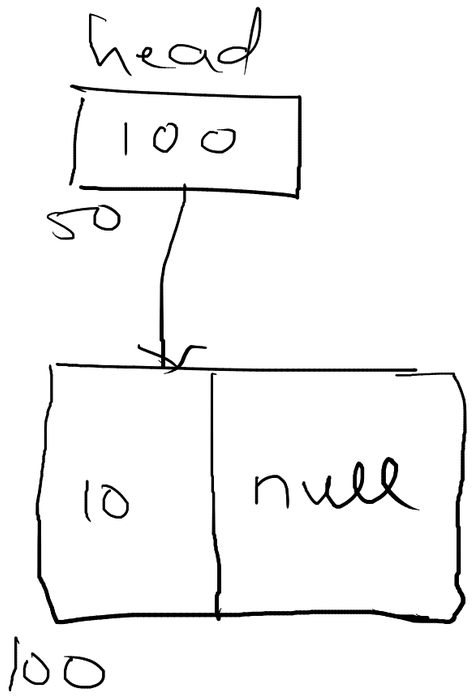
Print Singly Linked in Reverse order

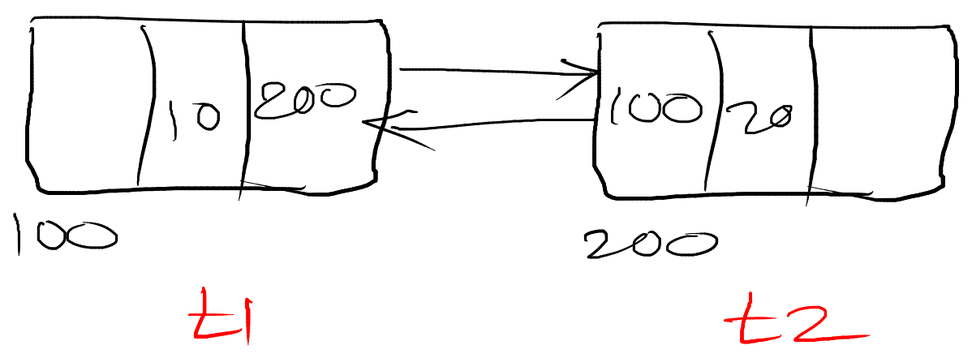



```
void printList(node_t *trav){
    if(trav == NULL)
        return;
    printList(trav->next);
    printf(trav->data);
}
```



Reverse Doubly List





t1 -> next = t2
t2 -> prev = t1
} before

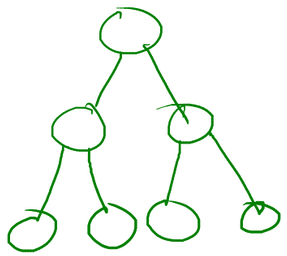
t2 -> next = t1
t1 -> prev = t2
} after

Max no. of nodes $(n) = 2^h - 1$

h=1

h=2

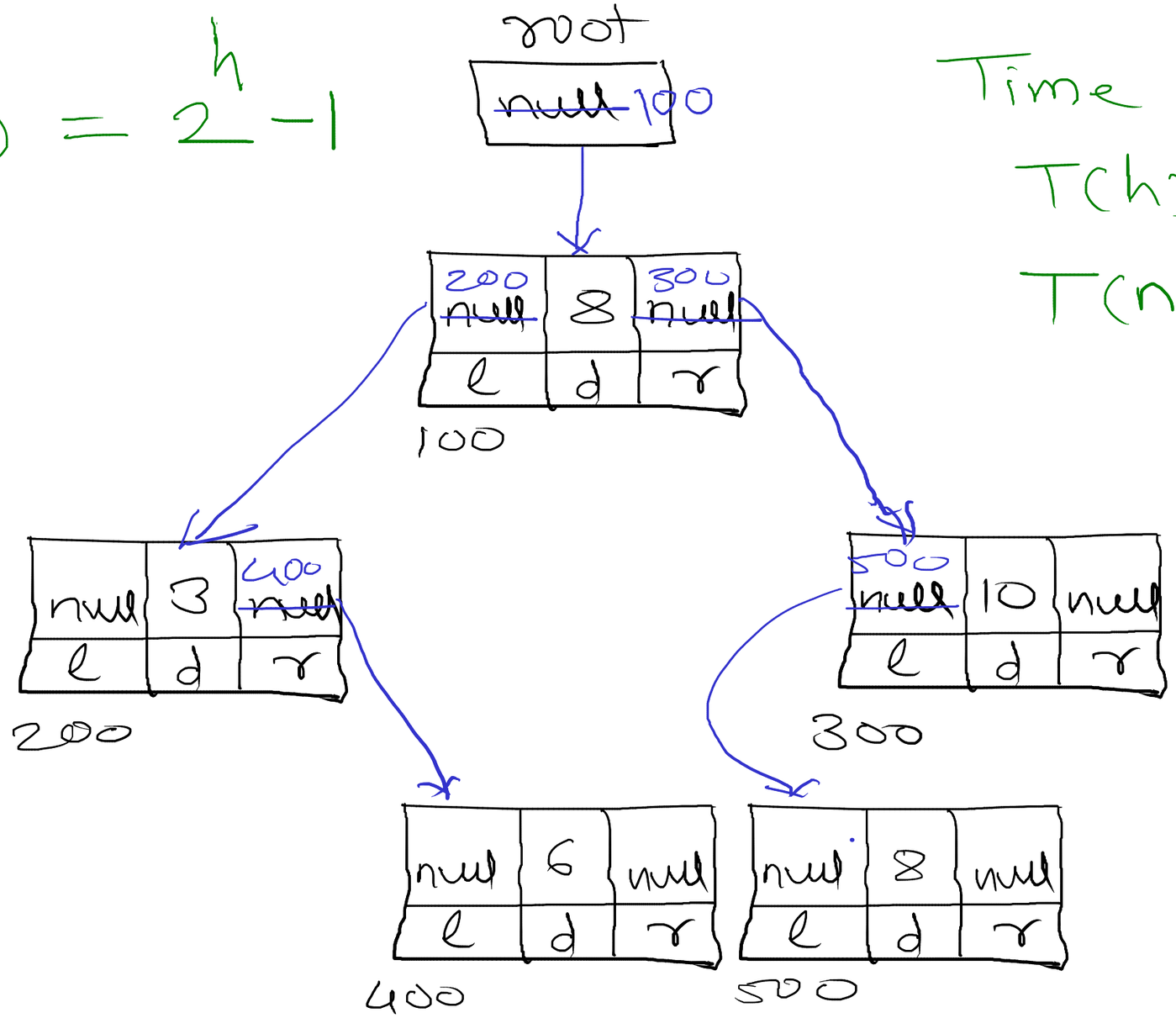
h=3



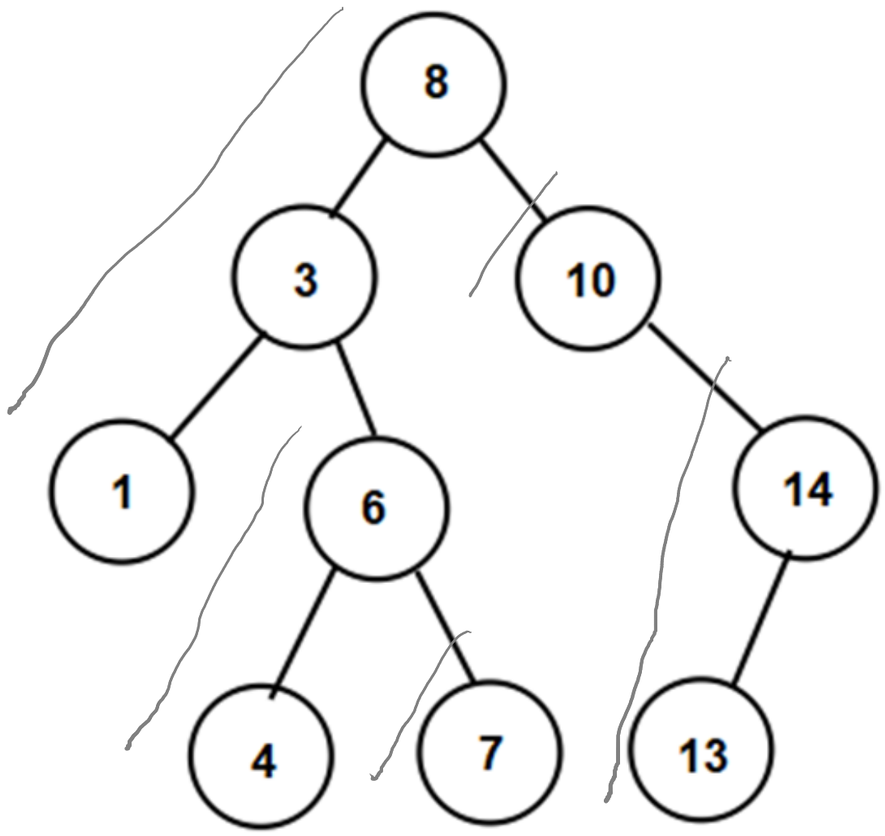
Time Complexity

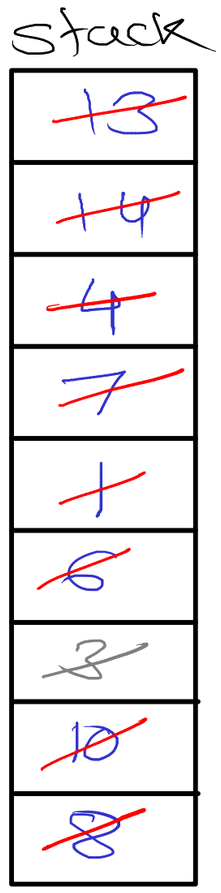
$T(h) = O(h)$

$T(n) = O(\log n)$



BST - DFS (Depth First Traversal)



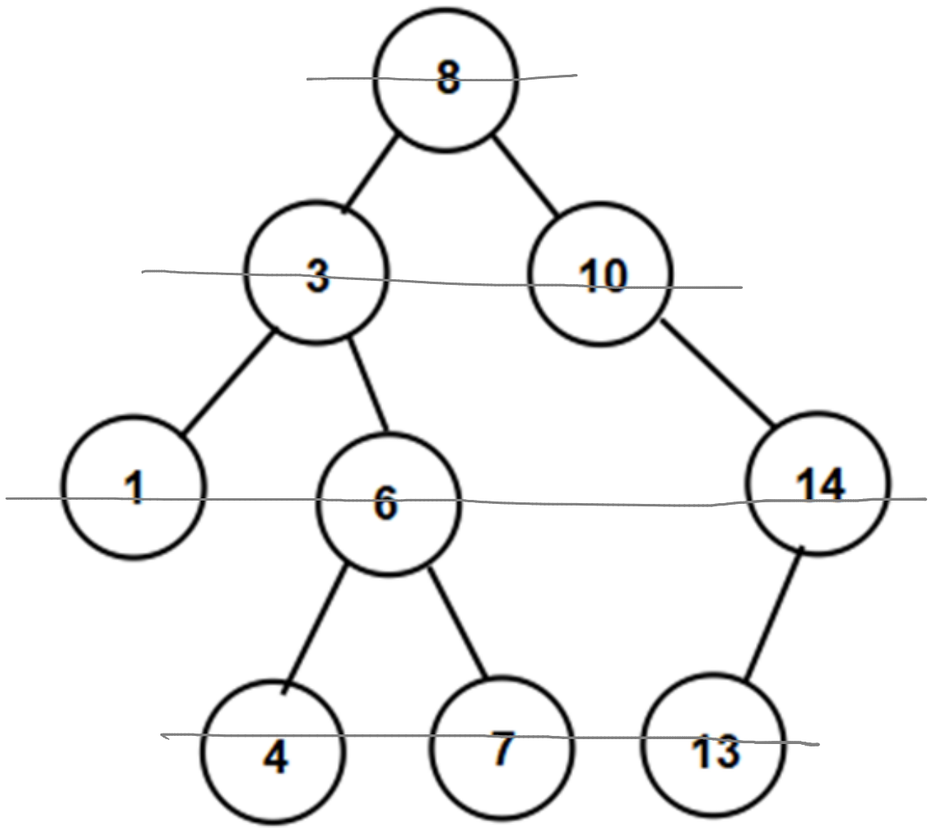


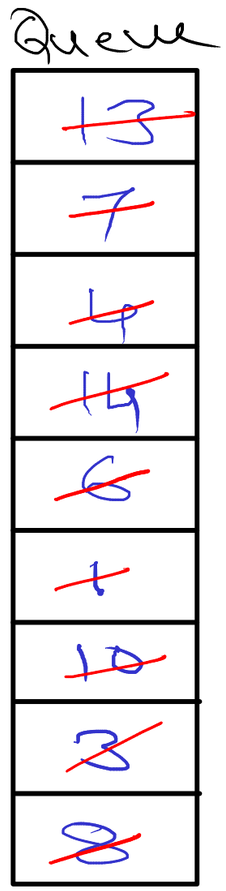
//1. push root on stack
//2. pop one node from stack
//3. visit(print) node
//4. if right exist, push it on stack
//5. if left exist, push it on stack
//6. while stack is not empty
//repeat ste 2 to 5

DFS Traversed:

8, 3, 1, 6, 4, 7, 10, 14, 13

BST - BFS (Bredth First Search)



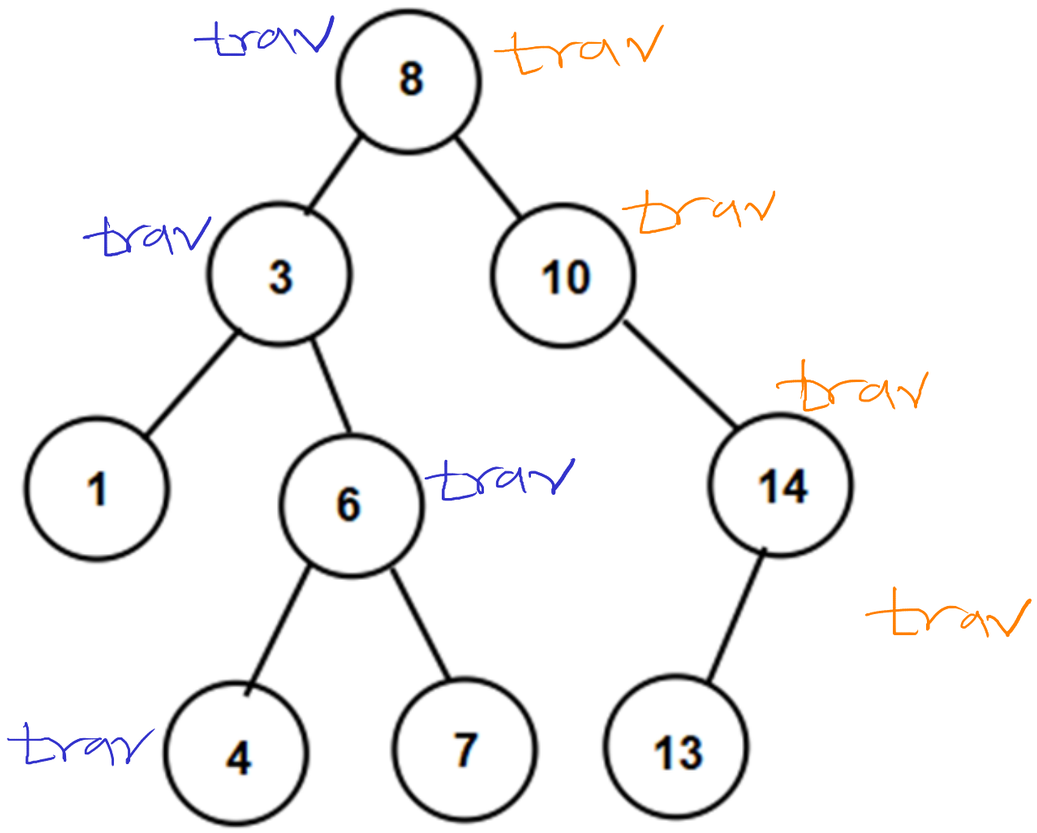


//1. push root on queue
//2. pop one node from queue
//3. visit(print) node
//4. if left exist, push it on queue
//5. if right exist, push it on queue
//6. while queue is not empty
//repeat ste 2 to 5

BFS Traversed:

8, 3, 10, 1, 6, 14, 4, 7, 13

BST - Binary Search



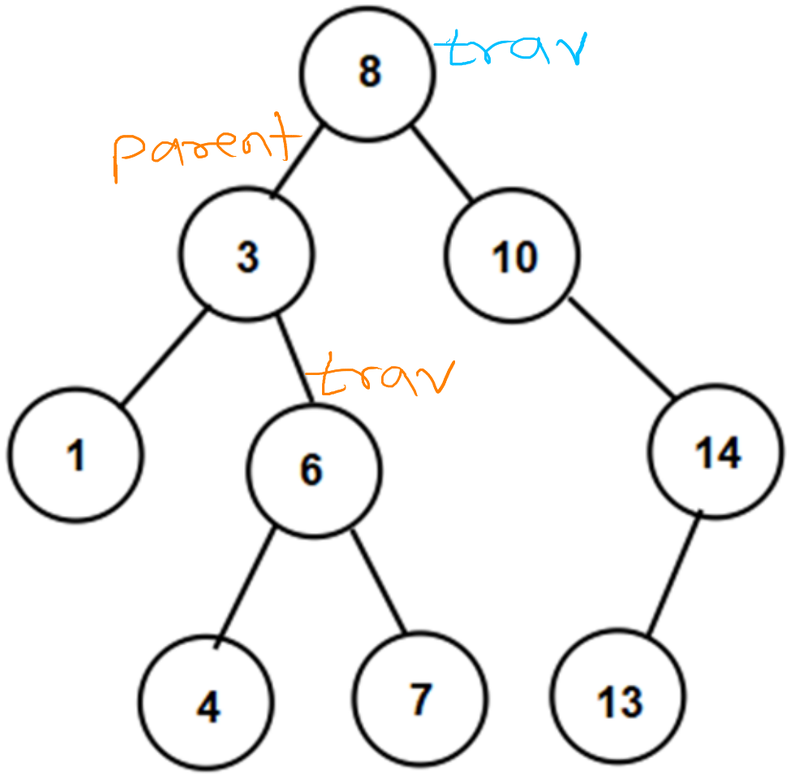
//1. start from root
//2. if key is equal to current data
//return current node
//3. if key is less than current data
// search key into left of current node
//4. if key is greater than current data
// search key into right of current node
//5. repeat step 2 to 4 till leaf nodes

key = 4 → key is found
key = 15 → key is not found

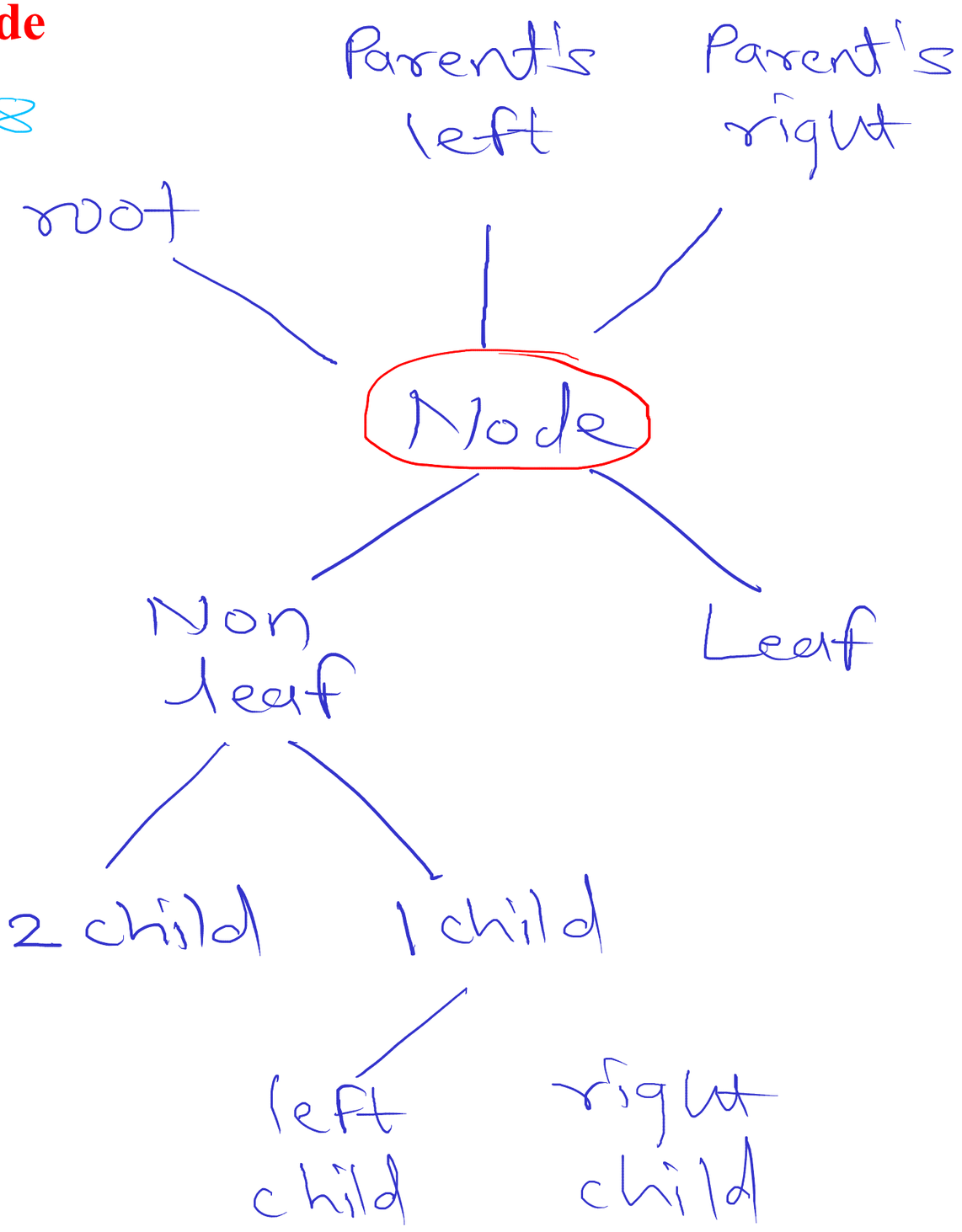
BST - Delete Node

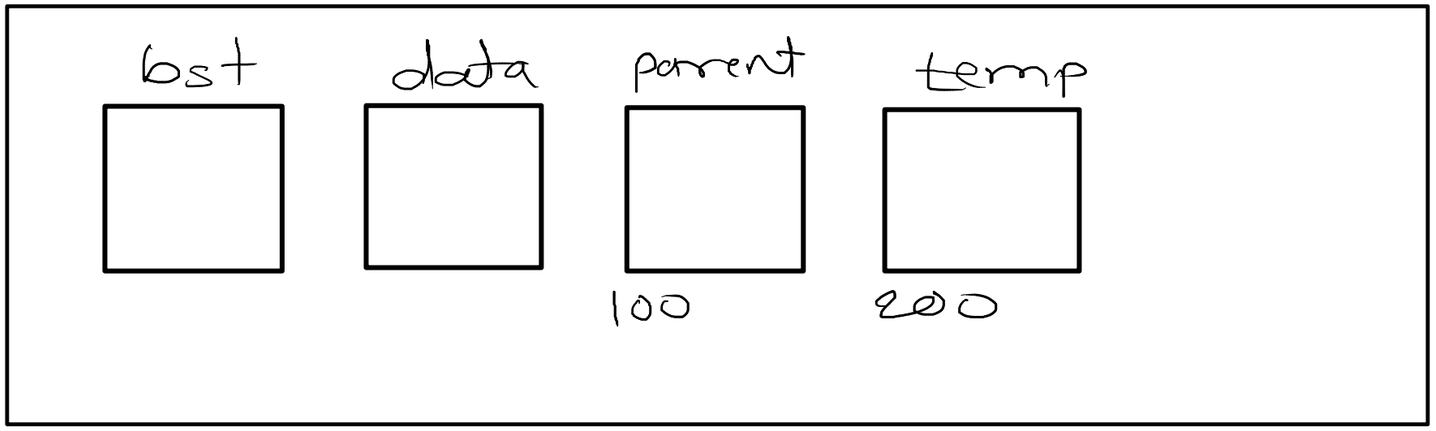
parent=NULL

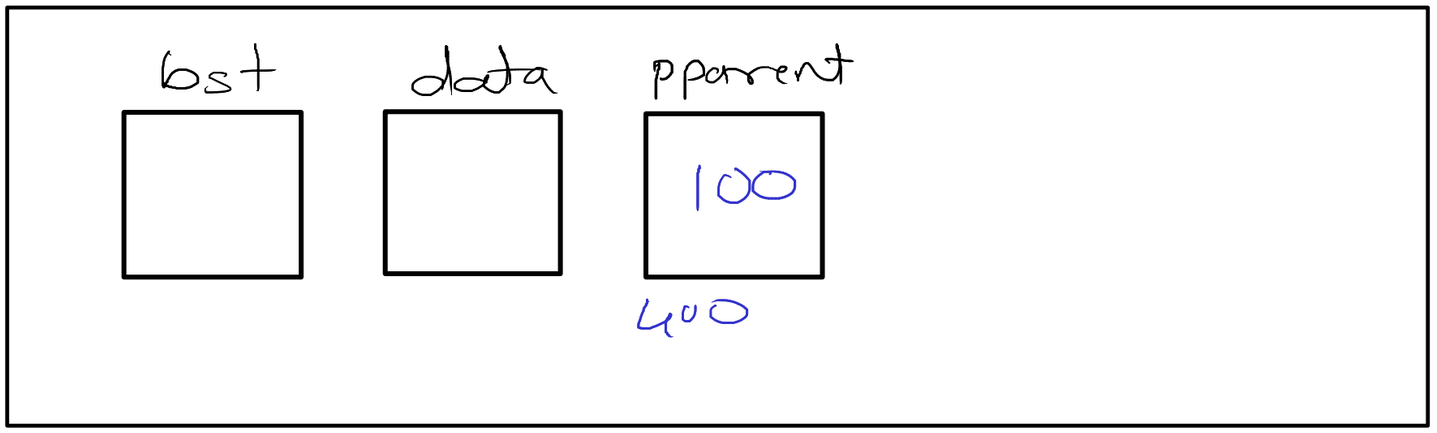
key=8



key=6

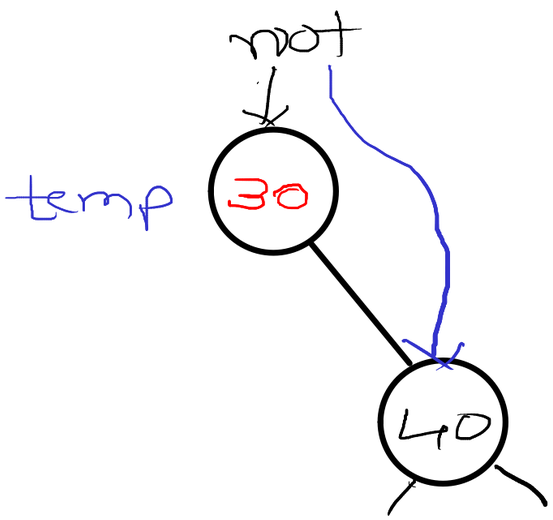



bst
data
parent
temp
100
200

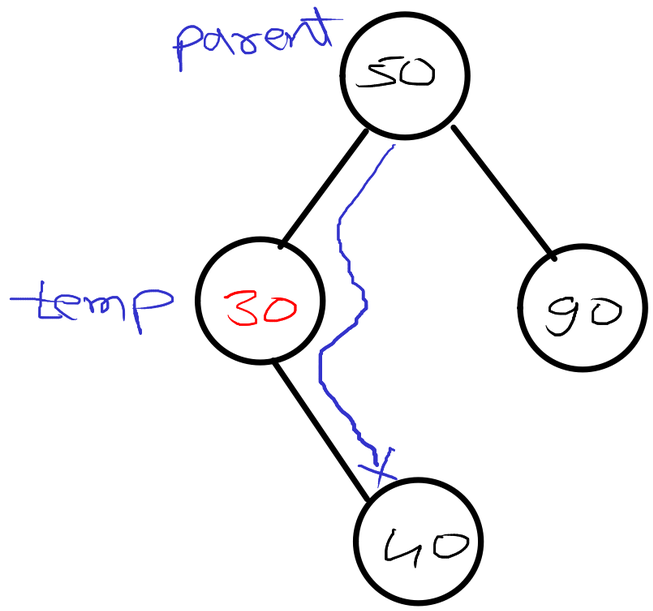

bst
data
pparent
100
400

BST - Delete node which has single child (right child)

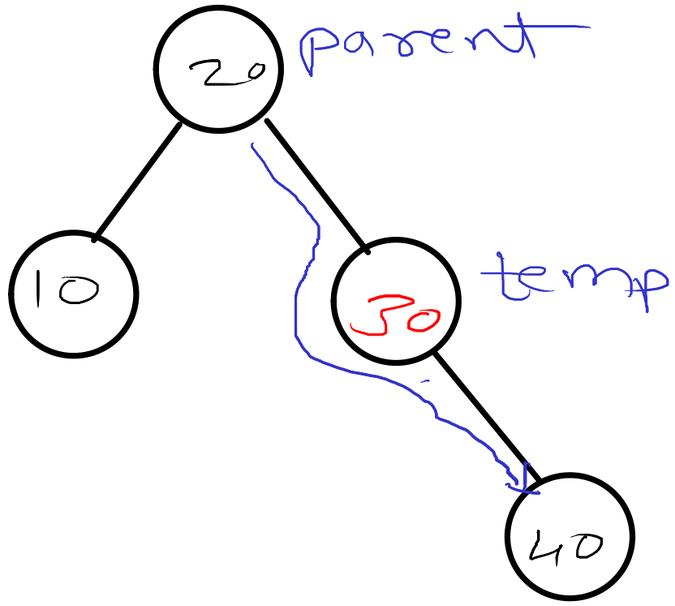
Root



Parent's left



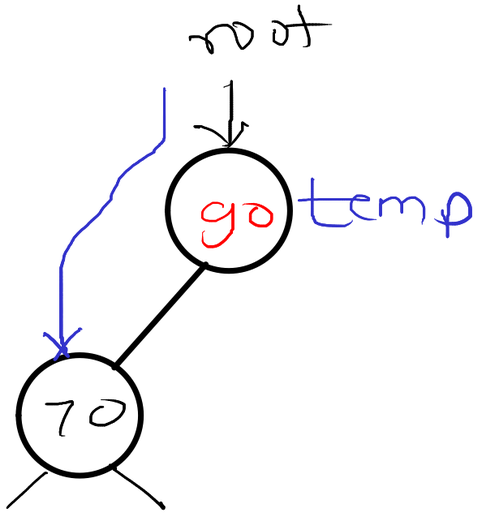
Parent's right



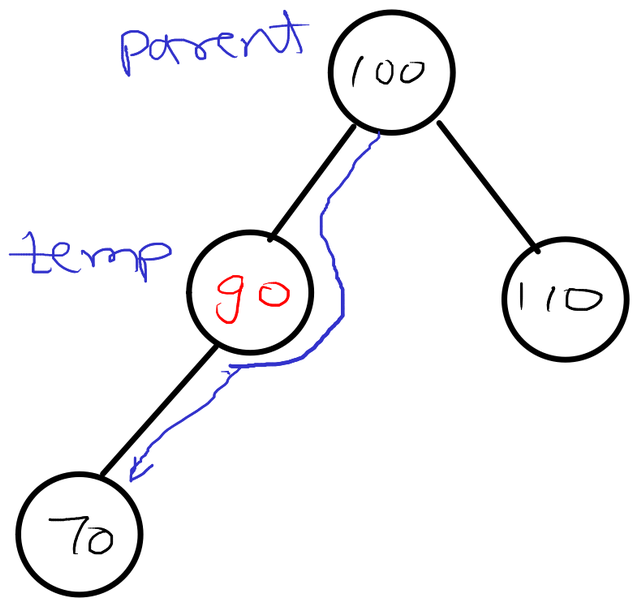

```
if(temp->left == NULL)
{
        if(temp == root)                        // root node
                root = temp->right;
        else if(temp == parent->left)                   // parent's left
                parent->left = temp->right;
        else                                            // parent;s right
                parent->right = temp->right;
}
```

BST - Delete node which has single child (left child)

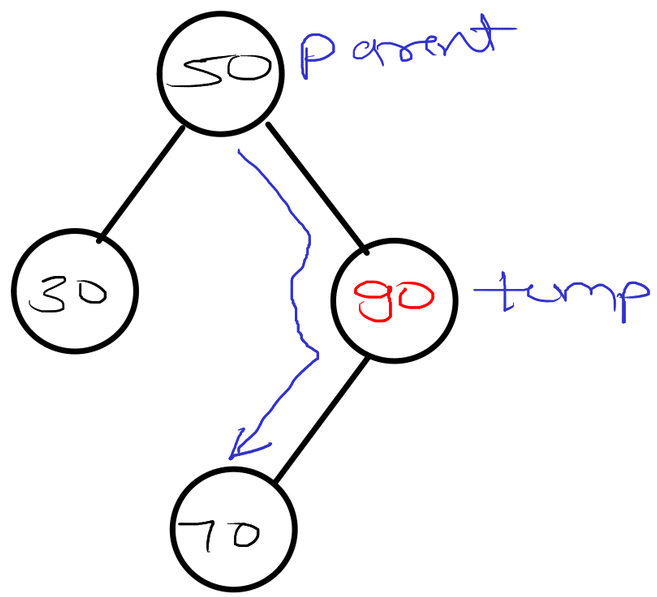
Root



Parent's left

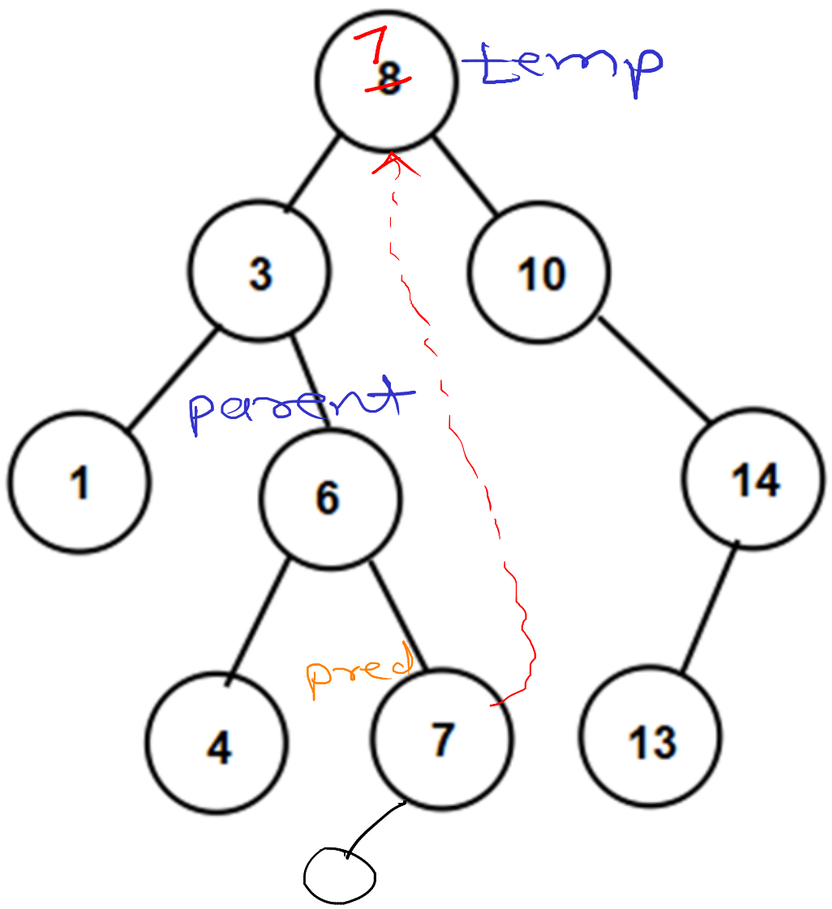


Parent's right




```
if(temp->right == NULL)
{
        if(temp == root)                        // root
                root = temp->left;
        else if(temp == parent->left)           // parent's left
                parent->left = temp->left;
        else                                    // parent;s right
                parent->right = temp->left;
}
```

BST - Delete node which has two childs




```
if(temp->left != NULL && temp->right != NULL)
{
        //a. find predecessor of node
        node_t *pred = temp->left;
        parent = temp;
        while(pred->right != NULL)
        {
                parent = pred;
                pred = pred->right;
        }
        //b. replace data by predecessor
        temp->data = pred->data;
        //c. move temp on predecessor
        temp = pred;
}
```

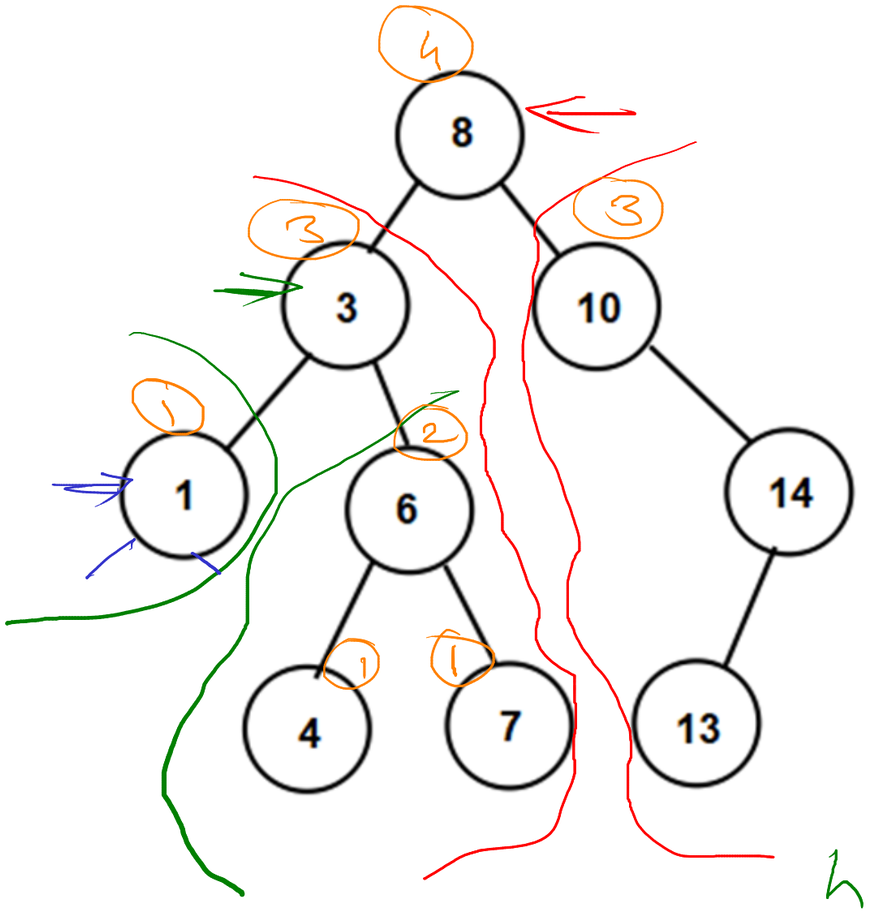
Inorder Traversal : 1 3 4 6 7 8 10 13 14

inorder predecessor (left extreme right)

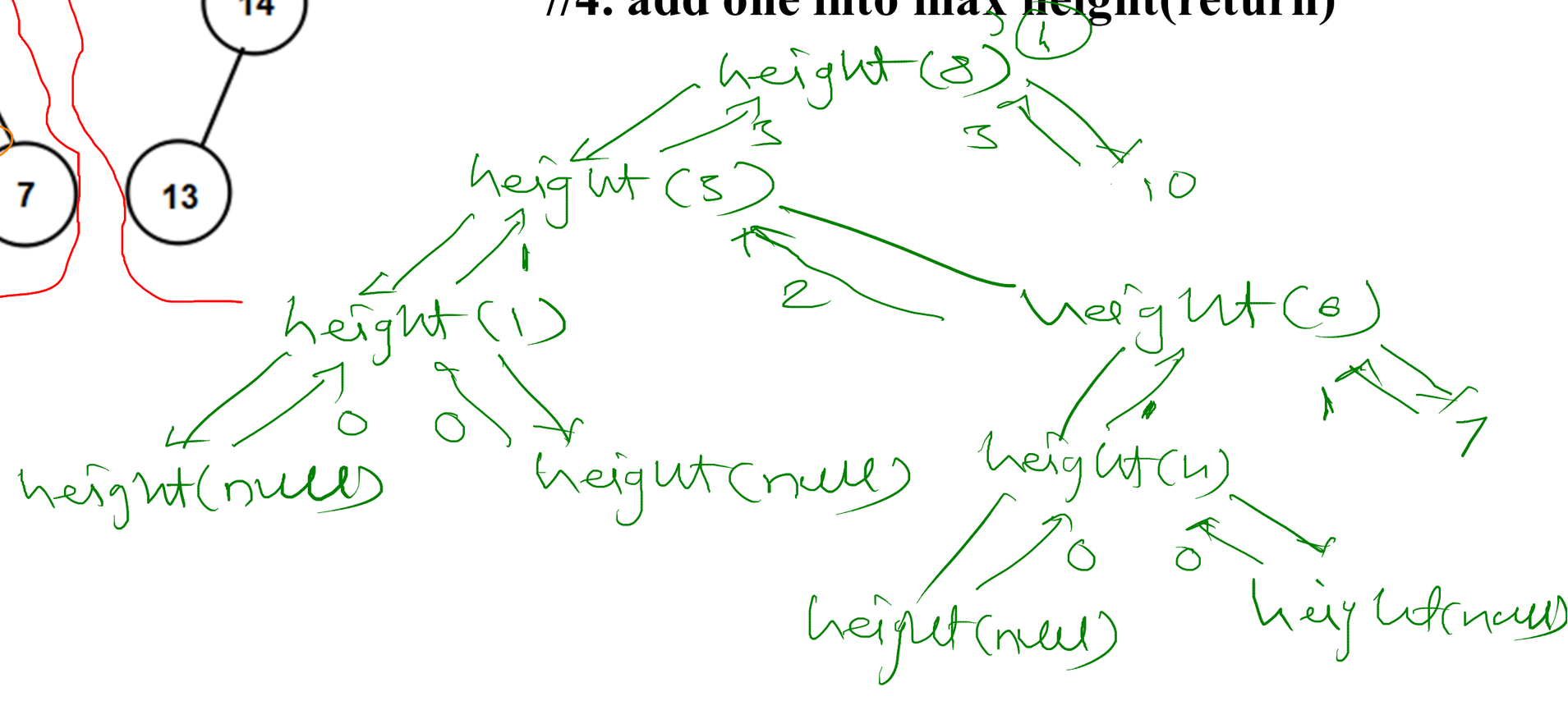
Inorder successor (right extreme left)

BST - Height

Height of tree = MAX(Height(left sub tree), Height(right sub tree)) + 1

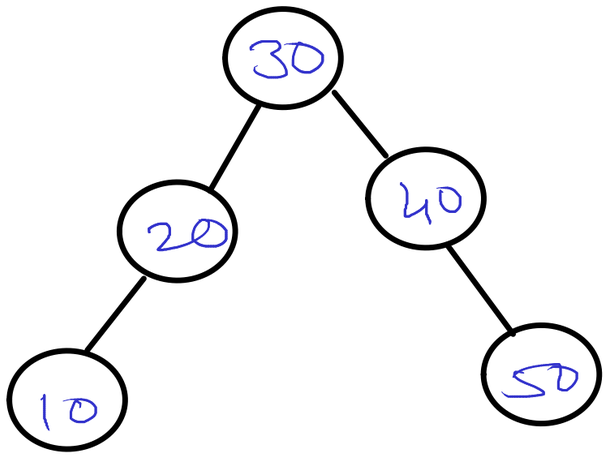



```
//0. if left or right sub tree is absent
                //then return 0
//1. find height of left subtree
//2. find height of right subtree
//3. find max height
//4. add one into max height(return)
```



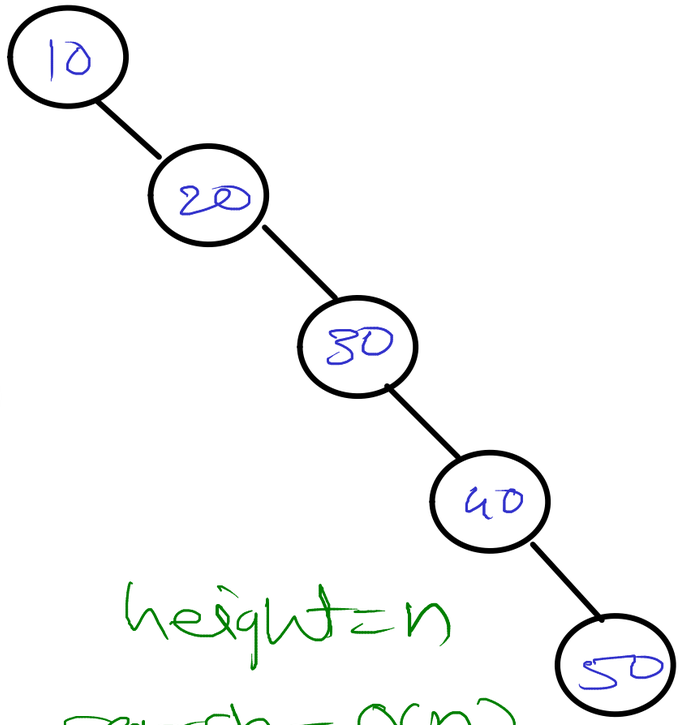
Skewed BST

Keys : 30, 40, 20, 50, 10



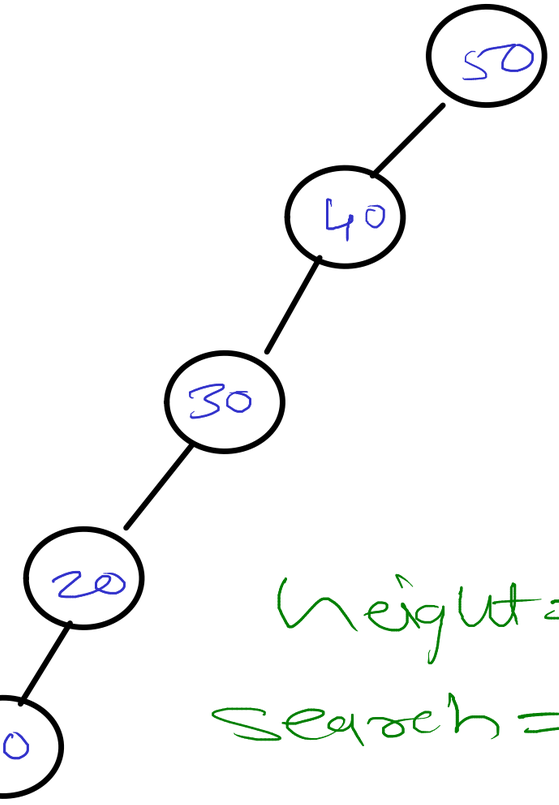
height = log n
search = O(log n)

Keys : 10, 20, 30, 40, 50



height = n
search = O(n)

Key : 50, 40, 30, 20, 10



height = n
search = O(n)

- **if tree is growing in only one direction , it is known as skewed BST**
- **if tree is growing in only right direction, it is known as right skewed BST**
- **if tree is growing in only left direction, it is known as left skewed BST**