Binary Search Tree

Each node has → key, left*, right*, p* ← point to parent

* Each node follows property left.key ≤ p.key
right.key ≥ p.key

left-most-node is minimum, right most node is maximum.

Inorder Tree walk

Traverse all left then right subtrees

```
INORDER-WALK(x)
if (x not NULL):
    INORDER-WALK(x.left)
    print(x.key)
    INORDER-WALK(x.right)
fi
```

Complexity: $\Theta(n)$

(Recursive)

$\Theta(h)$ ← height of tree

```
TREE-SEARCH(x, k)
if (x is NULL OR k = x.key):
        return x
if (k < x.key)
        return TREE-SEARCH(x.left, k)
else
        return TREE-SERCH(x.right, k)
```

```
TREE-SEARCH-ITERATIVE(x, key)

while (x not NULL & k not x.key)
    if k < x.key
        x = x.left
    else
        x = x.right
return x
```

in right subtree find min

Successor of ~~the~~ a node x ~~the~~ is the node y such that y.key is the smallest key > x.key

in left subtree find max

predecessor of a node x is the node y such that y.key is the largest key < x.key

MINIMUM(x)
while x.left not null
{do x = x.left }

Maximum(x)
while x.right not null {
do x = x.right

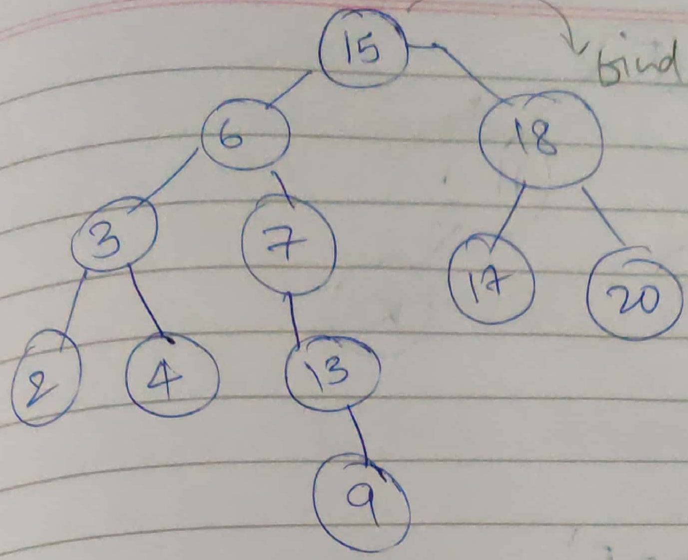
SUCCESSOR(x)

```
if x.right not NULL
        return TREE-MINIMUM(x.right)
y = x.p    // get parent of x
while y not NULL and x = y.right
{do   x = y
      y = x.p
return y
```

PREDECESSOR(x)

```
if x.left not NULL
        return TREEMAX(x.left)
y = x.p
while y not NULL & x = y.left
{do   x = y
      y = x.p
return y
```

successor(15)



find min(in 15.right)
= 17

successor(15) = 17
successor(6) = 7
successor(4) = 6
predecessor(6) = 4.

① parent(4) != NULL & x = y.right
y = parent of 4 = 3
x = 3; y = 6
② ~~y = parent of 3 = 6~~
② parent(3) != NULL & 3 = 6.right?
X

hence return y = 6

```
TREE-INSERT(T, x, z)
y = NULL      x = T.root
while (x not NULL) do
    y = x
    if (z.key < x.key) then
        x = x.left
    else
        x = x.right
z.p = y.
if (y == NULL)
    T.root = z
else if (z.key < y.key) then
    y.left = z
else
    y.right = z
```

z = node(value = v, l = Null, R = Null)

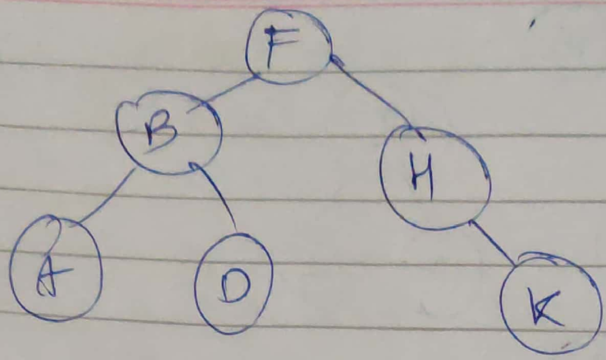
v is the value to be inserted

find position of z

follow BST property

place z

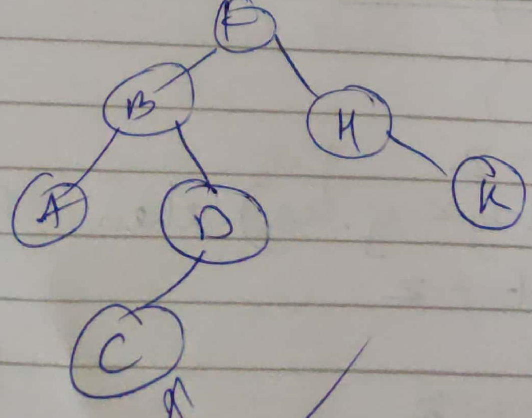
place acc to BST Rules



insert("C")

① y = NULL   x = "F"
while →

① x = F   y = F
C < F ⇒ x = B (left)
② x = B   y = B
C > B ⇒ x = D (Right)
③ x = D   y = D
C < D ⇒ x = NULL (left of D)
∵ x NULL
exit
now y = D
∴ z.p = D
Now   C < D ⇒ z.key < y.key
⇒ y.left = "C"
"D".left = "C"



BST-SORT(A)
let T be an empty BST
for i : 1 to n
{ do TREE-INSERT(T, A[i])
INORDER-TREEWALK(root(T))

Worst case $O(n^2)$ ~~Best O(~~ O(n·depth) = ~~$O(n\log n)$~~

Best = $O(n \log n)$

Deletion Operation ( )
replace ~~sub~~ subtree as the child of its parent by another subtree
~~TRANSPLANT(T, U, V)~~

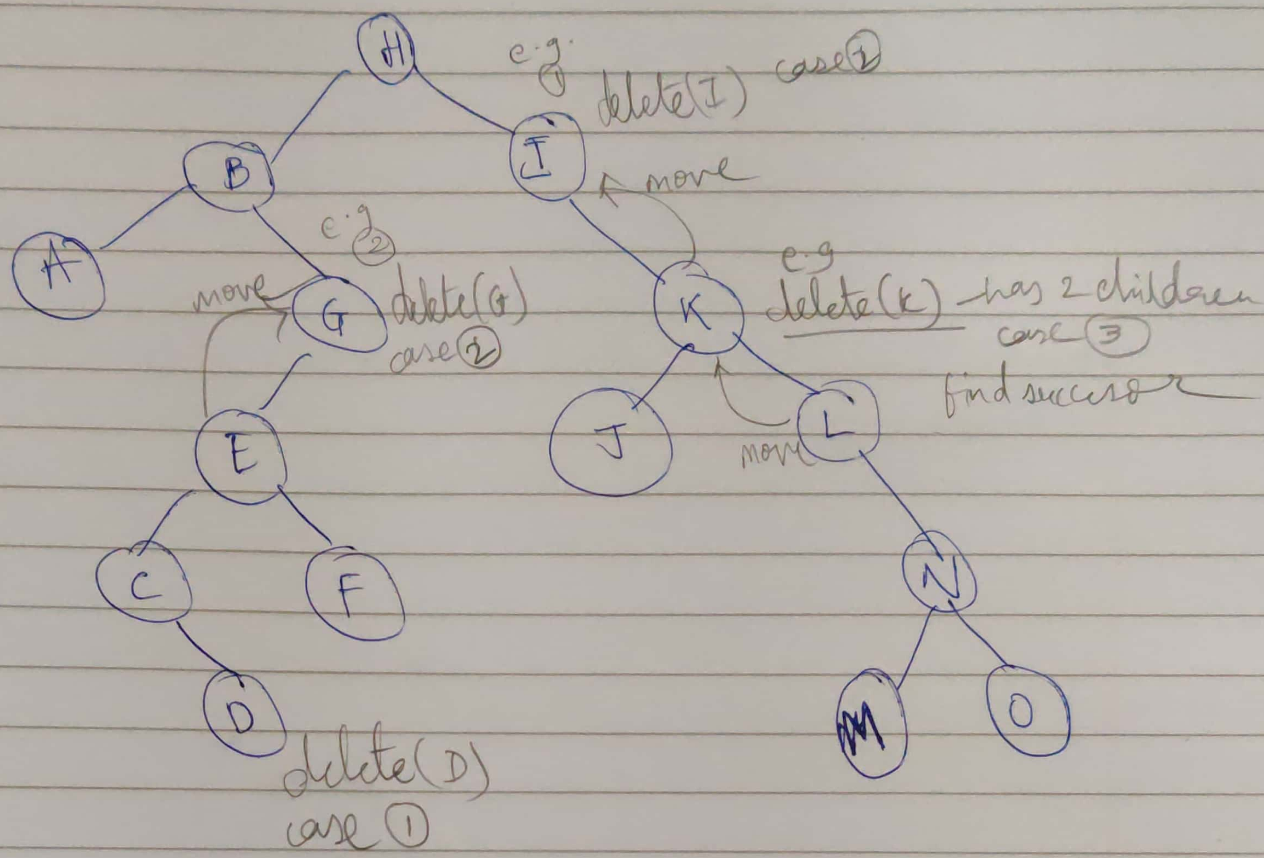
TRANSPLANT (T, u, v)
if (u.p = NULL)
  T.root = v
else
  if (u = u.p.left) then
    u.p.left = v
  else u.p.right = v
  if (v not NULL)
    v.p = u.p

3 cases of deletion
① if node is leaf node, delete easily

② node has one child so delete and replace with the only child

③ if node has 2 ~~subtree~~ children replace with successor of deleted node such that chosen successor has no left subtree

TREE-DELETE(T, z)
if (z.left = NULL)
    TRANSPLANT(T, z, z.right) — case ① & 2
else
    if (z.right = NULL) then
        TRANSPLANT(T, z, z.left) — case ① & ②
case 3
2 children
    else // look for successor:
        y = TREE-MINIMUM(z.right)
        if (y.p ≠ z) // y is not z.right
            TRANSPLANT(T, y, y.right)
            y.right = z.right
            y.right.p = y
        TRANSPLANT(T, z, y) // y is z.right
        y.left = z.left
        y.left.p = y



delete(B)
① find successor(B) = C

⑤ change B with C
D now become left child of E
satisfies BST
G becomes right child of C at B's place