Agenda

1. K^{th} Smallest Element
2. Morris Inorder Traversal
3. LCA in Binary Tree
4. LCA in BST
5. In time and out time

Binary Tree

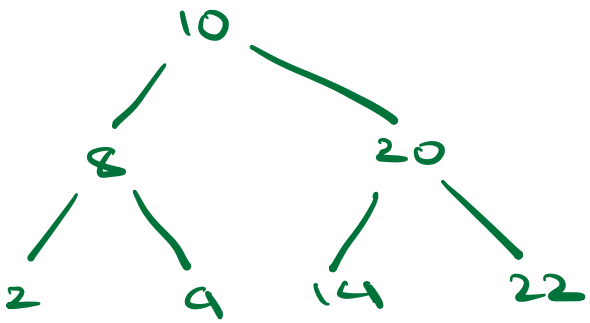
- Hierarchical data structure, composed of tree nodes
- Can have atmax 2 children: left and right

Binary Search Tree (BST)

O(H)

- For every node x
  - All nodes in LST $\leq x$
  - All nodes in RST $> x$

1. Given a BST and a positive integer k, find k^{th} smallest element in BST.



k	ans
3	9
5	14

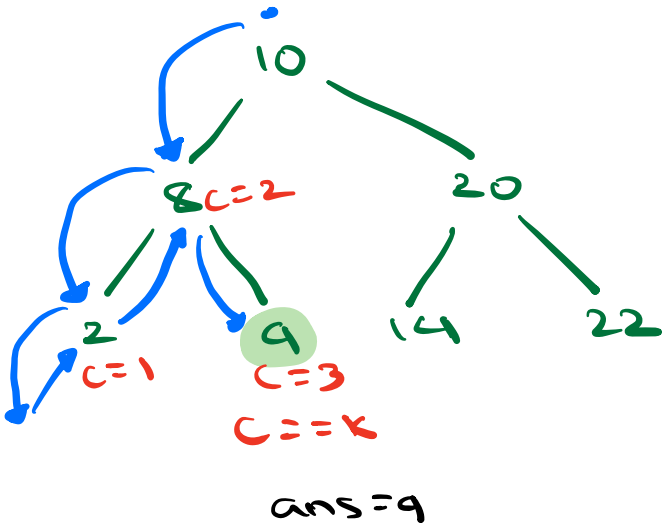
0 1 2 3 4 5 6
2 8 9 10 14 20 22

Approach 1:

Do inorder traversal of BST
↓
store in arr[ ]
↓
return arr[k-1]

TC: O(N) SC: O(H + N)
↓
Recursive stack

Approach 2 : keep a global count variable while doing inorder traversal



K=3

cnt = 0 1 2 3

ans=9

ans

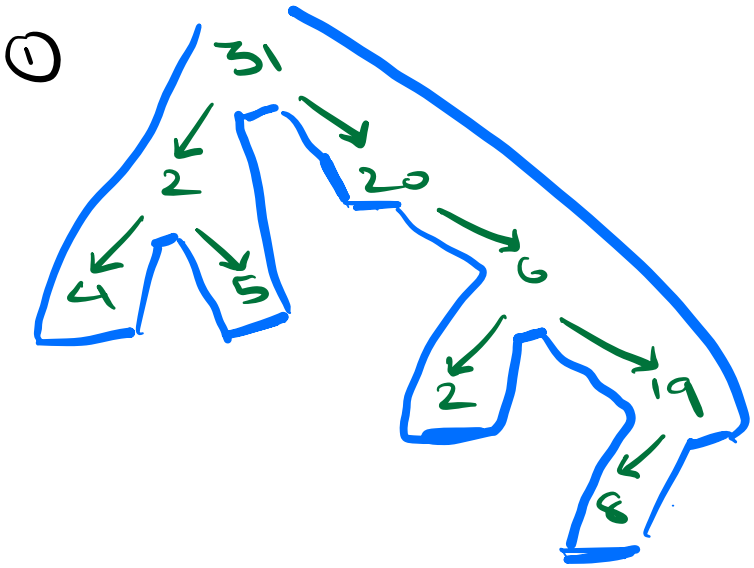
cnt = 0

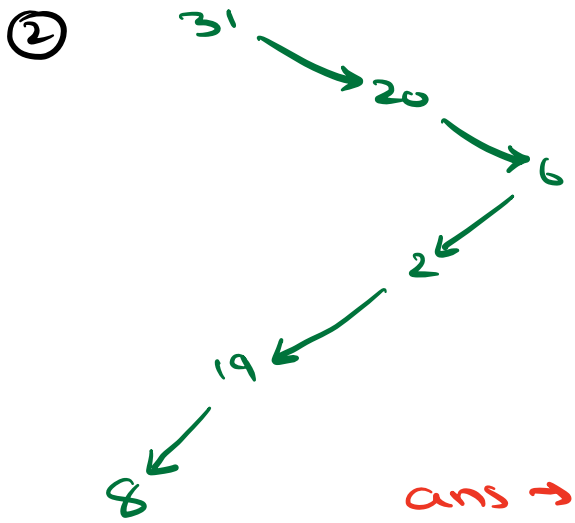
```
void inorder (Node root) {
    if (root == NULL)
        return
    inorder (root.left)
    cnt++
    if (cnt == K) {
        ans = root.data
        return
    }
    if (cnt < K) {
        inorder (root.right)
    }
}
```

TC : O(N)

SC : O(H)

Q. Which is the last node printed in inorder traversal ?

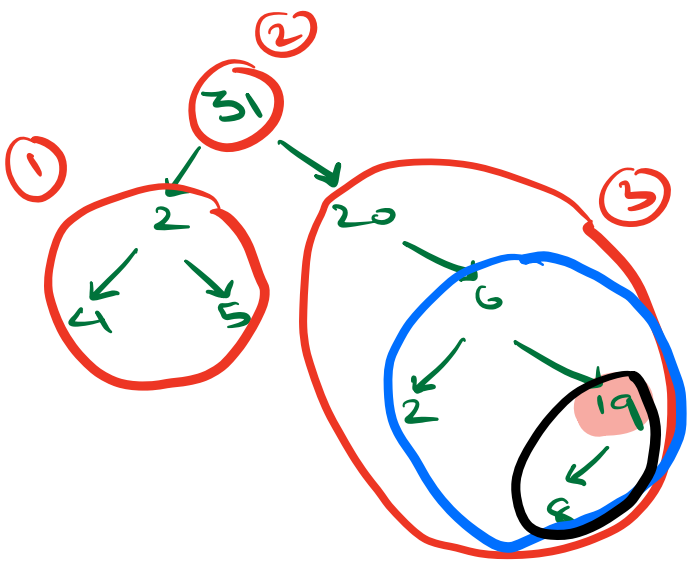




ans → 6

4 2 5 31 20 2 6 8 19

ans → 19

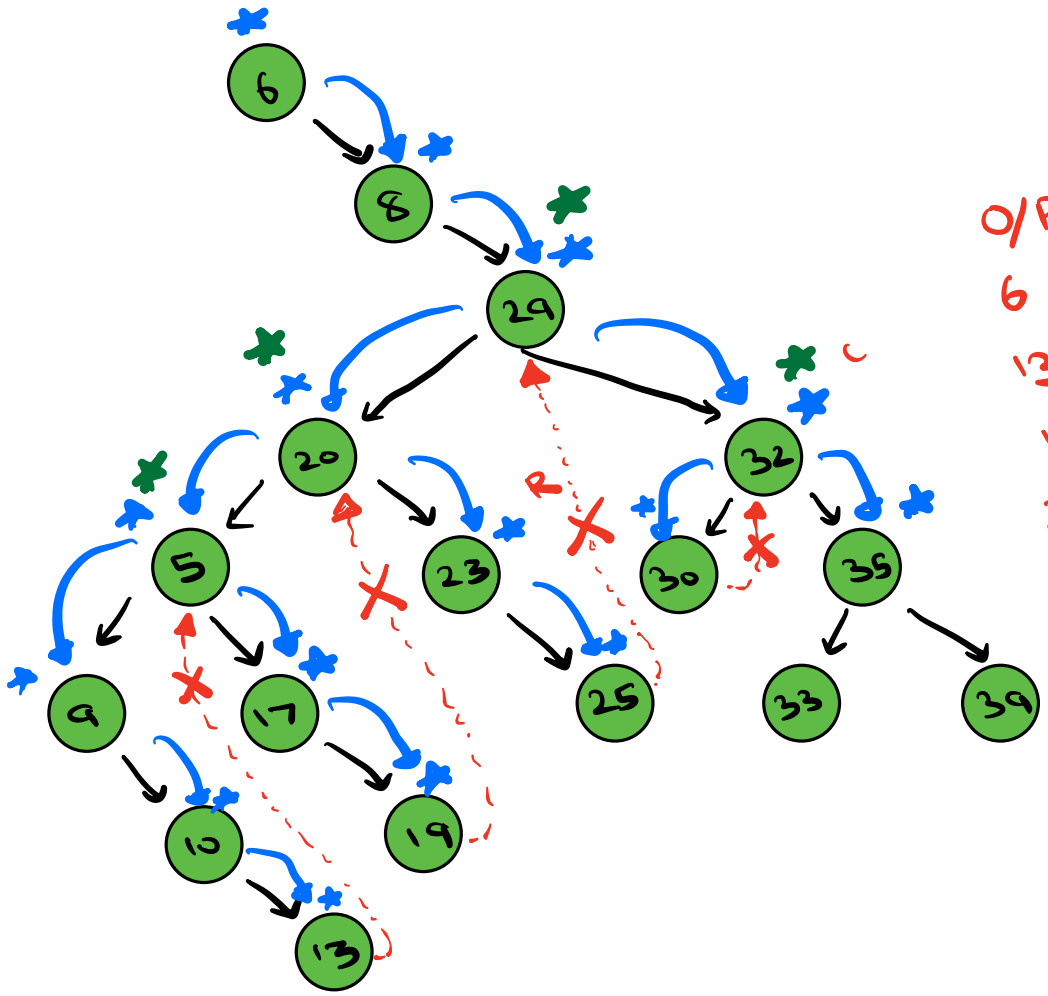


L N R

Rightmost node of tree → last node in inorder

2. Morris Inorder Traversal

SC → O(1)



O/P

6 8 9 10
13 5 17
19 20 23
25 29 30
32

25 . right = 29

Last node of LST of 29

↓

Rightmost node of LST

SC

Recursive → O(H)

Iterative → O(H)

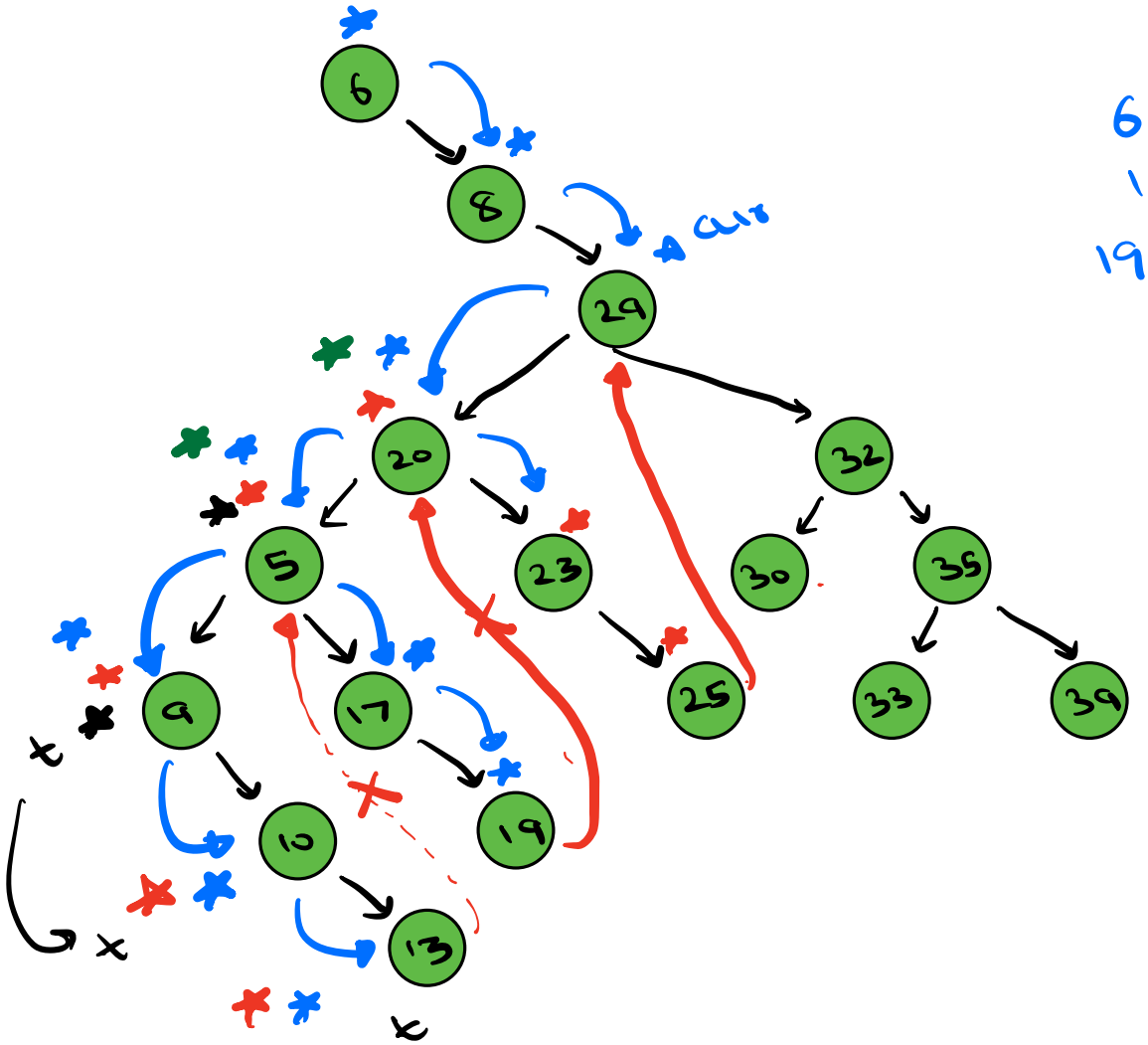
↓

O(1)

```
void inorder (Node root)
    Node cur = root
    while (cur != NULL) {
        if (cur.left == NULL) {
            print (cur.data)
            cur = cur.right
        }
        else {
            Node temp = cur.left
            while (temp.right != NULL && temp.right != cur) {
                temp = temp.right
            }
            if (temp.right == NULL) {
                temp.right = cur
                cur = cur.left
            }
            else if (temp.right == cur) {
                temp.right = NULL
                print (cur.data)
                cur = cur.right
            }
        }
    }
```

TC : O(N)

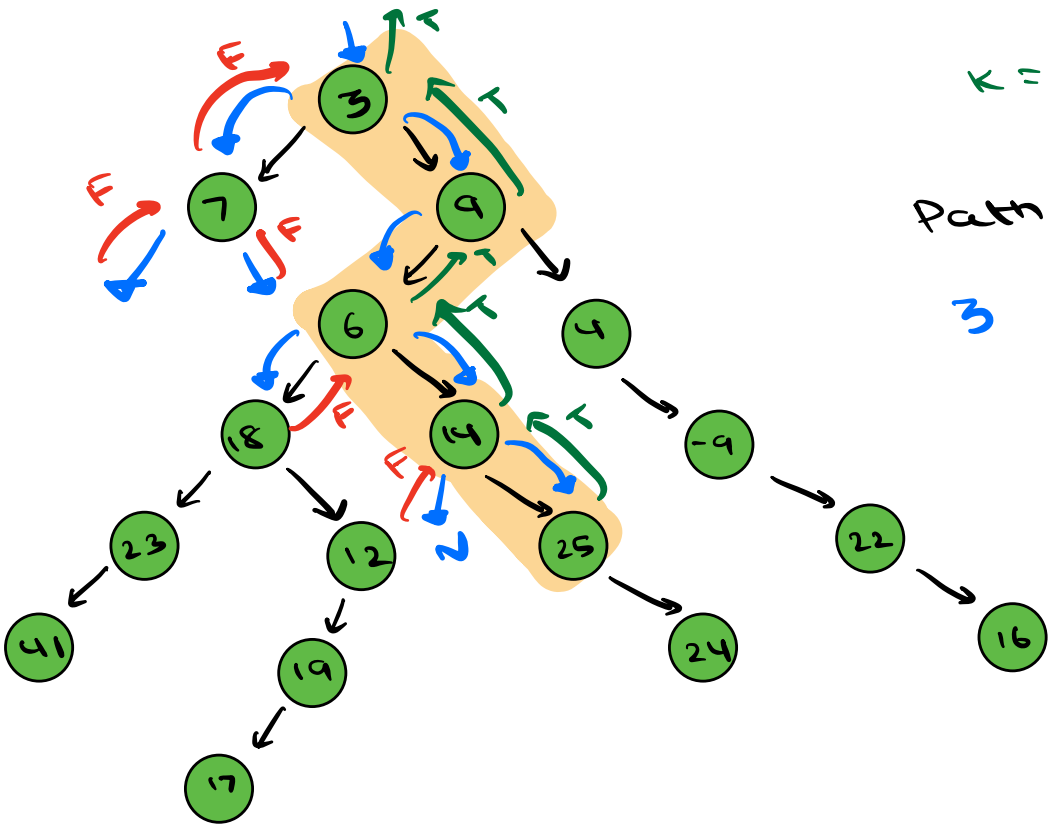
SC : O(1)



6 8 9 10
13 5 17
19 20

10:35

3. Search K in Binary Tree



K = 25

Path (3 → 25)

3 9 6 14 25

```
// Given root, find k in tree
bool search (Node root, int k) {
    if (root == NULL)
        return false
    if (root.data == k)
        return true
    return (search(root.left, k) ||
            search(root.right, k))
}
```

TC: O(N)

SC: O(H)

$\log_2 N \rightarrow N$

4. Path from root node to K

List <int> path

```
// Given root, find k in tree
bool search (Node root, int k) {
    if (root == NULL)
        return false
    if (root.data == k) {
        path.add (root.data)
        return true
    }
    if (search (root.left, k) == true ||
        search (root.right, k) == true) {
        path.add (root.data)
        return true
    }
    else
        return false
}
```

TC: O(N)

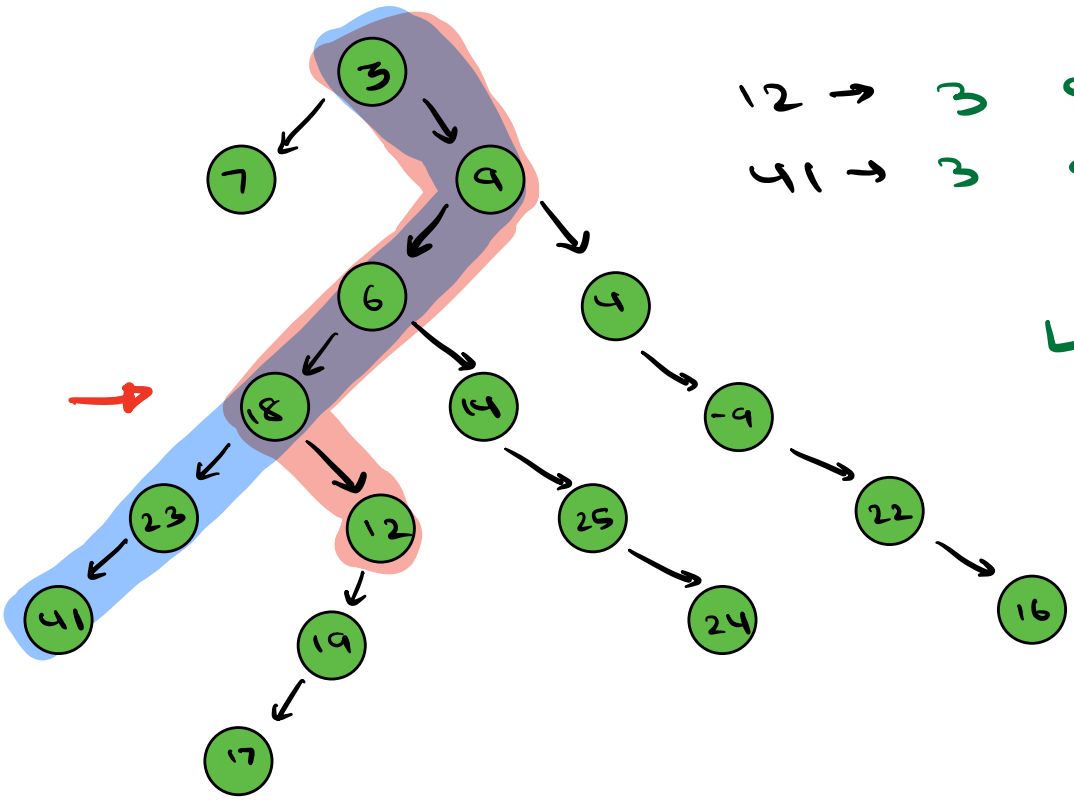
SC: O(H)

$\log_2 N \rightarrow N$

* Your path is in reverse order (k → root)

Reverse it

Lowest Common Ancestor in Binary Tree



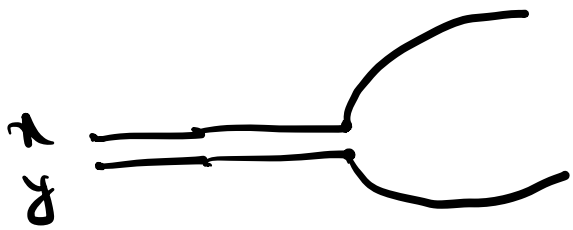
12 → 3 9 6 18 12

41 → 3 9 6 18 23 41

LCA (41, 12) = 18

ans = ~~3~~ ~~9~~ ~~6~~ 18

LCA (18, 41) = 18



18 : 3 9 6 18

41 : 3 9 6 18 23 41

Last node where path diverge → LCA

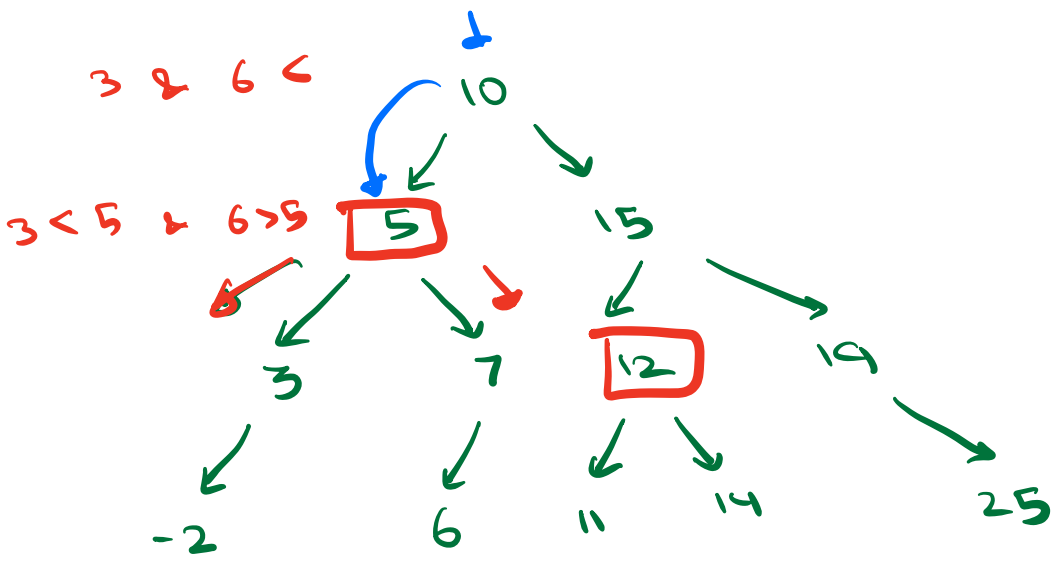
LCA (x, y)

1. Path (root → x)
2. Path (root → y)
3. Traversing both paths simultaneously until first non-common node → break

TC : O(N)

SC : O(H)

LCA of 2 nodes in BST



3 and 6

ans = 5

11 and 14

ans = 12

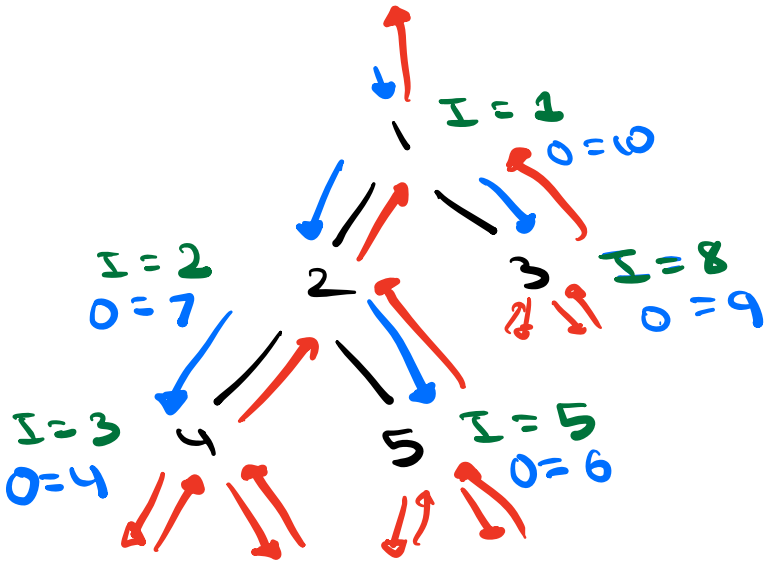
LCA(x,y)

```
Node cur = root
while (cur != NULL) {
    if (x < cur.data && y < cur.data)
        cur = cur.left
    else if (x > cur.data && y > cur.data)
        cur = cur.right
    else
        return cur
}
```

TC : O(H)

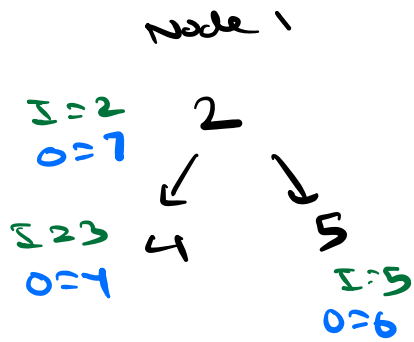
$\log_2 N \rightarrow N$

In time and out time



T = 1 2 3 4 5 6 7 8 9

10 11



Node 1		Node 2
In 1	<	In 2
Out 1	>	Out 2

Node 2 is in subtree of Node 1

Hashmap < Node, int > intime, outtime

T = 1

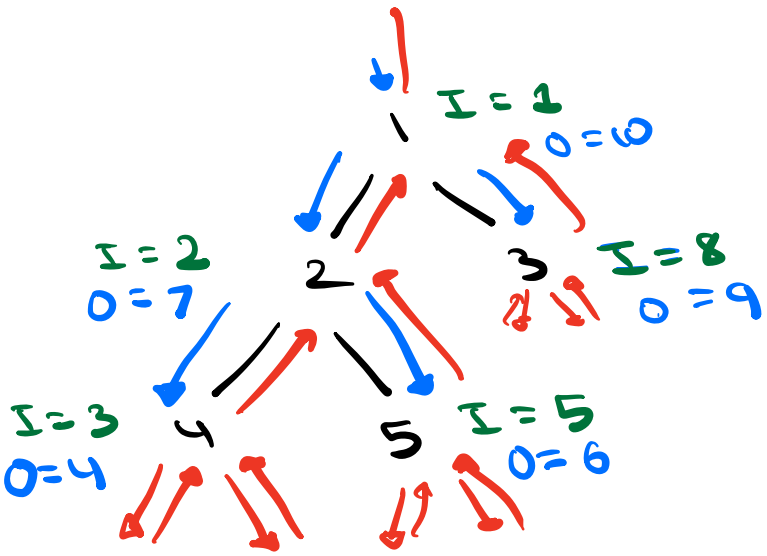
```
void dfs (Node root) {
    if (root == NULL)
        return

    intime [root] = T
    T++
    dfs (root.left)
    dfs (root.right)
    outtime [root] = T
    T++
}
```

Find LCA

→ Calculate in time, outtime

→ Store parent of each node

→ Store depth of each node



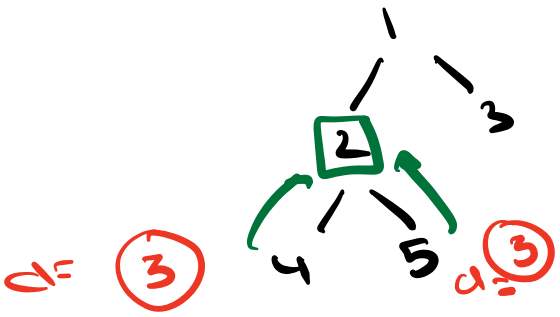
LCA(2,5) = 2

5 is in subtree of 2

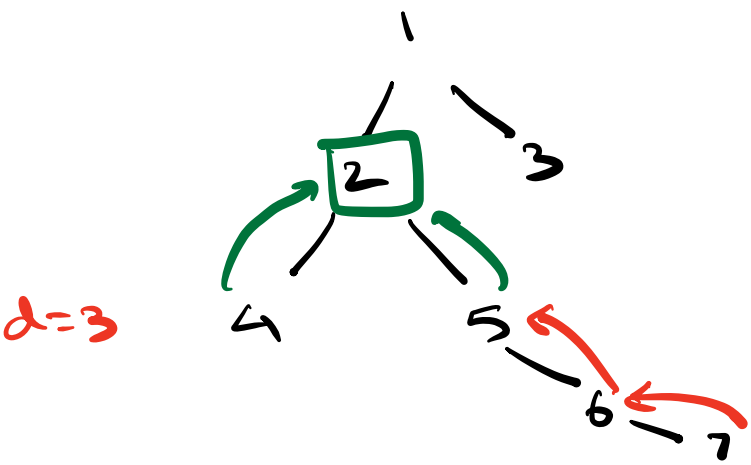
Node 2 ↓ Node 1

Out 2 < Out 1

In 2 > In 1



LCA(4,5) = 2



LCA(4,7) = 2

```
Node findLCA(x, y)

    if (inTime[x] <= inTime[y] &&
        outTime[x] >= outTime[y]) {
        return x
    }

    else if (inTime[x] >= inTime[y] &&
             outTime[x] <= outTime[y]) {
        return y
    }

    else {
        while (depth[x] > depth[y])
            x = parent[x]

        while (depth[x] < depth[y])
            y = parent[y]

        while (x != y) {
            x = parent[x]
            y = parent[y]
        }

        return x
    }
```

If y is subtree of x

Precomputation → TC : O(N) SC : O(N)

Query for LCA → TC : O(H) SC : O(1)