
O
G
D
O
O
O
O
D

Binary Trees   Binary Search Trees

Arrays   Linked List

Capture Order

Set

Binary Trees - capture hierarchy

Binary Tree is a set of elements with

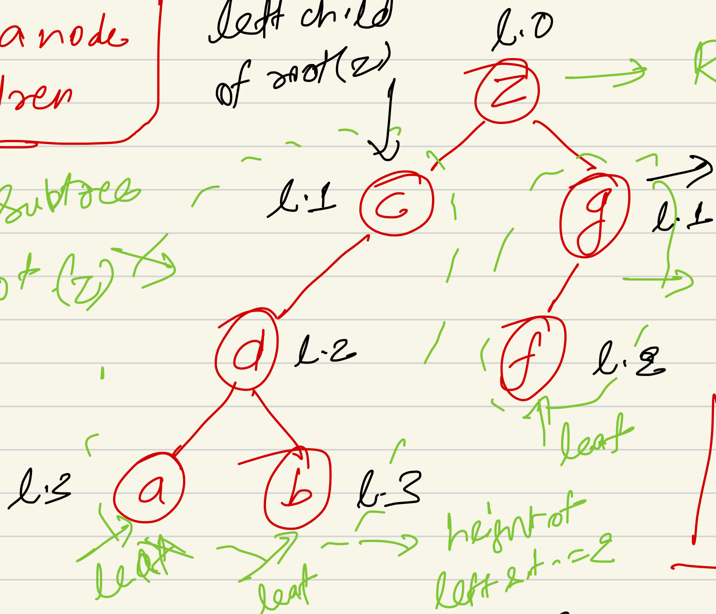
→ one special symbol called root

→ all other elements can be partitioned into two sets (possibly empty)

- left
- right

both these are binary trees themselves

leaf node ⇒ a node with 0 children



Height = 3 root(z)

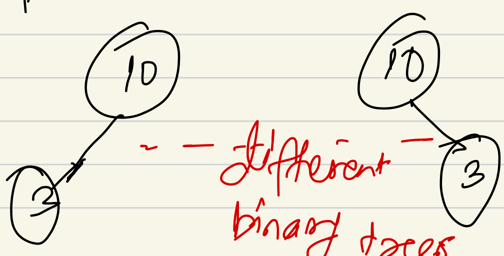
* There is no cycle ⇒ there is exactly one path from any node to any other node

left-right distinction is important

Parent of c? → z

left child of d? → a

left child of b? → NIL



{3, 10}

* level of root node = 0
* level of any node = distance of the node from root

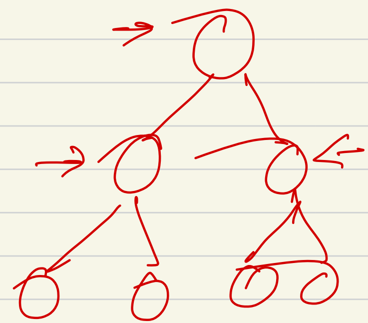
distance of a node from root

$\Rightarrow$ No. of edges in the path from root to the node

Height of a tree = maximum level of any node in the tree

* Max. no. of nodes at level h = 2^h

* Max. no. of nodes in a tree with height h



0		
1	2^1	
2	2^2	
⋮		
h	2^h	

$\Rightarrow$

$$2^0 + 2^1 + \dots 2^h$$

$$= 2^{h+1} - 1$$

* Min height of a tree with n nodes
h

h max. $2^{h+1}-1$ nodes

$$n \le 2^{h+1}-1$$

$$h \ge \log_2(n+1) - 1$$

$$= \Omega(\log n)$$

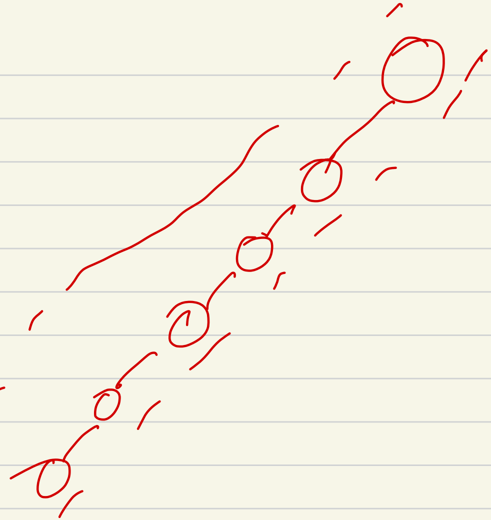
Can I say $h = O(\log n)$?

Max. possible height for a tree with n nodes

n nodes

$\Rightarrow$ height

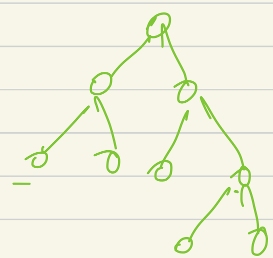
$= n - 1$

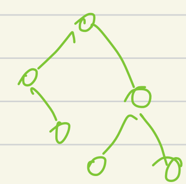


$h \Rightarrow O(n)$

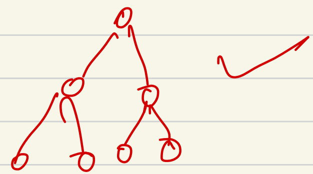
$\Omega(\log n)$

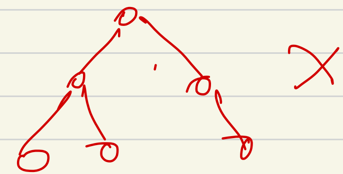
Full Binary Trees $\Rightarrow$ Binary Tree where all node have either 0 or 2 children





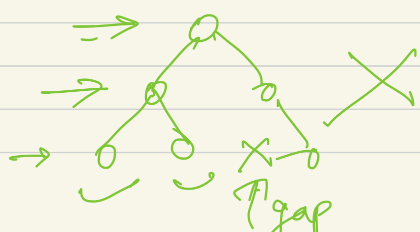
Complete Binary trees of height h ⇒ $2^{h+1}-1$ nodes

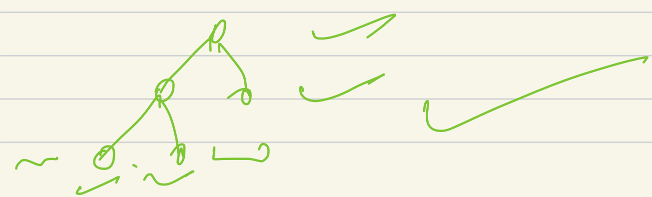




Almost / nearly complete Binary trees ⇒ Nodes are filled from top to bottom & left to right (at any level) with no gaps left

⇒ At the last level, you may not have all the nodes



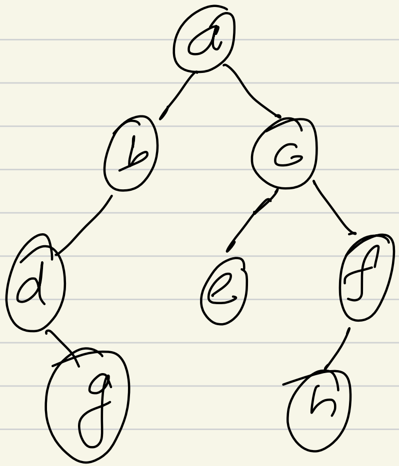


Operations on a binary Tree

Main application comes with special types of b. tree (with more restrictions on structure)

* Main operation is Traversal
  - 'visit' each node once
    → What should be the order?

PreOrder	Inorder	Post-order
- root	- left s. tree	- left s tree
- left s tree	- root	- right s tree
- right s tree	- right s Tree	- root



Preorder root, left, right

a, b, d, g, c, e, f, h

Inorder left, root, right

d, g, b, a, e, c, h, f

Post order left, right, root

g, d, b, e, h, f, c, a

for a node x

- Key (x) = data stored in x [e.g. c]
- left (x) = left child of x [e.g. e]
- right (x) = right child of x [e.g. f]
- parent (x) = parent of x [e.g. a]

= NIL if doesn't exist

Preorder (x) root, left, right

if $x = NIL$ return

visit (x) // $\Rightarrow$ print, additions ...

preorder (left [x])

preorder (right [x])

main crux

Inorder (x)

Inorder (left [x])

visit (x)

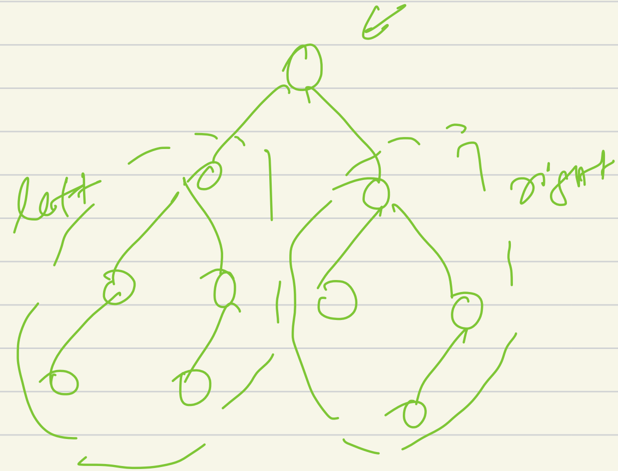
Inorder (right [x])

Postorder

Complexity :- Constant # of operations on each node $\leq cn$

$= O(n)$

* What is the size of the trees? (# nodes)



1 + size(left) + size(right)

```
int find Size (x)
    Count = 0;
    if (x = NULL) return 0;
    C1 = find Size (left[x]);
    C2 = find Size (right[x]);    return C1+C2+1;
```

Suppose each node stores an integer value.

* Find the sum of the values of nodes

* Count the elements $<$ / $>$ a certain value

< 10 ?

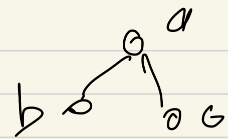
$O(n)$

* Count the number of leaf nodes
(# children = 0)

How do you represent a binary tree?

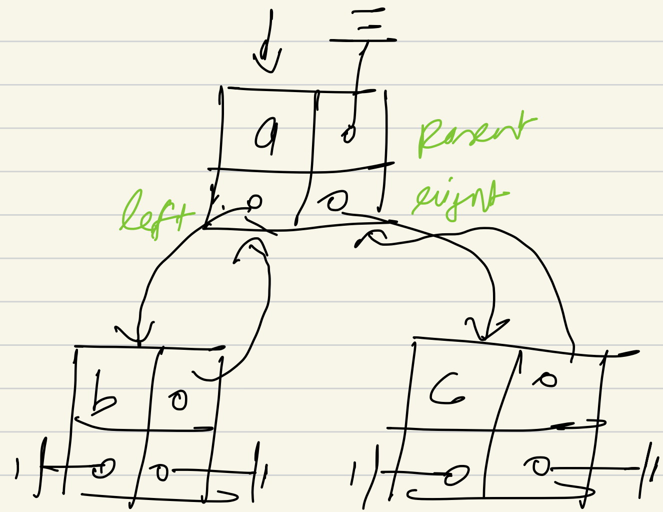
```
typedef struct node{
        int key;
        struct node *left, *right, *parent;
} NODE;
```

Represent tree by a pointer to the root



* left & right pointers
are always there

* parent may / may not
be there



put some restrictions

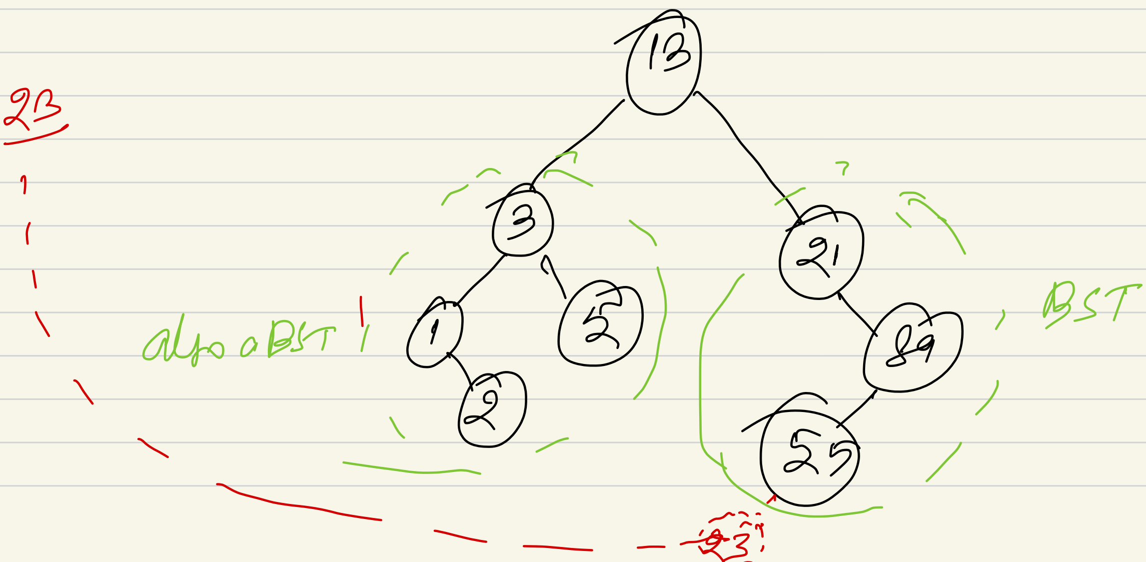
special kinds of binary trees

Binary Search Tree (BST)

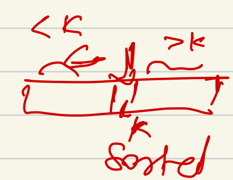
Binary Search Tree (BST)

BST is a binary tree such that for any node (root), the values in its left subtree are smaller than the value of the node, and the values in the right subtree are greater than the value of the node.

Insert 23



Search 89



Main Operations

- Search: ✓
- Insert.
- Delete.

Others

- Find Min
- Find Max
- Find Successor
- Find Predecessor

Complexity

$= O(h(T))$

$= O(n)$

Pain points

Search (x, k)

if (x = NIL) return not found;

if (key[x] = k) return found;

if (key[x] > k) return search (left[x], k);

if (key[x] < k) return search (right[x], k);

Searching in BST resembles binary search in sorted arrays

Insert operation in BST

If the value is already present in BST we return the same tree.

Otherwise, we insert the new value as a leaf node in an appropriate position.

```
Insert (x, K)
    if K = Keys[x]   return
    if K < Keys[x]
        if left[x] = NIL
            add K as left child of x
        else Insert (left[x], K)
    else
```

define a new node

put K as the key of that node

left, right = NIL

parent = x

x: left[x] = new node

Create a BST, print elements (some traversal order)

Search

Insert → Copy

→ C program

BST

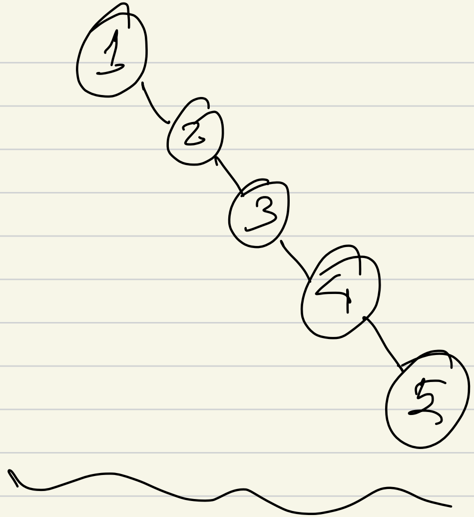
Search	$O(h)$	$O(n)$
Insert	$O(h)$	$O(n)$

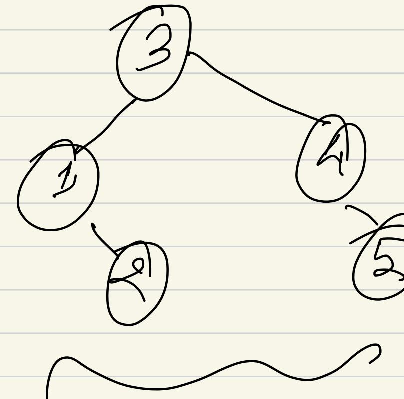
set $\{1, 2, 3, 4, 5\}$

{BST}

Sorted array

1	2	3	4	5





Separate BSTs

1, 2, 3, 4, 5

Given a BST, can you print all the values in a sorted order?

Hint: Traversal? In-order

1, 2, 3, 4, 5

Find Min(x)

while left[x] ≠ NIL

x = left[x]

return x

Find Max(x)

while right[x] ≠ NIL

x = right[x]

return x

O(h) → O(log n)

O(n) – Traversal

Find Successor : immediately next value in the sorted order

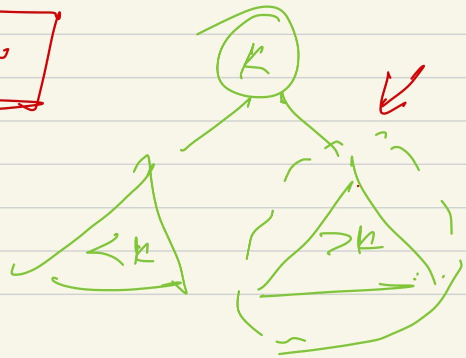
O(h)?

If ∃ a right s.t

↳ $\boxed{\text{min. of right s.t.}}$

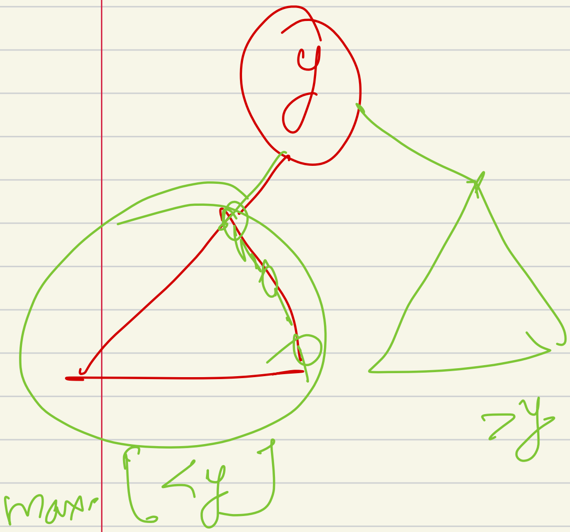
If no right s.t.

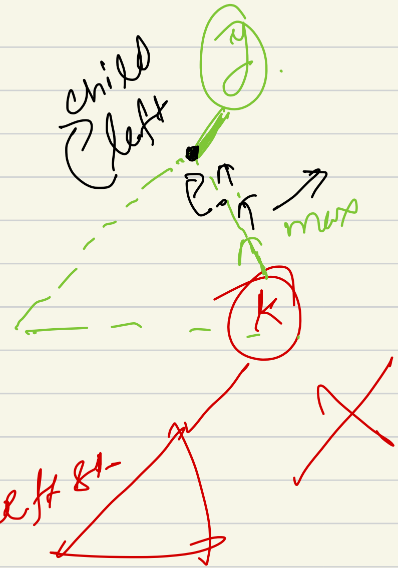
??



min. of all values larger than k

right s.t.





k is the right child

~~successor?~~

Let y be the successor

k

k will lie in the left st. of y

it will be the ~~min~~ max element

k will be predecessor of y

prev. element inserted.

You're to find a node such that K is the largest in its left subtree

Keep following the parent pointers until you find a node that is left child of its parent

return parent

Suppose we're the parent pointer

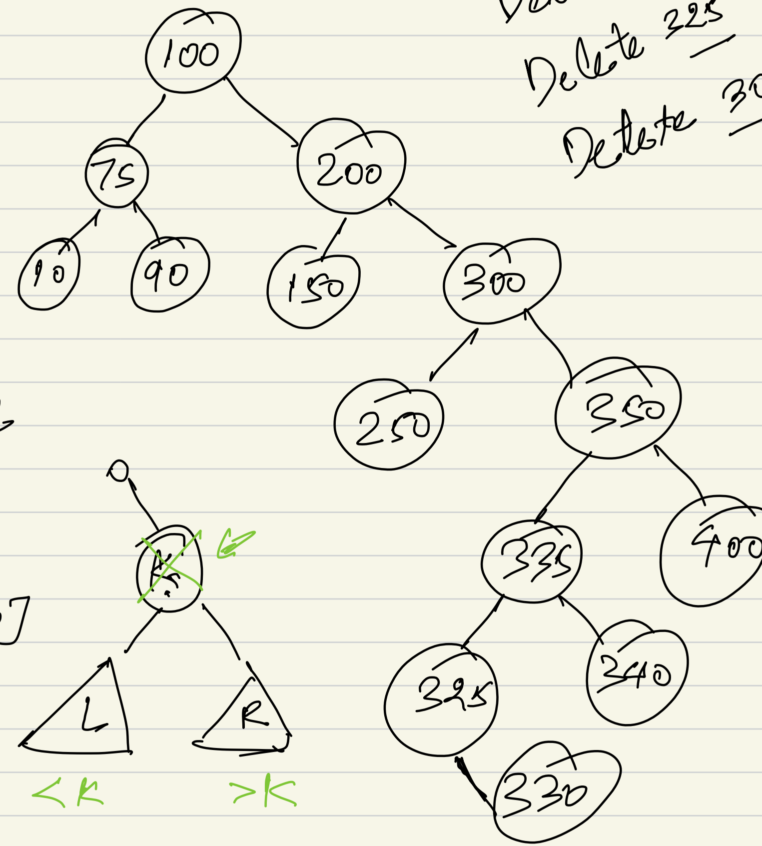
```
BST- successor (x)
    if right[x] ≠ NIL
        return findmin( right[x] )
    y = x.p          // parent[x]
    while  y ≠ NIL  &  x = y.right  // right[y]
        x = y
        y = y.p
    return y
```

BST- predecessor (x)

BST - DELETE

Delete a node from BST - x

Delete 150
Delete 225
Delete 300



Case 1: If x has no children

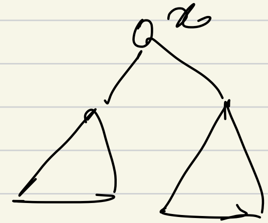
⇒ Remove it by modifying the parent to replace x with NIL as the corr. child.

[x is left/right child of p] replace corr. pointer by NIL

Case 2: If x has just one child [one sub-tree]

⇒ We elevate the child to take x's position in the tree

Case 3: If x has 2 children



Take the min. value from the right subtree of x : y

1. $Key[x] = Key[y]$
2. Delete the node y using either Case 1 or 2

[because y can't have a left child]

Alternate :-

find the predecessor of x (in left s.t.)

Repeat similar steps

Each of the operations in BST T

- search
- insert
- delete

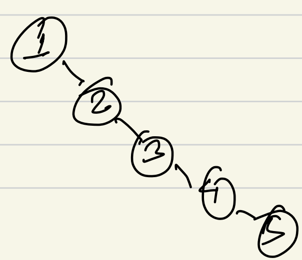
take $O(h)$ time

in the worst case

$O(n)$

Height of a BST depends on the sequence in which the values are inserted.

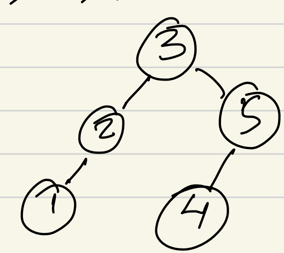
1, 2, 3, 4, 5



$\frac{4}{n-1}$

1, 2, 3, 4, 5

3, 2, 1, 5, 4



2

Prove/ Disprove: The maximum height of a BST is achieved only when the values are in sorted order (increasing / decreasing)

$O(h)$ $\qquad\qquad$ $O(n)$

Can we somehow ensure the height of a BST to be $O(\log n)$?

The idea :→ Modify the insert & delete functions in such a way that the height of the tree at any time does not grow too much & stays not far away from the best possible height (logarithm in the size of the tree)

* If such adjustments can be made efficiently we keep the height bounded by $O(\log n)$. ⇒ All major opⁿ are $O(\log n)$

Keeping the height of a trees with n nodes limited by an $O(\log n)$ value is commonly known as height-balancing.

a tree with height $O(\log n)$ is called a height-balanced trees

- Red-Black Tree
- AVL trees ✓

AVL Trees

(height balanced BSTs)

admissible trees

$h = O(\log n)$

Defining Property of AVL Trees:→

Let u be any node in an AVL tree. Let L & R be the left and right subtrees of u. Then, we must have

$$|h(R) - h(L)| \leq 1$$

In other words, the height of the left & right subtree of any node in an AVL tree differ by at most one.

* If we can maintain this property ⇒ $h = O(\log n)$
* How do we maintain this property (efficiently)

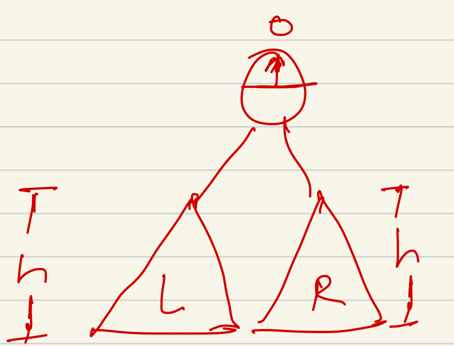
In addition to key, left, right, (parent) a node in an AVL tree should maintain the balance factor of the node.

$\cancel{h(L) - h(R)}$ $\quad$ $h(R) - h(L)$

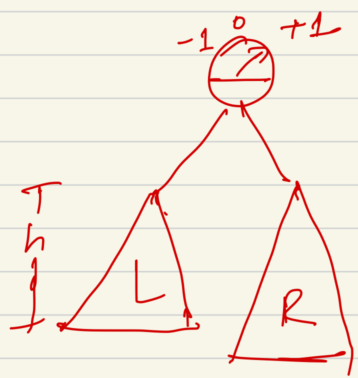
Values 0, +1, -1 can be used

[No need to store ind. heights]

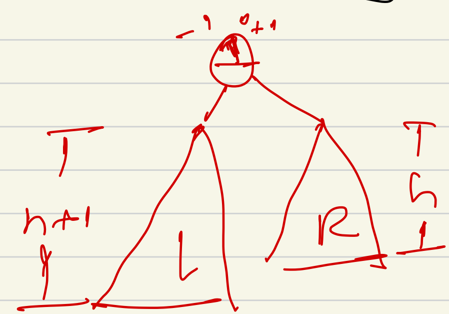
[If by insert/delete balance factor goes beyond do sth.]



$h(R) - h(L) = 0$



$h(R) - h(L) = 1$



$h(R) - h(L) = -1$

$h = \Omega(\log n)$ $\quad\quad$ $h = O(\log n)$

$|h(R) - h(L)| \le 1 \Rightarrow h = O(\log n)$

Given an AVL tree of height h let m_h be the min number of possible nodes.

	Left & right subtrees		
L	$h-1$	$h-1$	$h-2$
R	$h-1$	$h-2$	$h-1$
		2	3

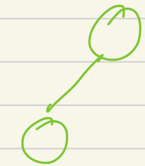
at least one more node that the min possible nodes in Case 2 / Case 3

Assume left & right subtrees are AVL trees with min # of nodes

$$M_h = M_{h-1} + M_{h-2} + 1$$

$M_0 = 1$ $\qquad$ $M_1 = 2$

$$\underbrace{M_h + 1} = \underbrace{M_{h-1} + 1} + \underbrace{M_{h-2} + 1}$$



Let $N_h = M_h + 1$

$$\Rightarrow \quad N_h = N_{h-1} + N_{h-2}$$

$N_0 = 2$ $\qquad$ $N_1 = 3$ $\qquad$ Recurrence for Fibonacci Numbers

$$N_h = F_{h+3} \approx \frac{1}{\sqrt{5}} \varphi^{h+3}$$

* AVL tree is also referred as a Fibonacci tree *

Let T be any AVL tree of height h with n nodes

$$n \geq M_h \approx \frac{1}{\sqrt{5}} \rho^{h+3} - 1$$

$$h \leq \frac{\log(n+1) + \log(\sqrt{5})}{\log(\rho)} - 3$$

$$\approx 1.44 \log n \quad \Rightarrow h = O(\log n)$$

$\Rightarrow$ AVL tree is height balanced $\quad |h(R) - h(L)| \leq 1$

Operations :-

Search :- Can be carried out similar to BST.

$O(\log n)$

Insertion / Deletion

→ also proceed in the same way as in BST, but it may throw the tree out of balance.

⇒ Some additional work is required to maintain the AVL property.

<u>Insertion</u> :→ Follow the procedure similar to BST.

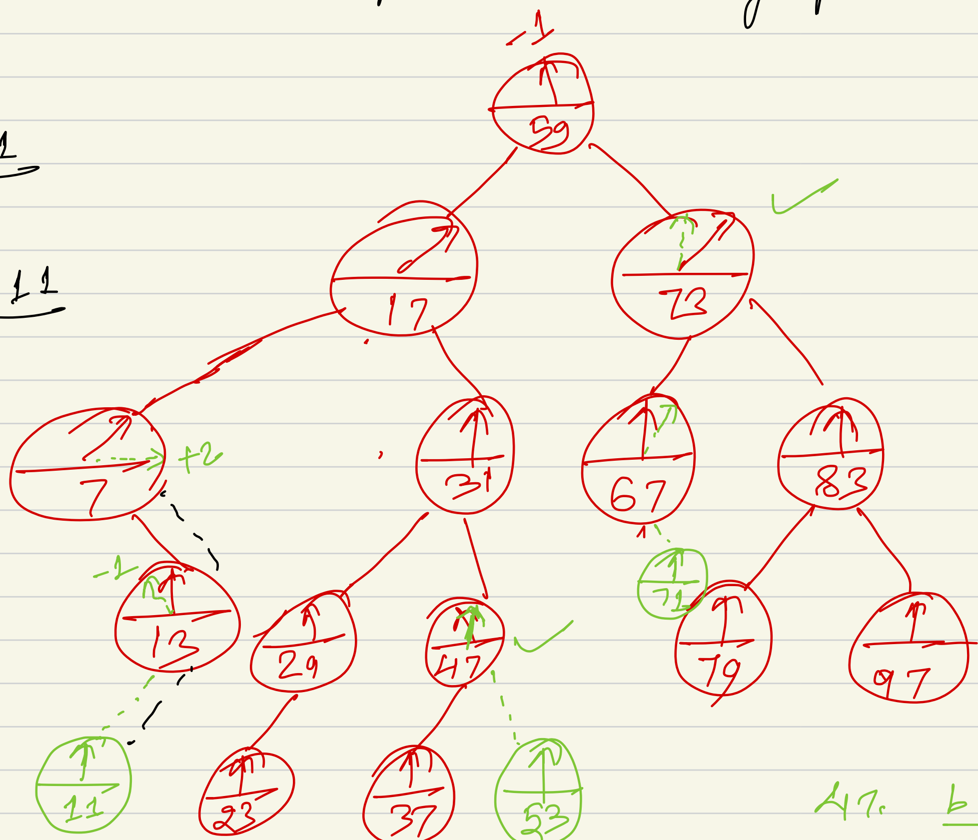
→ Finding a unique path from the root to node u, and insert the new value in a new leaf node as a child of u.

Now, we traverse the path upward, from this inserted leaf node to the root, and adjust the balance factors.

If the adjustment leads to a height difference of 2, we've to perform something special to restore height.

Insert 71

Insert 11



Insert 53

For a node u,

if balance factor changes from ±1 to 0

⇒ Now, it has become balanced but the height of the st. rooted at u doesn't change

47. b. −1 → 0

inserted right child

Overall tree at v has the same height ⇐ $\begin{cases} -1 \text{ to } 0 \Rightarrow \text{Right st. has gained height } (+1) \\ +1 \quad \text{left st has same height} \end{cases}$

Overall tree at v has gained height $\begin{cases} 0 \text{ to } \pm 1 \Rightarrow \text{One of the subtrees has gained height} \end{cases}$

→ You'll have to go up.

Let's abstract this.

Consider a situation where balance factor of -2 is detected at a node $\boxed{u}$.

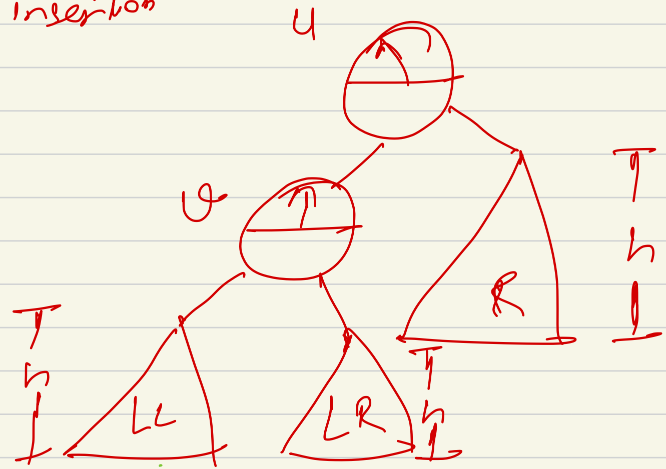
$\Rightarrow$ Prior to insertion, this would have been left-heavy (-1),

Change from -1 to -2 implies that left child v has gained height $\Rightarrow$ Left child's balance factor would have gone from $\underbrace{0 \text{ to } \pm 1}$

Let's take cases

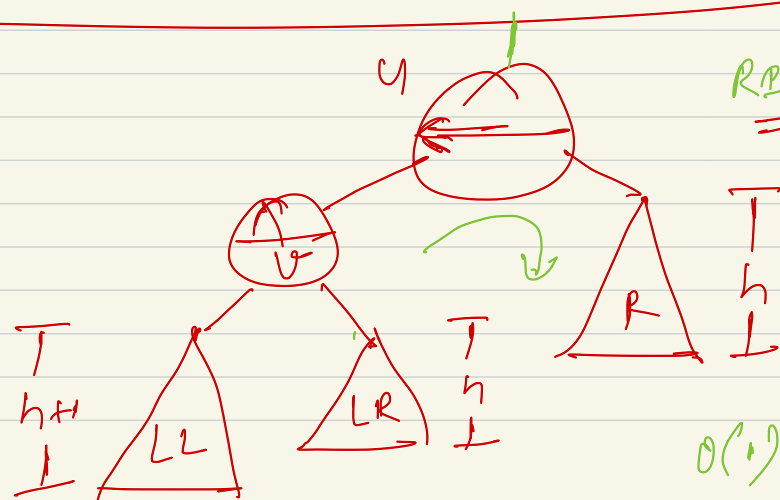
u	$\underline{-2}$	
v	± 1	
	$+1$	$\underline{-1}$

Before insertion



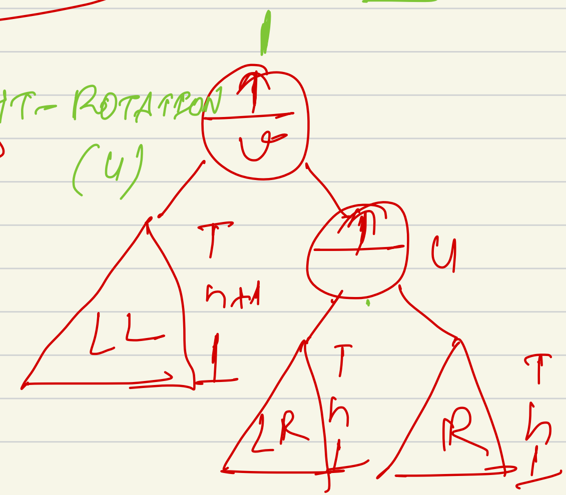
u & v have the same sign after insertion

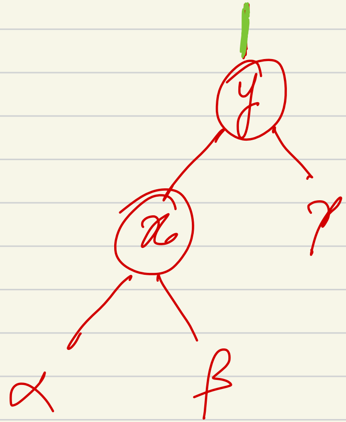
Case 1.
insertion in LL



RIGHT-ROTATION (u)

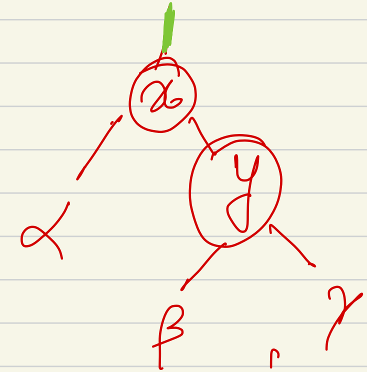
O(1)





$\xleftarrow{\text{LEFT-ROTATE}(x)}$

$\xrightarrow[\text{RIGHT-ROTATE}(T, y)]{}$



~~Ops~~ that preserve BST properties

Inorder

$$\underbrace{\alpha.key < x.key < \beta.key < y.key < \gamma.key}$$

Change some pointers

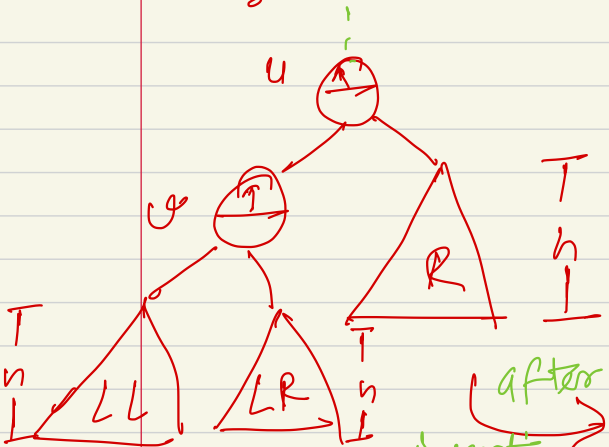
Constant numbers of pointer assignments

$\alpha.key < x.key < \beta.key < y.key < \gamma.key$

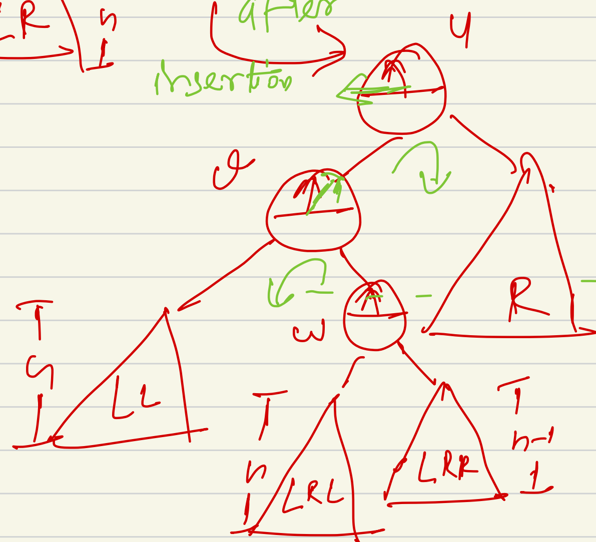
$O(1)$ - time

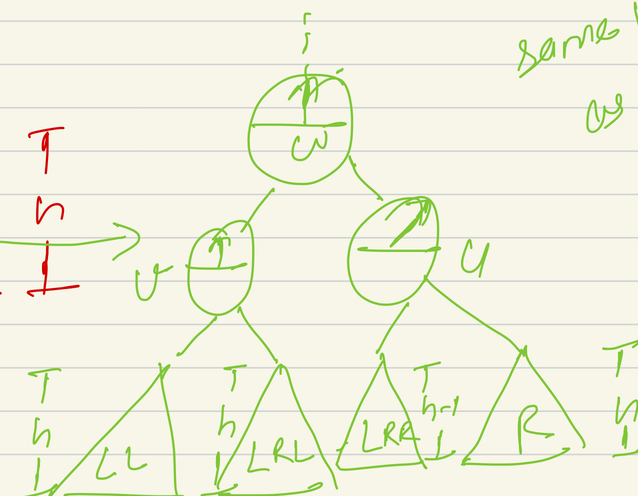
Case 2: Balance factors of u & v have opposite signs

Insertion in LR



after insertion





same height as before insertion.

* Even if insertion in LRR → w still has a 0 balance.

Insertion in AVL Trees:-

Proceed like in BST (insert a new node) $\to O(h)$

$\to$ Go up (traverse upto the root following path from this new leaf) $O(h)$

$O(h) + O(h)$

$= O(\log n)$

$= O(h)$

$\to$ Update the balance factors

either

- You're done & don't have to go up (± 1 to 0)

or

- You find a node at which balance factor is ± 2.

Restore the balance (don't have to go up)

1 rotation Case 1

2 rotations Case 2

Deletion in AVL Trees

- Follow deletion in BST
- Restore the balance
  → rotations
    ↳ at every node on the path in the worst case

$O(\log n)$. constant operations

$= O(\log n)$

Find the cases of deletion

Search
Insert
Delete

$\left.\begin{array}{l}\text{Search}\\ \text{Insert}\\ \text{Delete}\end{array}\right\} O(\log n)$

Storing Keys

Sorted array + Binary Search –

Roll No's
- insert
- delete

vs

AVL Tree + Search

	Search	Insert	Delete
Sorted Array	$O(\log n)$	$O(n)$	$O(1)$
AVL Tree	$O(\log n)$	$O(\log n)$	$O(\log n)$

Sorting based on AVL trees

n integer values : Sort them in increasing order.

Sorting — BST [in-order traversal]

$O(n)$

Creating an AVL tree for n integers

$$\underbrace{O(n \log n)}_{\text{Creating AVL tree}} + \underbrace{O(n)}_{\text{In-order traversal}}$$

AVL-Sort

$\Rightarrow O(n \log n)$

Binary Trees

Binary Search Trees

ROTATIONS

Height-balanced Binary Search Trees

(AVL Trees)

Red-black

Balanced Binary Search Trees

AVL-Sort