Tree Traversal

→ Breadth First (OR Level Order) → Visit the root then traverse next level from left to right.
↳ Top → Bottom

→ Depth First
- Inorder
- Preorder
- Postorder

→ Process the the root, then process the left subtree then we process Right Subtree.

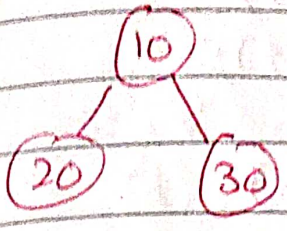
Recursive →
- Traverse Root
- Traverse left Subtree
- Traverse Right Subtree

total 3! permutation

Out of these 3! permutation, 3 popular are used :-

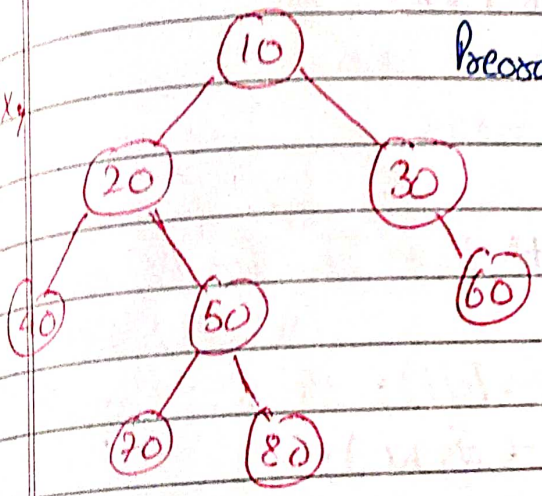
1. Inorder (Left Root Right)
2. Preorder (Root Left Right)
3. Postorder (Left Right Root)

Ex:



Inorder : 20 10 30
Preorder : 10 20 30
Postorder : 20 30 10

Ex:



Preorder ~~Inorder~~ :- 10 20 40 50 70 80 30 60
Inorder :- 40 20 70 50 80 10 30 60
Postorder :- 40 70 80 50 20 60 30 10

→ Implementation of Inorder Traversal

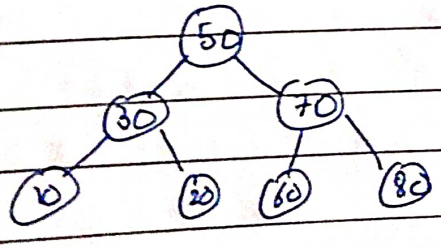
```
void inorder (Node *root) {
    if (root != NULL) {
        inorder (root → left);
        cout << root → key << " ";
        inorder (root → right);
    }
}
```

Time : O(n)
Space : O(log n) ≈ O(h)
↳ height of Binary tree

BINARY TREE (GFG)

	Array (Unsorted)	Array (Sorted)	Linked List	BST (Balanced)	Hash table
Search	O(n)	O(log n)	O(n)	O(log n)	O(1)
Insert	O(1)	O(n)	O(1) or O(n) → Sorted LL	O(log n)	O(1)
Delete	O(n)	O(n)	O(n)	O(log n)	O(1)
Find Closest	O(n)	O(log n)	O(n)	O(log n)	O(n)
Sorted traversal	O(n log n)	O(n)	O(n log n) O(n) in sorted LL	O(n)	O(n log n)

① For every node, Keys in left side are smaller and keys in right side are greater.
② All keys are typically considered as distinct.
③ Like linked list, it is a linked data structure. → not cache friendly



④ Implemented in C++ as map, set, multimap and multiset. (self balancing BST)

Date :
Page No.

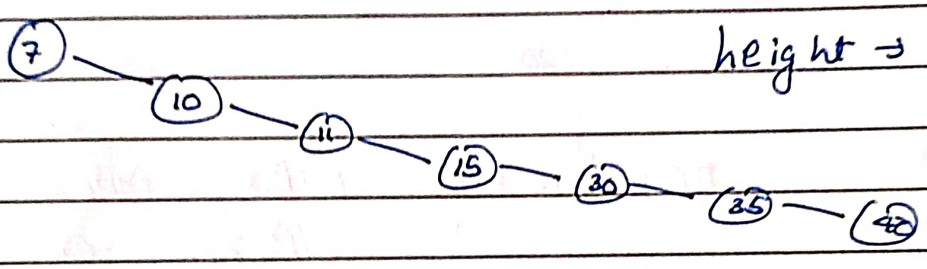
Self Balancing BST

Idea : keep height as O(log n)

Background : Same set of keys can make different height BST's.

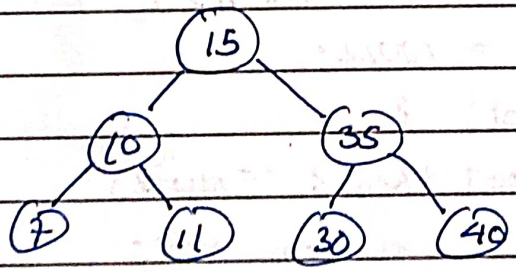
↳ height totally depends on the order in which we insert keys in BST.

eg, Order 1 : 7, 10, 11, 15, 30, 35, 40



height → n

Order 2 : 15, 10, 7, 11, 35, 30, 40



height → log n.

* If we know keys to be inserted in advance, we can make perfectly Balanced BST.

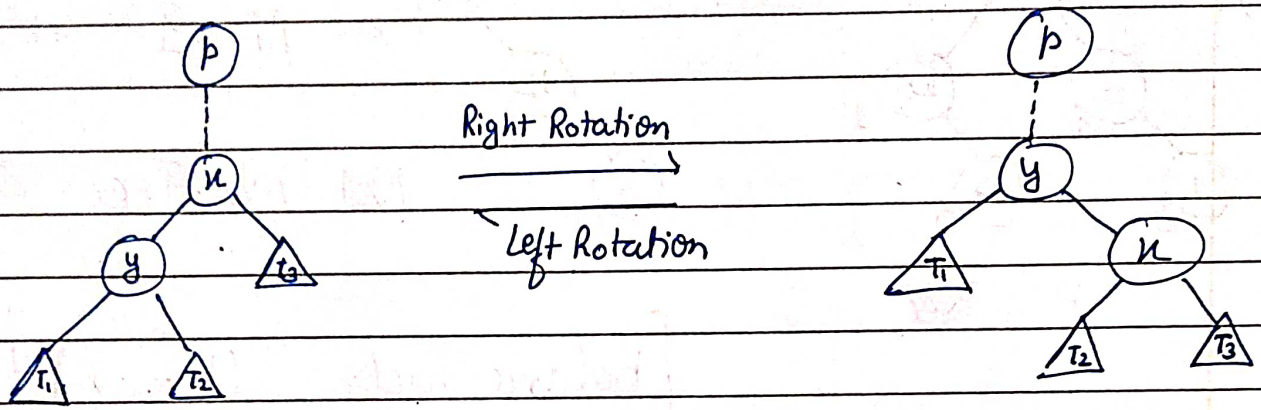
How to ↳ Sort these keys, make mid key as root and half as left and half as right child and follow this for respective left and right half.

Date :
Page No.

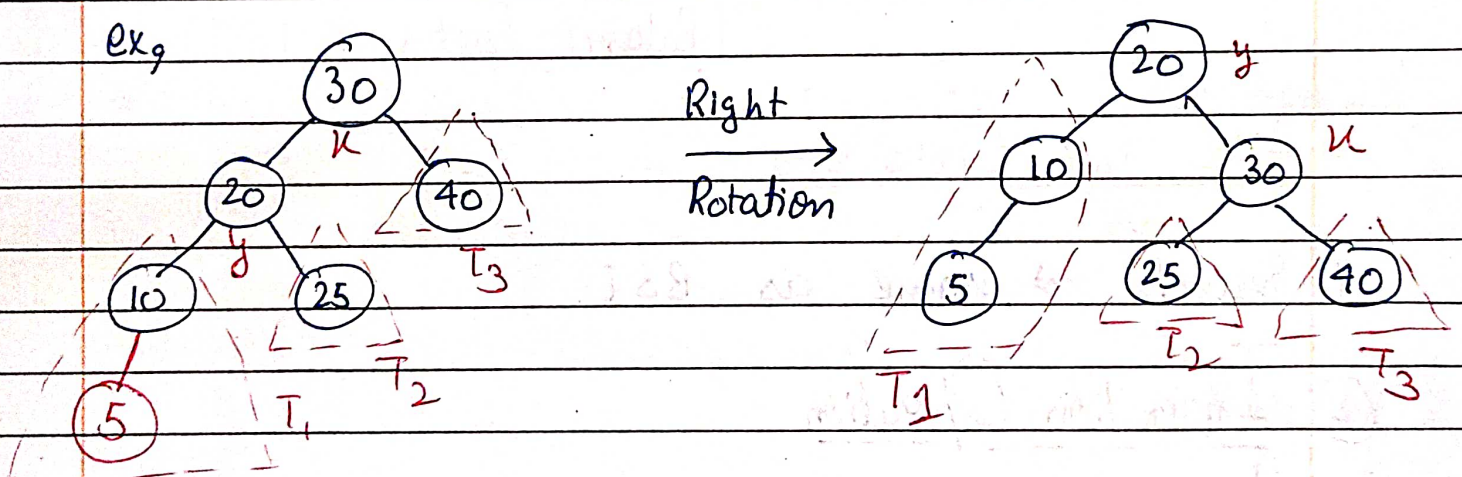
Q How to keep it balanced when random insertions and deletions happening ?

→ The idea is to do some restructuring (or re-balancing) when doing insertions / deletions.

Rotation :- O(1) operation.



Ex:



5 is inserted, then tree becomes unbalanced.

Examples of Self Balancing BST → C++, maps, set use RB tree

AVL Trees , Red Black Tree

↓ Strict in terms of Balance

↓ Less strict w.r.t. in terms of balance. (Less structuring needed)

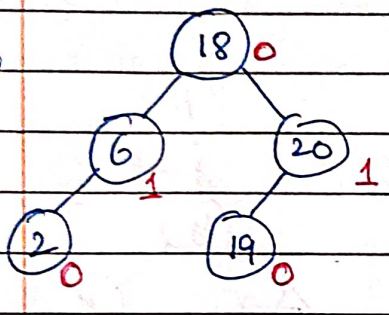
Date :
Page No.

AVL TREE

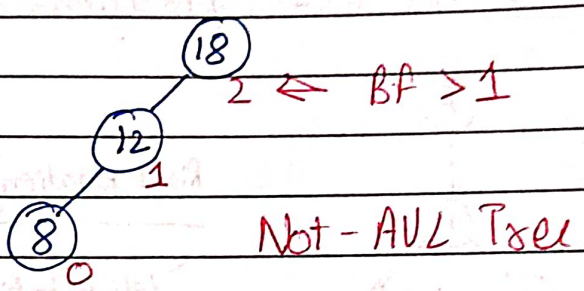
(1) It is a BST (for every node, left subtree is smaller and right is greater)

(2) It is a ~~Unbalanced~~ balanced tree (for every node, diff. b/w left and right heights does not exceed 1 (one)).

ex;



AVL Tree



Not - AVL Tree

Balance Factor = |lh − rh|

Balance Factor ≤ 1

⇒ AVL Tree Operations

(1) Search → same as BST.

(2) Insertion Operation

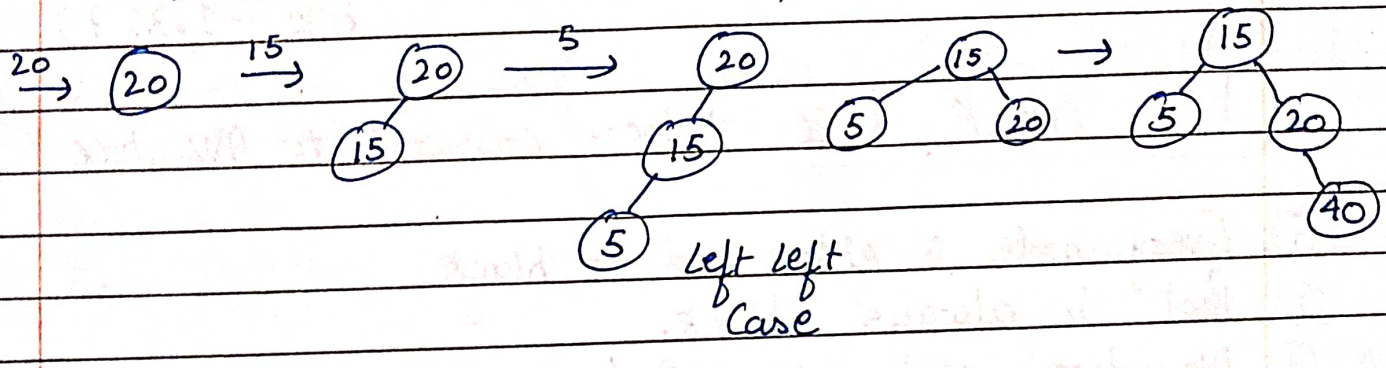
↳ (i) Perform normal BST insert
(ii) Traverse all ancestors of newly inserted node from the node to root.
(iii) If find an unbalanced node, check for any of the below cases :-

(a) Left Left) Single Rotation
(b) Right Right

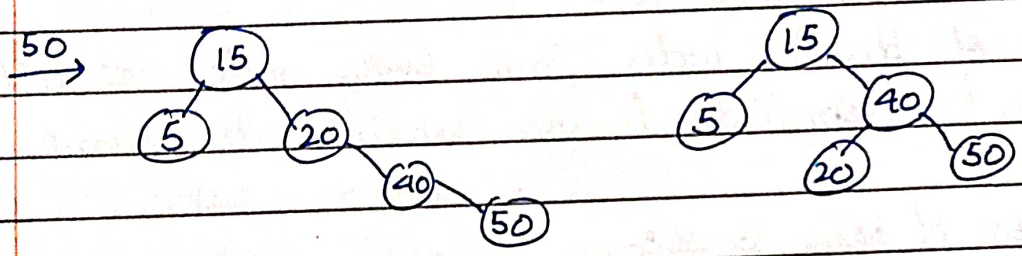
Date :
Page No.

(c) Left Right
(d) Right Left
) Double Rotation

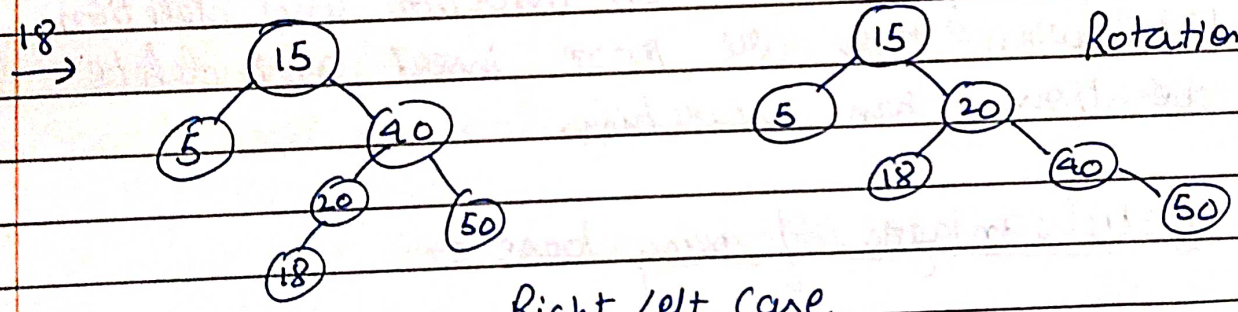
Ex, Insert: 20, 15, 5, 40, 50, 18



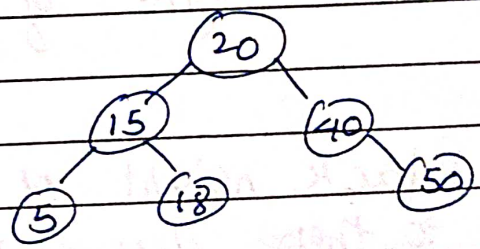
Left Left Case



Right Right Case



Right Left Case



Time Complexity : O(log n)

Date :
Page No.

height of AVL Tree

$$h < c \log_2 (n+2) + b$$, where

$c \approx 1.4405$

$b \approx -1.3277$

- **Red-Black Tree** → loose compared to AVL tree.

1. Every node is either red or black
2. Root is always black.
3. No two consecutive Reds
4. Number of black nodes from every node to all of ~~the~~ its decendent leaves should be same.

Advantages of being loose :-

1. Less no. of rotation on insertion and deletion.
2. Used when there are more insert and delete operations than searching.

Disadvantages of being loose :-

1. Max height of tree increases → time to search increases.

* Black height of all nodes should be same to their decendent leaves.

Date :
Page No.

* No. of nodes on the path from a node to its farthest descendant leaf should not be more than twice than the no. of nodes of the path to its closest leaf.

i.e we are allowed to have diff of height more than 2.

- **Applications Of BST** → more like self balancing BST

1. To maintain sorted stream of data (or sorted set of data)
2. To implement doubly ended priority queue.
3. To solve problem like:
   (a) Count smaller / greater in a stream.
   (b) Floor / Ceil / Greater / smaller in a stream.

→ Search, Insert, Delete in O(log n) → Use self balancing BST.
Sorted traversal in O(n)

→ When we have only search, insert, delete → use hash table.
or subset of these

- **SET in C++ STL**

→ Store data in sorted data every time we insert the data in a container.

Date :
Page No.

- find (ele) → return iterator to that element ele, if not then end() iter.
- count (ele) → return 0 if element not found
  1 → if element is not found.
- erase (ele) → remove element from set or group of elements.
  erase (it, s.end())
- lower_bound (x) → return iterator to element
  x → if present in set
  next greater → if not present in set.

  if x is greater than greatest than this return end() iterator.

- upper_bound (x) →
  - if x is present → its next greatest is returned
  - x not present → its next greatest in set is return.
  - x greater than gretest → end() iterator is returned.

* Set is build on top of self balancing BST. (RB tree)

→ Internal working and Time Complexities

begin(), end()
rbegin(), rend()
cbegin(), cend()
size(), empty()
crbegin(), crend()
→ O(1)

Date :
Page No.

insert(), find()
count(), lower_bound(),
upper_bound(), erase() } O(log n)

erase (iterator) ⟶ Amortized O(1)

* Applications of Set
  - Sorted Stream of Data
  - Doubly Ended Priority Queue
    - Both max and min item in Priority Queue.

° MAP in C++ STL → All values are always in sorted order.

→ Uses self Balancing BST, (RB tree)
→ Used to store key, value pairs.
→ Items ~~by default~~ ordered according to key.

m[10] → if 10 is not present insert 10 in map.

m.at(10) → If 10 is not present, exception is thrown.

other functions work similar to set in C++.

[] → O(log n)
at() → O(log n)

~~find~~ erase (iterator) → on average O(1) time.