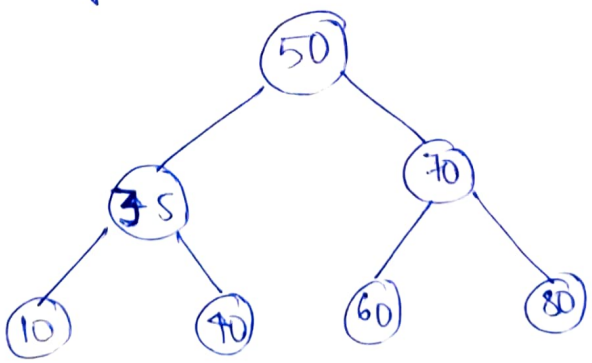
Binary Search Trees

	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) O(n) in sorted	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(1)
Sorted Traversal	O(n log n)	O(n)	O(n log n). O(n) in already sorted	O(n)	O(n log n).

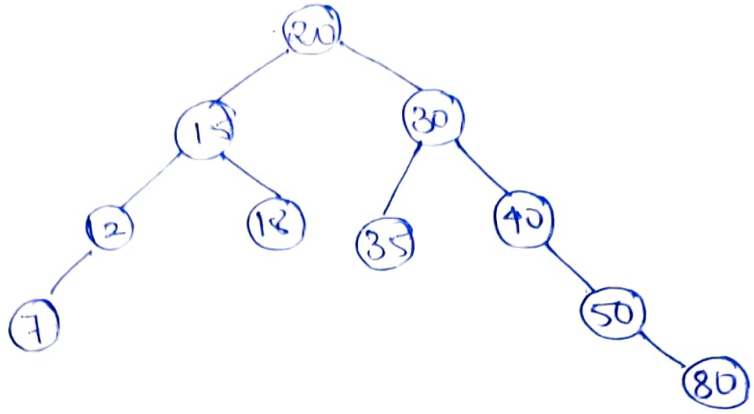
- BST is preferred when we've to find closest i.e ceil, floor etc.
- Data is organised in such a way that it reduces more than half.
- Data is organised on the binary search algo of arrays
- keys on the left are smaller and keys on the right are greater



In C++, implemented as map, set, multimap & multiset

Create a BST

insert 20, 15, 30, 10, 50, 12, 18, 35, 80, 7



Search in a BST

```
bool searchBST (Node * root, int val) {
    if (!root) return false;
    if (root → data == val) return true
    if (root → data > val) {
        return searchBST (root → left, val);
    }
    return searchBST (root → right, val);
}.
```

worst case time - O(h)

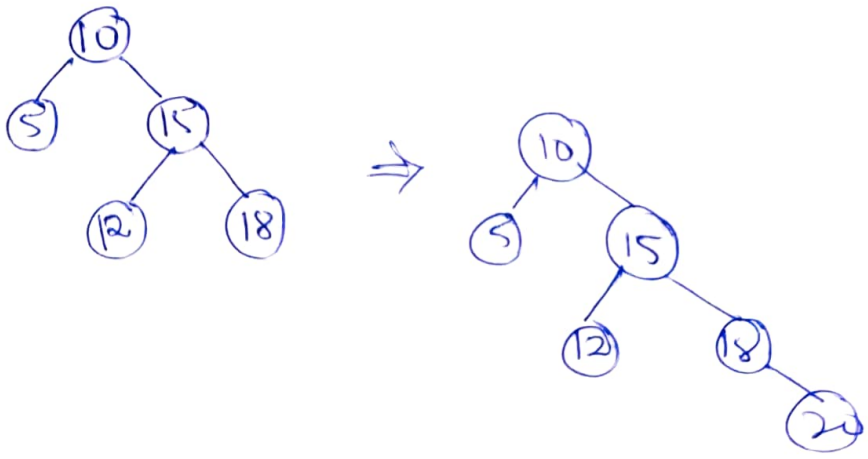
Aux space - O(h).

best case comp → O(logn).

Insert in BST

I/P → x = 20

BST →

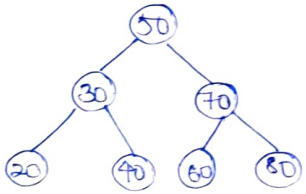


```
Node *insert (Node *root, int x) {
    if (root == NULL)
        return new Node(x);
    if (root->data > x)
        root->left = insert (root->left, x);
    else if (root->data < x)
        root->right = insert (root->right, x);
    return root;
};
```

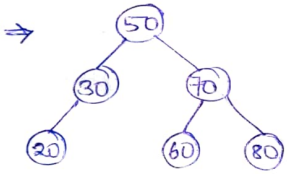
Iterative → first search the node, keep track of parent and join the node to the parent.

Deletion of Node in BST

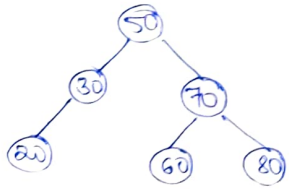
I/P → 40



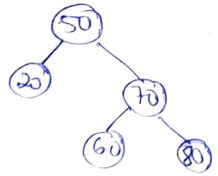
O/P ⇒



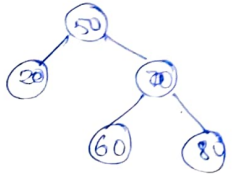
I/P → x = 30

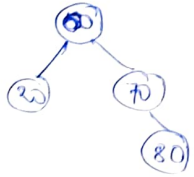


O/P →



I/P - x = 50





If the root is deleted it can be replaced by closest lower/greater

If the ~~root~~ root is to be deleted, it ~~can be~~ will be replaced by closest lower value or closest higher value.

① You can make a rule that either you'll pick closest greater value or of closest smaller value.

* As the inorder traversal of BST is always sorted,
  - If we pick the smallest closest ⇒ inorder predecessor
  - greatest closest ⇒ inorder successor.

There can be three conditions

① both child are null & deleted node is the leaf node. → delete node and return NULL

② ~~both~~ either child is NULL → delete node & return then non-NULL child

③ both child are not NULL

↳ get the successor of the root
  ↳ left most child of the right subtree

root->key = succ->key
and we will delete the succ->key.

Algorithm

① getting the successor node:-

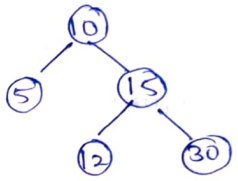
```
Node *getSuccessor (Node *curr) {
    curr = curr->right;
    while (curr != NULL && curr->left)
        curr = curr->left;
    return curr;
};
```

```
Node *delNode (Node *root, int x) {
    if (!root) return root;
    if (root->Key > x)
        root->left = delNode (root->left, x);
    else if (root->Key < x)
        root->right = delNode (root->right, x);
    else {
        if (root->left == NULL) {
            Node *temp = root->right;
            delete root;
            return temp;
        }
        else if (root->right == NULL) {
            Node *temp = root->left;
            delete root;
            return temp;
        }
        else {
            Node *succ = getSuccessor (root);
            root->key = succ->key
            root->right = delNode (root->right,
                                   succ->key);
            return root;
        }
}
```

Time complexity = O(h)

Floor of BST

- closest smaller value.

I/P → 

x = 14 → Node - 12

I/P → x = 4 same tree
O/P → NULL.

I/P → x = 30
O/P → 30.

Naive solution

Traverse the whole tree and maintain closest smaller

Efficient solution

→ use the concept of binary search.

```
Node *floor (Node * root, int x) {
    Node *res = NULL;
    while (!root) {
        if (root->key == x) {
            return root;
        }
        if (root->key > x)
            root = root->left;
        else {
            res = root;
            root = root->right;
        }
    return res;
}
```

Ceil of BST

→ greater than or equal to given key (closest).

```
Node *getCeil ( Node * root, int x) {
    if (!root) return NULL;      *res = NULL
    while (root) {
        if (root → key == x)
            return root;
        if (root → key < x) {
            ~~root~~ root = root→right;
            ~~res~~ = ~~getCeil (root→right)~~
        }
        else {
            res = root;
            root = root → left;
        }
    }
    return res;
}
```

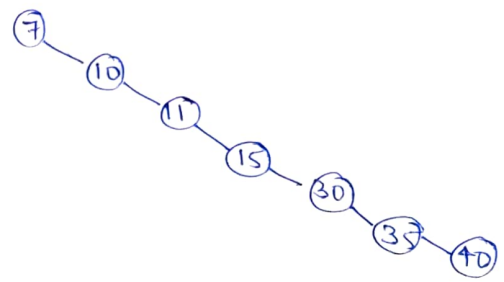
~~Left~~

Self Balancing BST

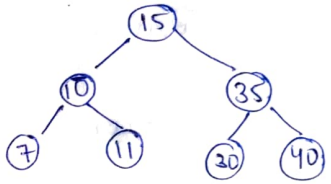
Keep the height as O(logn).

Example

order-1 - 7, 10, 11, 15, 30, 35, 40.

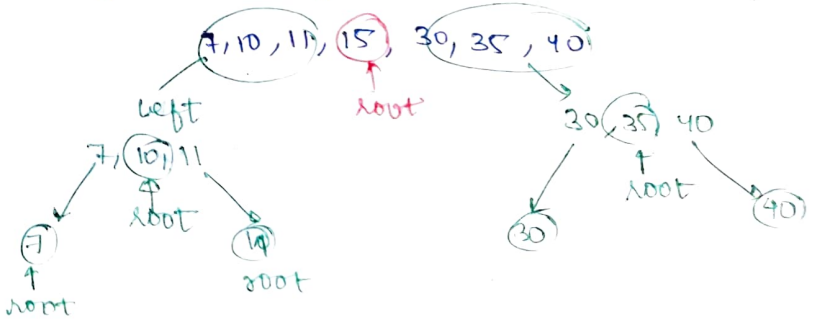


order-2 - 15, 10, 7, 11, 35, 30, 40



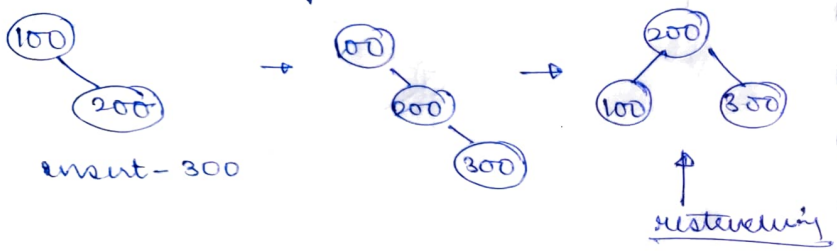
→ height = log(n)

- Height and structure is decided by height
- If the keys are known prior, we can sort the key and select the middle key as root, and recursively do the same for left and right subtrees

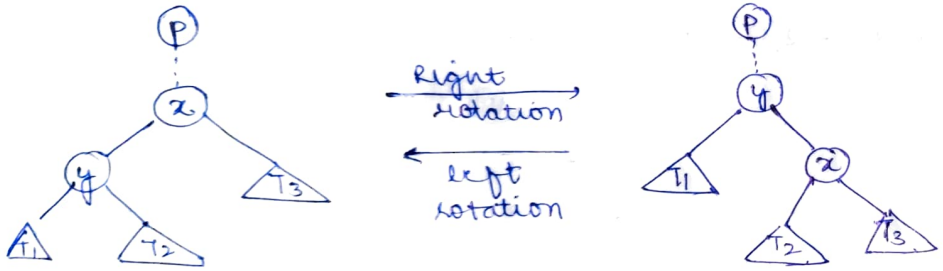


How to maintain height when user is inserting in random way.

→ Do some restructuring



Rotation



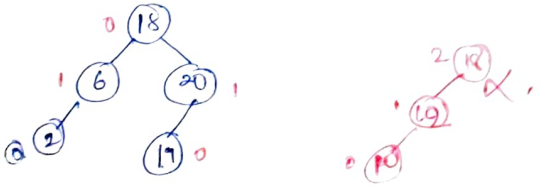
Self Balancing
- AVL Tree (Very strict in terms of height)
- Red Black tree

AVL Tree

(1) For every node the difference of left subtree and right subtree should not exceed 1

Balance Factor = $|h_l - h_r|$

BF $\leq$ 1

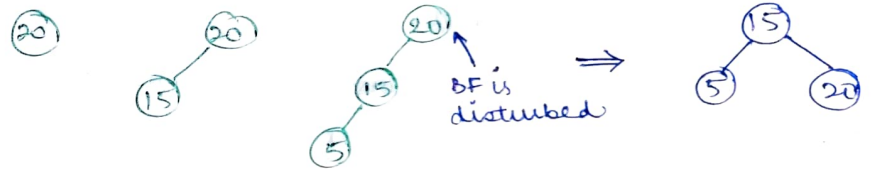


Insert operation

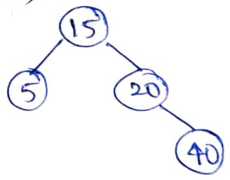
- Perform normal BST insert.
- Traverse all ancestors of newly inserted node from node to root
- If find an unbalanced node, check for any of the below cases
  1. left left) single Rotation
  2. Right Right) single Rotation
  3. left Right) Double rotation
  4. Right left) Double rotation

Order - 20, 15, 5, 40, 50, 18

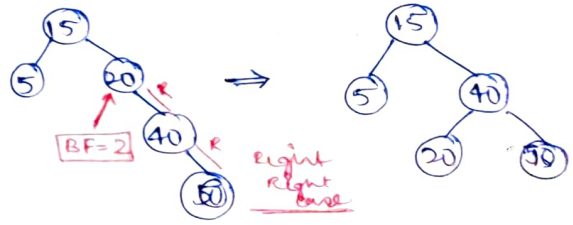
I(20) I(15) I(5)



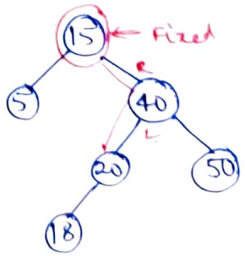
I(40)



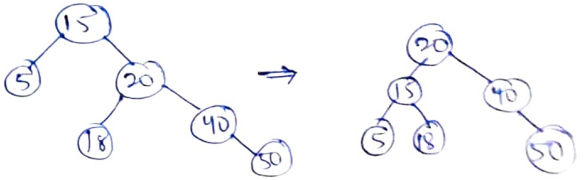
I(50)



I(18)



Right left case
↳ make it right-right case



time complexity

insertion — $O(\log n) * O(\log n)$

- $O(\log n)$ ↑ insertion
- $O(\log n)$ ↑ traversing to top to find unbalanced ancestor

$h < c \log(n+2) + b$

$c \approx 1.4405$

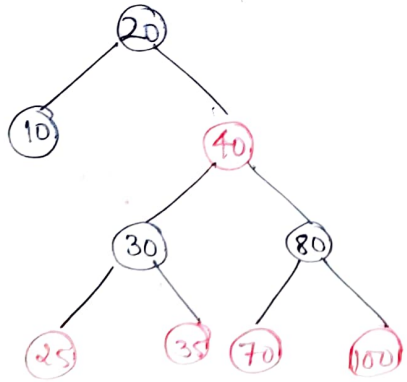
$b \approx -1.3277$

Red Black BST

- every node is either black or red
- Root is always black
- No two consecutive Reds.
- Number of black nodes from every node to all of its descendants leaves should be same

- Loose as compared to AVL tree, and have less complex insert and delete operations
- Search is more complex.

- Number of nodes to the leaves to the farthest descendant can be almost twice the number of nodes in the path to the closest leaf descendant.
- Insert and deletions are less complex and also in insertion
  1. try recoloring
  2. Rotation



Applications of BST

1) To maintain sorted stream of data
2) To implement doubly ended priority queue.
3) To solve problems like:
   (a) count smaller / greater in a stream
   (b) Floor / ceil / greater / smaller in a stream

set in C++

→ maintain data in sorted order (increasing).
→ Duplicates are not allowed.

- **insert()** → insert element in a set and maintain a sorted order
- **begin()** → points to first element of set
- **end()** → point to the location beyond last element
- **rbegin()** & **rend()** → Reverse iterators
- **find()** → search an element

```
auto it = s.find();
if (it == s.end()) → not found
else → found
```

- **clear()** → Remove all elements
- **count()** → Returns 0 / 1 (as count of any element ≤ 1).
- **erase()** → can remove
  - a single element
  - a range of elements.

```
s.erase(it1, it2)
```

↑ not inclusive

lower_bound() → auto it = s.lower_bound(5)

↳ gives an iterator to element if present

↳ if not present, give the element just greater than that

↳ If there is no element greater than x it returns s.end()

upper_bound() → auto it = s.upper_bound(x)

↳ If x is present, it returns iterator to next element

↳ If x is not present – it returns iterator to the next greater element

↳ greater than greatest ~~is not~~ ~~pata~~ ↳ s.end()

Internally

↳ Set uses Red Black Tree

insert(), find()
count(), lower_bound()
upper_bound(), erase(val) > log(n).

erase(it) — Amortized, O(1)

Map in C++ (Ordered)

↳ built using Red Black tree

↳ store a key value pair

↳ element are ordered on the basis of key

insert() → inserts element into the map

m.insert({10, 20});

operator[] → m[10] = 20;

If we access something which is not present in map.

cout << m[20];

then it will insert a value to the map key = 20
value = default int → 0

(20, 0) → inserted

at() → m.at(10) = 300 (updating)

m.at(20)

↳ if 20 is not present, it will throw an exception

m.first() → key
m.second() → values.

upper_bound() → same as set
lower_bound() → same as set.

erase → can pass a key
can pass iterator
can pass range it, to it2

Ceiling on left side

I/P = arr[] = {2, 8, 30, 15, 25, 12}

O/P → {-1, -1, -1, 30, 30, 25}

three elements = 15, 25, 30 → smaller among them

I/P = arr[] = {30, 20, 10}

→ -1, 30, 20

I/P = {10, 20, 30}

-1, -1, -1

Approach

- Empty self balancing binary tree
- Insert arr[0] into s
- print (-1) ← for 1st element (fixed).

ceil → lower bound

```
for (int i = 1; i < n; i++) {
    if (s contains a ceiling of arr[i])
        print the ceiling
    else
        print -1

    Insert arr[i] into s
}
```

K^th Smallest Element

Design a DS such that insert, delete, search & kth smallest efficiently.

① Naive approach

use a BST and traverse inorder and maintain a count. if count == k, return root → key.

Efficient approach

↦ change the structure of BST, augmented BST

```
class Node {
    int x; lcount;
    Node (x) {
        this->x = x;
        -- pointer init --
        lcount = 0;
    }
}.
```

lcount → gives of count of nodes in the left subtree

compare (lcount+1) with k.

1) if same, return root.
2) if greater, recur for left subtree.
3) if smaller, recur for right subtree with k as (k - (lcount+1))

skipping left nodes + node → skipping the ~~right~~ left subtree + node

How to maintain lcount

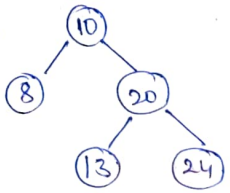
① insert something in left, increament the lcount
② while inserting on right, lcount remains unchanged.

```
Node * insert (Node *root, int x) {
    if (root == NULL)
        return new Node (x);

    if (root -> key > x) {
        root -> left = insert (root -> left, x);
        root -> lcount ++;
    }
    else if (x > root -> key) root -> right = insert (root->right, x)
    return root;
}
```

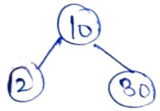
Check if the Binary Tree is BST

I/P →



⇒ Yes.

I/P



⇒ NO.

~~Naive~~ solution (super Effient) :D

Do inorder traversal of the tree, if the res array is sorted, given tree is BST. (store prev **)

2nd solution

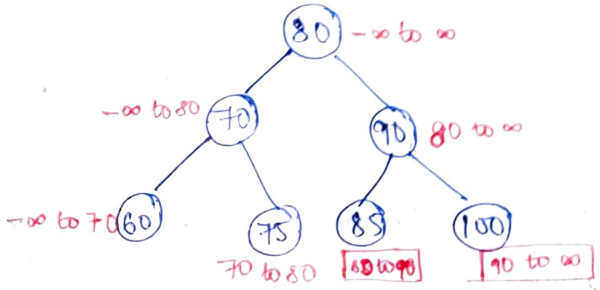
calculate max on left subtree and ⇒ left
min on right subtree ⇒ right

left < root < right

O(n²)

Efficient solution

⇒ pass range with every node.
⇒ for root, range is −∞ to +∞.
⇒ for left child → range upper bound = root→key
for right child → range lower bound = root→key




```
bool isBST (Node * root, int min, int max){
    if (root == NULL) return true;
    return ( root→key > min &&
             root→key < max &&
             isBST (root→left, min, root→key)
             && isBST (root→right, root→key, max))
}.

isBST (root, INT_MIN, INT_MAX);
```

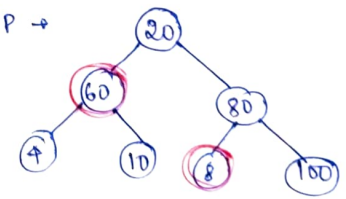
O(n)

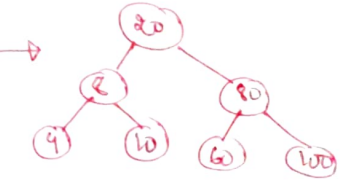
implementation of 1st solution

```
int prev = INT_MIN;
bool isBST (Node * root) {
    if (root == NULL) return true;
    if ( isBST (root→left) == false ) return false
    if (root→key ≤ prev) return false;
    prev = root→key.
    return isBST (root→right);
}.
```

Fix a BST with two nodes swapped

I/P →

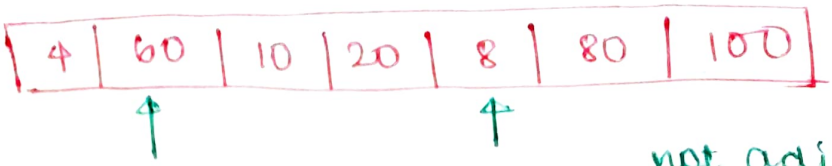




↓

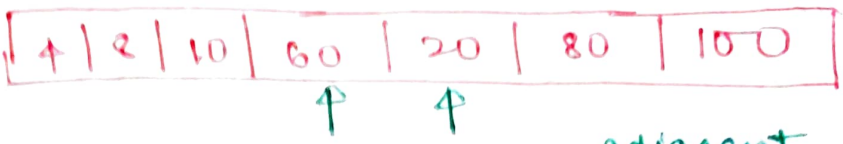
↑ 60 10 20 8 80 100

case-1



not adjacent

case-2



adjacent

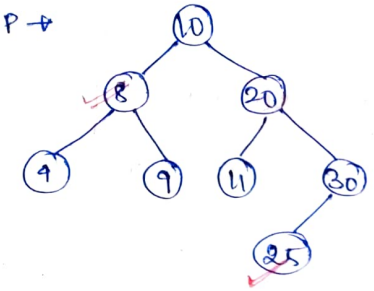
- we will update first only one time

```
prev = INT_MIN, first = NIL , second = NIL
for(int i=0 ; i<n ; i++){
    if(arr[i] < prev){
        if(first == NIL)
            first = prev;          } only set one time
        second = arr[i];   <- multiple time updates.
    }
    prev = arr[i];
}.
```

```
Node *prev = NULL, *first = NULL, *sec = NULL;
void fixBST(Node *root){
    if(root == NULL) return;
    fixBST(root->left);
    if(prev != NULL && root->key < prev->key){
        if(first == NULL)
            first = prev;
        second = root;
    }
    prev = root;
    fixBST(root->right);
}.
```

Pair with given sum in BST

I/P →



sum = 33

Yes

Naive solution

- Do an inorder traversal of the tree
- get the sorted array
- Use two pointer approach to get sum

time → O(n) + O(n)
= O(n)

Aux → O(n)

Efficient solution

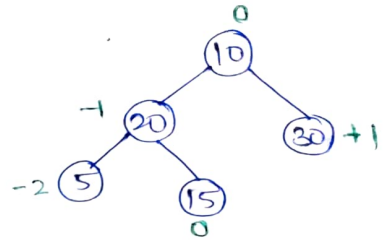
- while traversing through the tree, keep adding the nodes in the hashtable if (sum − root→key) is present in hashtable return true.

```
bool isPairSum ( Node *root, int sum,
                 unordered_set<int> &s)
{
if (root == NULL) return false;
if (pairSum (root→left, sum, s) == true)
    return true;
if (s.find (sum − root→key) != s.end())
    return true;
else  s.insert (root→key);
return isPairBST (root→right, sum, s);
}
```

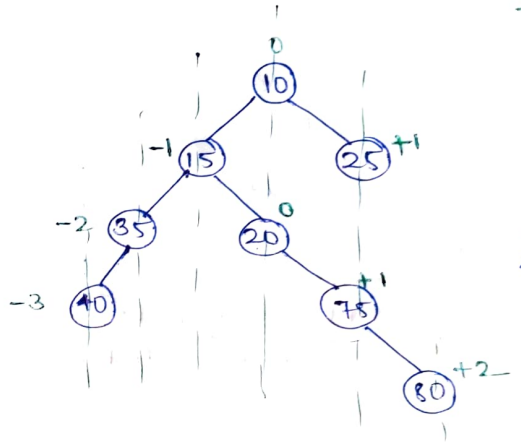
Vertical sum in a Binary Tree

↳ sum of elements having same horizontal distance

right child = +1
left child = −1



I/P



-3	-2	-1	0	+1	+2
40	35	15	30	100	80

solution

use a map (ordered)

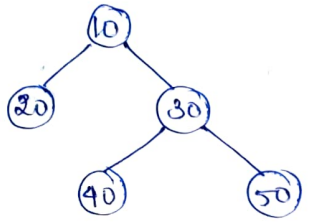
n log hd + hd

```
void vSum ( Node *root, int hd, map<int,
                                    int> &m)
{
if (root == NULL) return;
vSum( root→, hd−1, m);
m[hd] += root→key;
vSum (root→right , hd+1, m);
}

void vSum ( Node *root) {
map <int ,int> m;
vSum (root, 0 , mp)
for (auto sum : mp)
    cout << sum.second << " ";
}
```

Vertical Traversal

I/P —



20
10 40
30
50

- We'll use level order traversal, because we have to maintain order also.

→ We'll use two data structures

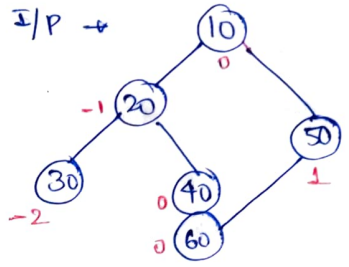
① queue
  ⊢ it will contain
    ⊢ address of node
    ⊢ horizontal distance.

② map<int, vector<int>>.

```
void VTraversal (Node *root) {
    map<int, vector<int>> mp;
    queue <pair< Node*, int >> q;
    q.push ({root, 0});
    while (q.empty () == false) {
        auto p = q.front ();
        Node *curr = p.first;
        int hd = p.second;
        mp[hd].push_back (curr->data);
        q.pop();
        if (curr->left) q.push ({curr->left, hd-1});
        if (curr->right) q.push ({curr->right, hd+1});
    }
    print map
}
```

Top view of Binary Tree

I/P →



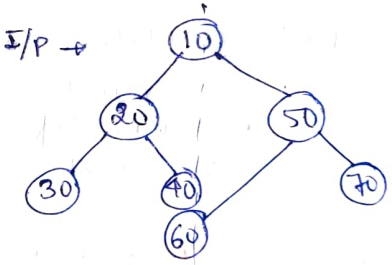
30 20 10 50

→ We'll use the same approach, but this time ~~will~~ we'll not push elements instead we'll ~~or~~ have only one time insertion for a key(hd) and will not update that.

```
if (mp.find (hd) == mp.end ())
    mp[hd] = curr->key;
```

Bottom view of Binary Tree

I/P →



→ 30 20 60 50 70

using the same concept, this time we'll update the mp[hd] every time, to get the bottom most node