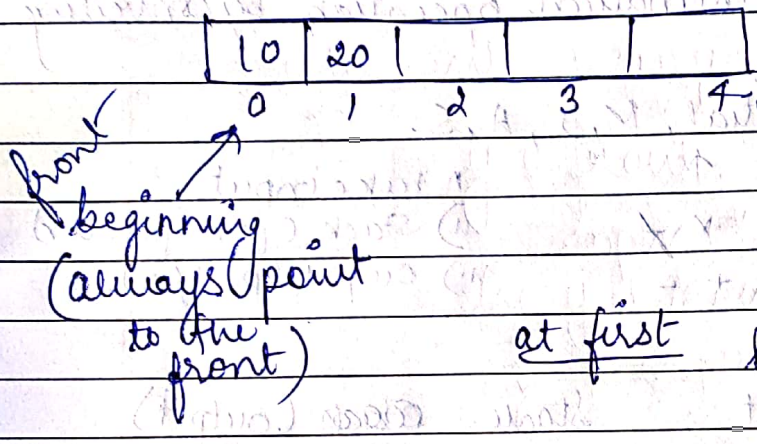
Array, Linked List → Physical data structure
Stack, Queue, Trees, Graphs → Logical data Structure.

Queue → Rear and front (pointers)
insertion from rear
deletion from front.
First come first Serve. (FIFO).



at first beginning = -1 } Queue not
top = -1 } found.
beginning = -1 no element
to insert → begin -1+1 = 0
top -1+1 = 0.
at element at 0 → 10.
to insert: begin = 0
top = 0+1 = 1

10	20	30	40	50.

at element at 1 → 20

To perform delete operations - DQueue Operations
now use beginning.
beginning = beginning + 1.
And to utilise memory, wastage in array we use circular queue

Queue as Array

Problem → when no element in array (front = rear)

- When we are inserting the value in the Queue then, rear will be incremented.

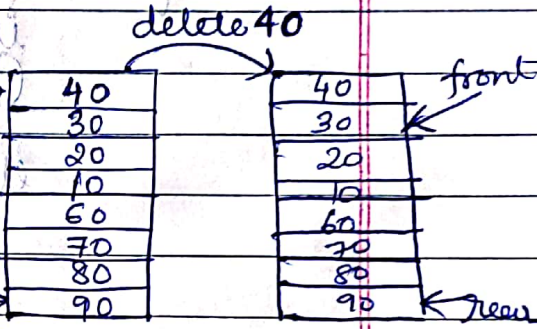
  no of elements in Queue = rear - front.

- When we are deleting the value in Queue then, we will be swapping values and keeping front fixed. (Operational cost increases) ✗ not used.

  or

  change the position of front.

front cannot overtake rear



Circular queue removes the problem of fixed size.

Problem:- array index out of Bound error when array is full.

and even if we delete the elements in an normal array we cannot utilize it because we have shifted front and no access to previous i.e. we use circular queue to remove problem of fixed size.

Queue

```
class MyQueueADTImplementation
{
    int front;
    int rear;
    int [] queue;
    int maxSize;
    MyQueueADT (int maxSize){
        queue = new int[maxSize];
        this.maxSize = maxSize;
    }
front = 0;
rear = 0;
}
```

```
public boolean enqueue (int data)
{   if (rear < maxSize)
    {queue[rear++] = data; } return true; }
    else
    { Sopln ("No space in queue"); return false;
    }
}

public int dequeue ()
{                int response=0;
    if (front == rear)
        Sopln ("Queue Empty");
    else if (front < rear)
        { response = queue[front++];
        }
    return response;
}
```

Deque (Double ended queue) or deck
insert and delete operations allowed on both ends.
operation: add first → addFirst addFront
add last → addLast
remove first → removeFirst
remove last → removeLast. removeRear
array implement / doubly linked list

head (front) tail (rear)

reference variable uses constant space complexity

order 1 remove from last

next. (head, prev. tail)

reduce complexity

Order (1)

Queue
Linked List
head → front
tail → rear

no operation like traversing allowed on queue theoritically but we can perform traversing. you cannot change the state of queue.

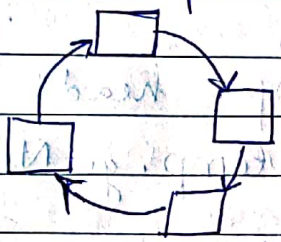
Circular queue

Starting from any element you can comeback to any particular element, round robin scheduling m implementation implemented by circular linked list (singly).

queheads —

only contain tail reference in most optimal form.

doubly circular not used because no t concept of deleting from last cause there is no last



Q how to identify head of circular linked list when you know only about its tail?

ans next to tail is head

because tail points last node and in circular reference part of last node contains address of first node (head) so tail.getNext gives head.

tail.setNext(tail.getNext.getNext())

↳ brings second element of circular linked list

Why only tail not head?

because by knowing head we have to traverse a loop to know tail costing O(n) but by knowing tail we can immediately find head.

cost of adding any node → should be at last

therefore → order (n) complexity.

Q WAP to create a circular linked list which only contains a head reference.

1) WAF to get the last node.
2) WAF to delete the last node.
   WAF to add new node at end of list.

} write complexity

```
Class CircularLinkedList
{ Node head;

public Node getLastNode()
{   Node temp = head;
    if (temp.getNext() != null)
    { temp = temp.getNext();
    }
    Node response = null;
    if (head != null)
    { Node temp = head;
      while(temp.getNext() != head)
      { temp = temp.getNext();
      }
      response = temp;
      return response;
    }
```

time complexity - O(n)

Space complexity = O(1) because there is no declaration of variable (space) multiple times

```
public deleteLastNode()
{ Node response = null;
  if (head != null)
  { Node previous = null;
    Node temp = head;
    while(temp.getNext() != head)
    { previous = temp;
      temp = temp.getNext();
    }
```

```
if ( previous != null)
        { response = temp;
          previous.setNext(head);
        }
    else { response = head;
           head = null;
         }
    return response;
}
```

How to achieve constant complexity for all the function and minimum space complexity for all functions.

Priority Queue

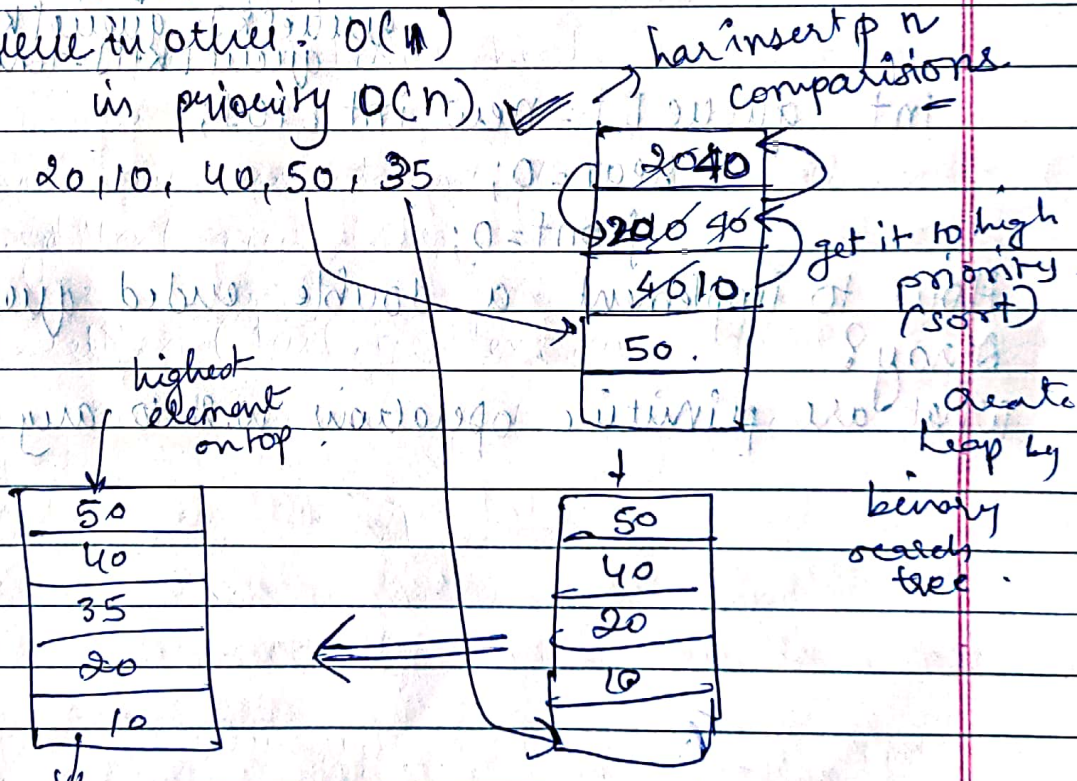
by means of an integer queue (priority decided on integer value)

Same operations except enqueue operation (addition of elements)

enqueue in other : O(n)

" in priority O(n)

enqueue - 20, 10, 40, 50, 35



When same elements occurs we sort by order one which comes first stays towards front and which comes after stays near rear

Priority Queue is used by creating heap.
heap is created by using Binary search tree.
then the complexity of enqueue is O(log k)
not log(n).

Descending priority Queue → max heap.
Ascending priority Queue → min heap.

Priority Queue Implementation using array (maxHeap)

```
public void enqueue(int data)
  { queue[rear++] = data;
for(int j=0; j<rear; j++)
  { for(int k=0; k<rear-1; k++)
      { if(queue[k] < queue[k+1]){
          temp = queue[k];
          queue[k] = queue[k+1];
          queue[k+1] = temp; }}}}
int queue[] = new int[10];
    rear = 0;
    front = 0;
```

- insert the element in array.
- sort the array in descending order.

How to implement a double ended queue using Array?

what are primitive operations w.r.to any method?

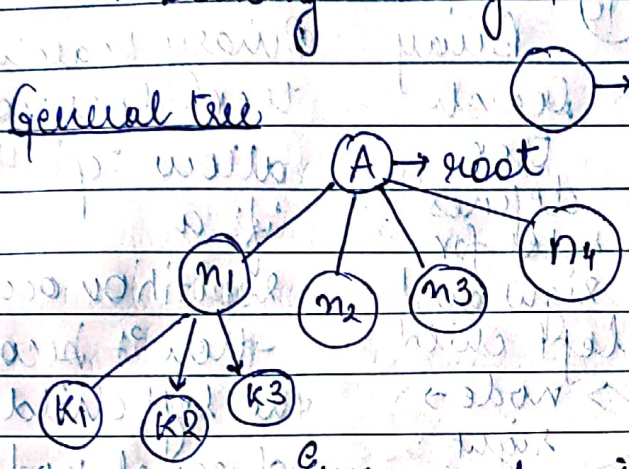
↓
queue?
dequeue?
&

Trees.

A tree is an hierarchial data structure. A tree in general can contain n number of node. The first node is called root of the tree and all other nodes are called children of the tree. A tree can be empty or can contain a single node.

Root - a node which does not have parent.
↳ Starting node of the tree

General tree



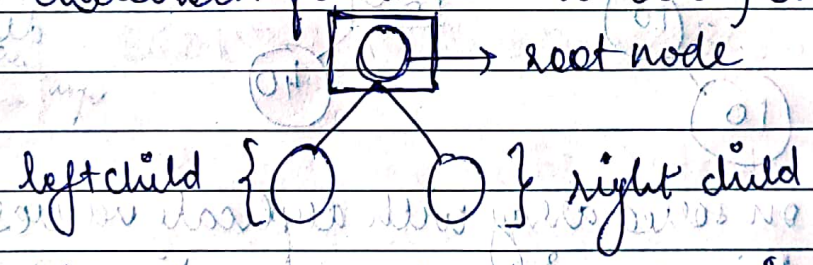
Every node can contain multiple node. or no node.

→ Application
used to represent file directory in an computer.

Every node will have dynamically growing data structure and will have list of childnodes.

Binary Tree

can be an empty tree or can contain a single root. and every node in a binary tree can have almost (max) two children further more every child can also contain almost two child.

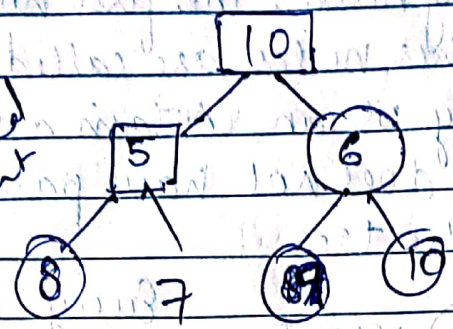


Node having common parent → siblings.
Node that do not have any children are called → leaf node.
leaf nodes are also called external nodes.
Node containing child as well as parent are known as internal nodes.

* any non leaf node except root are called internal node.

From data set to Binary tree.

10, 5, 6, 8, 7, 9, 10



Binary tree does not define any rule for making left or right node.

Subse pehle left child i. if there is a left child create right child.

Complex implementation of Binary Tree therefore we define Binary search tree

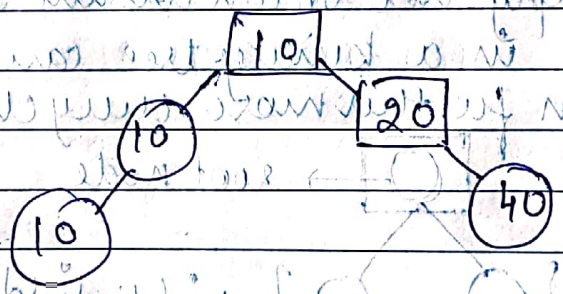
Binary search tree defines rule for right and left child

> node → right

< node → left

Binary search tree does not allow repitition if a repitition occur then it become the left child of equal node.

10, 20, 40, 10, 10, (Binary search tree).



ab jab to dundhoga upar se to jayega neeche

Binary search on sorted array with duplicate values.

Binary search done by binary tree making and as soon as first occurence of element is encountered we return and never go to other repeated occurences hence ambiguity occurs. Therefore no duplicates in binary tree.

Binary search tree is a type of Binary tree.

Date / /
Pg. No.
Notebook

10 → 5 → 6 → 8 insertand

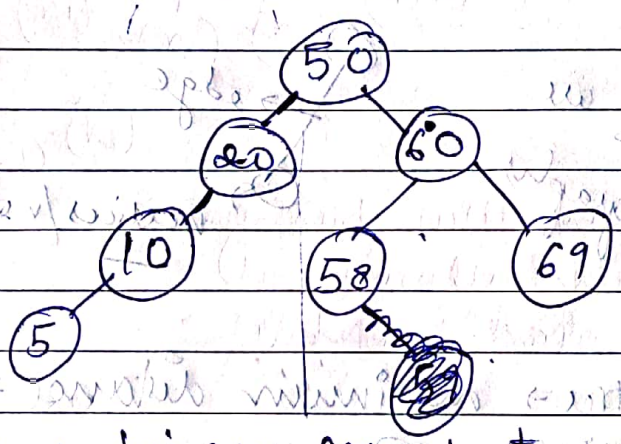
Binary search tree - (log n) search operation
Binary tree → m kuch bhi de sakta!

=

Implementation of Binary search Tree (BST)

concept :- first element to become the root
- if next value smaller left side jao.
- next value greater right side jao

50, 20, 60, 10, 5, 58, 69.



compare always with root.

operations: insert, look up

Create a binary search tree for 10 elements represented by a sorted array.

10 → 20 → 30

10, 20, 30, 40, 50

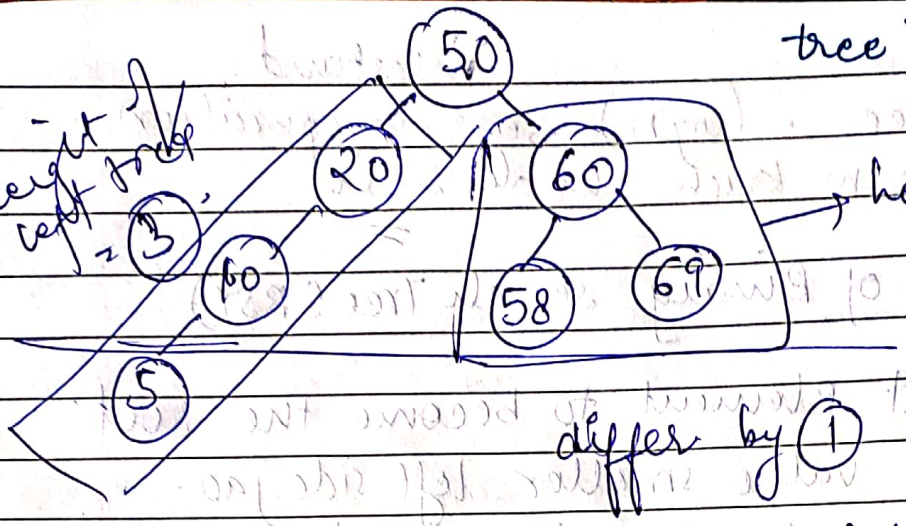
if tree growing only on right side - right skew tree.
if tree growing only on left side - left skew tree

unbalanced tree.

for an unbalanced tree you will not get (log n) complexity for insert and look up operation

Balanced tree.

A tree is said to be balanced if the height of left subtree and right subtree is only differ by 1



tree is a recursive data structure.

differ by (1) ∴ balanced tree.

Height of tree → is the total no of edges in between the root and the leaf node.

Non cyclic graphs are called trees and cyclic graphs are called graphs.

edge

vertices/nodes.

height of tree → maximum distance = 3

for left type comparision = (3)

right type comparisions = (3)

for worst case complexity comparision $\left(\frac{7}{2}+1\right)$

There are some self balancing binary search tree available that are

- AVL trees
- Red Black trees.

Binary search tree always sorted (depends upon traversal)

agar binary search tree h or balanced nhi h to order of (log n) nhi milega.

You can traverse a tree by using different traversal algorithm:-

i.) Breadth first traverse, } set
ii) depth first traverse }
iii) in order, preorder and post order.

Note:- Identify the match of 2 different sets of traversal algorithm in each other.

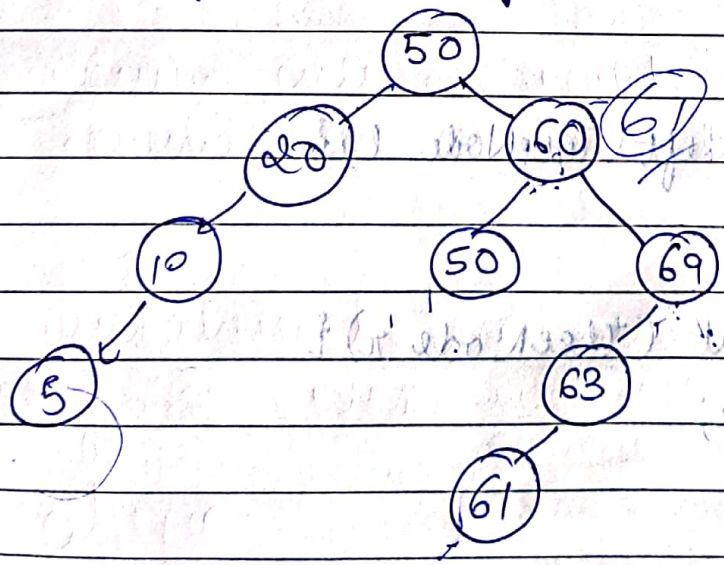
Operations of tree

traversal
insertion
. look up (search)
deletion of node from a binary search tree.

Case 1: Deleting a node with no child.
(parent se link hatado)

Case 2 Deleting a node with single child
(node to be deleted must be replaced with child).

Case 3 Deleting node with 2 child.
the deleted node will be replaced by the leftmost child of the right child.



To delete 60.
It will be replaced by 61

Implementing a binary Search Tree using a Linked List.

```
class TreeNode {
        private int data;
        private TreeNode left;
        private TreeNode right;

public TreeNode (int data){
  { data = data;
    left = null;
    right = null;
  }
public int getData (){
   return data;
  }
public TreeNode getLeft (){
   return left;
}
public TreeNode getRight (){
   return right;
}
public void setData(int d){
    data = d;
  }
public void setLeft(TreeNode l){
    left = l;
}
public void setRight (TreeNode r){
    right = r;
  }
}
```

can contain boolean isLeaf. & isInternal

Now collection of all nodes will form a tree.

```
class MyBinaryTree {
    TreeNode * root;          // default constructor to initialise it null.
public void Insert(int data);
{ TreeNode node = new TreeNode(data);
    if (root == null)
        root = node;
    else
    { TreeNode temp = root;          // dont disturb the root
// jiska child naya node hoga.  TreeNode parent = null;
        while (temp != null)
        { parent = Temp;
          if (node.getData() <= temp.getData())
          { temp = temp.getLeft();
          }
          else
          { temp = temp.getRight();
          }
        }
// at end of the loop we will know the parent of the incoming node
// now we have to decide whether it is left or right
        if (node.getData() <= parent.getData())
        { parent.setLeft(node);
        }
        else
        { parent.setRight(node);
        }
}
```

DryRun 50, 5, 20, 60, 80, 10.

[50 | null] node1

root = null ✓

root = [50 | null] ✓

[5 | null]

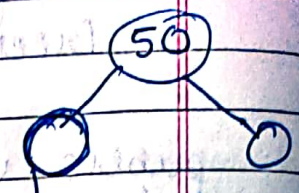
root = null X

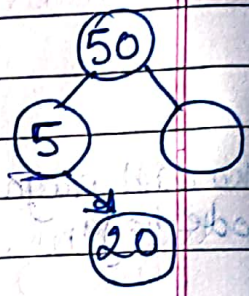
temp = root temp : 50.

parent = null

50

while (temp != null) ✓
{ parent = 50;
if (5 < 50) ✓
temp = null (hit)
temp = null ← } //while loop end.
temp = 50 ke left (null)
parent = 50.
if (5 < 50).
parent.setLeft(5)





Third data 20 | 20 | null |

root = null X
temp = 50 parent = null.
while (temp != null) ✓
{ parent = 20.
if (20 <= 50)
temp = 5
temp != null ✓
parent = 5
20 <= 5 X
temp.getRight (null)
temp = = null ✓
if (20 <= 5) false
parent.setRight(20)
5.setRight (20)

Depending on data tree can be balanced, unbalanced, right skew, left skew

At every position we either discard right, or discard left therefore discard 50% possibility
Therefore complexity = log n.

```
boolean search (int data)
{   boolean response = false;
    Temp TreeNode temp = root;
    while (temp != null)
    {   if (temp.getData() == data)
        {   response = true;
            break;
        }
```

```
else {
    if (data < temp.getData())
        temp = temp.getLeft();
    else
        temp = temp.getRight();
  }
}
return response;
}
```

complexity = log (n) at a time ya to left ya to right jaoge to isiliye adhi complexity.
balanced binary search tree.

Traversing a Binary Search Tree. ① Breadth first
② Depth First (recursive) 1.) preorder (recursive traversal).
→ process the root n. (print n)
→ traverse left subtree in preorder.
→ traverse the right subtree in preorder
Jaise hi last left child mil gaya phir uske right pe ja phir agar uska left hoga to phele wo phir uska hoga to uska bhi left
recursive Recursive technique
Not every call explores all nodes
complexity = 2^n (polynomial)

② inorder :- → increasing order m nodes access karege
① traverse the left subtree in order.
② visit the root node
③ traverse the right subtree in order.

③ postorder ① traverse left subtree in postorder
② traverse right
③ traverse the root

reverse inorder → returns the element in decreasing order.

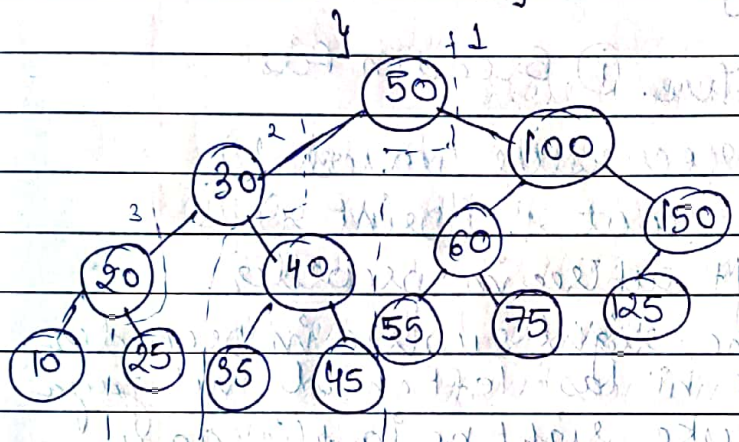
non general algorithm.

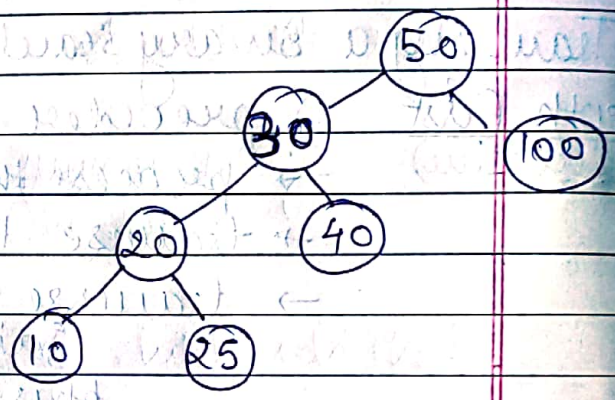
Code for Traversal

Root/Left/Right

root like rana.

```
void traversalPreOrder(TreeNode node)
{ if(node != null)
  { System.out.println(node.getData());
    traversePreOrder(node.getLeft());
    traversePreOrder(node.getRight());
  }
```





50, 30, 20, 10, 25, 40, 35, 45, 100, 60, 55, 75, 150, 125

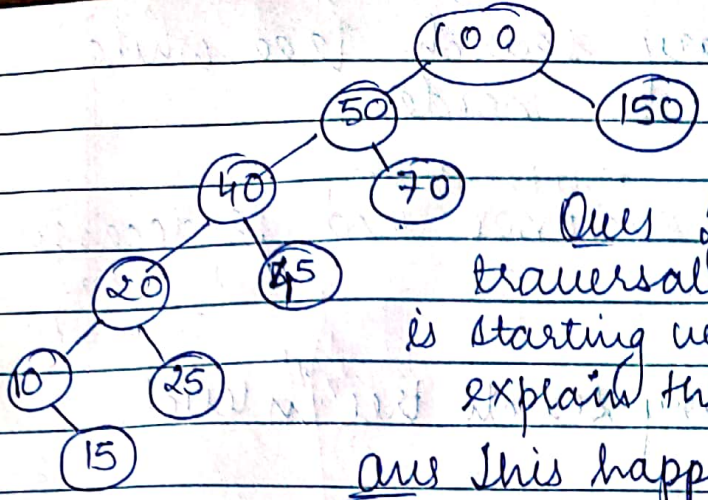
50, 30, 20, 10, 25, 40, 100

If root present at first position, then it is a preorder traversal.

If root present at last position, then it is a postorder traversal.

If root cannot be found but answer in ascending order (inorder traversal)

```
void traverseInOrder(TreeNode node)
{ if(node != null)
  {
    traverseInOrder(node.getLeft());
    sopln(node.getData());
    traverseInOrder(node.getRight());
  }
}
```



10, 15, 20, 25, 40, 45, 50, 70, 100, 150.

Ques If the tree inorder tree traversal is giving some output which is starting with the root value explain the structure of tree.

Ans This happens when tree is right skewed and there exist no left subtree to the ~~right~~ root
→ shows an unbalanced tree.

```
public void traversePostOrder(TreeNode node)
{   if (node != null)
    {   traversePostOrder(node.getLeft());
        traversePostOrder(node.getRight());
        sopln(node.getData());
    }
}
```

Why use recursion in traversal of tree?
→ tree is a recursive data structure as usual. Traversal can also be done using stack. insertion can also done using recursion.

Deleting a node from Binary Search Tree.
↓ complexity log(n).

```
public TreeNode delete(int data) {
    // search for data
    TreeNode response = null;
    TreeNode temp = root;
    TreeNode parent = null;
    while (temp != null && temp.getData() != data)
    {   parent = temp;
```

WAP to create a binary search tree while traversing a existing BST in inorder.

WAP to find inorder successor and predecessor (right ka leftmost, left ka rightmost) for any given node.

Q. WAP to search a binary search tree in level order.

BreadthFirstTraverse

if (tree is ~~root~~ empty)
then print cannot traverse
else
create a queue and add the root of the tree into the queue.

2) Traverse the queue until the queue is empty.

I Remove a element from the queue

II Process that element.

III If the element contain a left child, add that child into the queue

IV If the element contain a right child, add that child into the queue.

```
void traverseLevelOrder()
{ if (root == null)
    { Sopln ("cannot be traversed");
    }
  else
    { Queue<TreeNode> queue = new
      LinkedList<>();
      queue.add(root);
      while (! queue.isEmpty())
      {
```

LinkedList class is concrete implementation of Queue

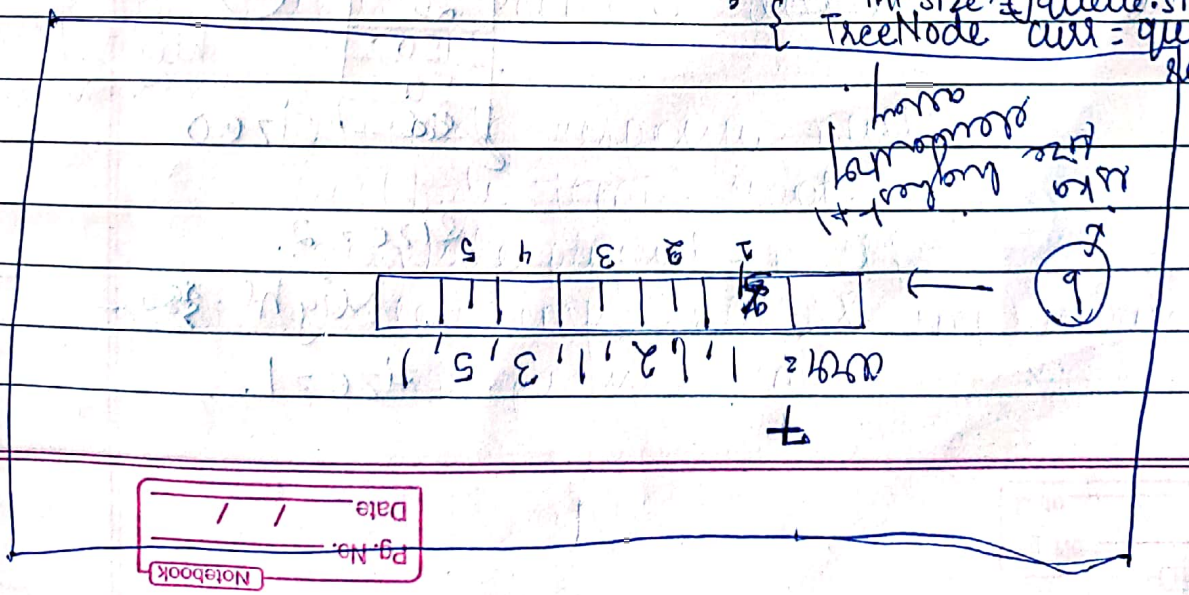
Ques Create a array by traversing a binary search tree.

Height of Tree

```
int height (TreeNode node)
{ if (node == null)
  { return -1;
  }
  else {
    return 1 + Math.max
    ( height (node.getLeft()),
      height (node.getRight()));
  }
}
```

Without Recursion

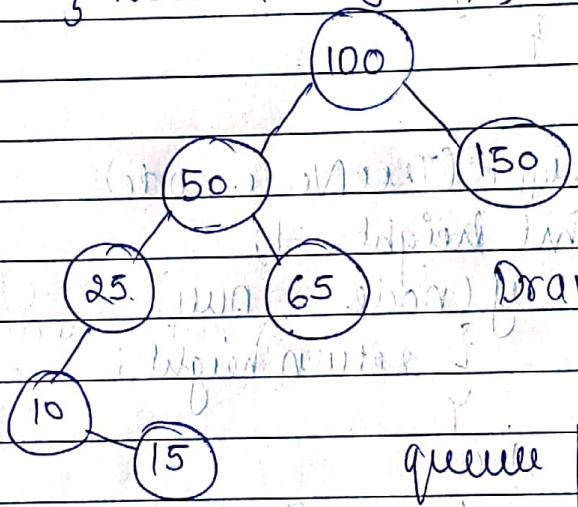
```
int height (TreeNode node)
{  int height = -1;
   if (node == null)
     { return height;
     }
   else
   { Queue <TreeNode> queue =
  new LinkedList <> ();
  queue.add (node);
     while ( !queue.isEmpty())
     { int size = queue.size();
     { TreeNode curr = queue.
                       remove ();
```



```
height++;
int size = queue.size();
while (size > 0)
{ if (curr.getLeft() != null)
  { queue.add(curr.getLeft());
  }
if (curr.getRight() != null)
  { queue.add(curr.getRight());
  }
size--; curr = curr.get
} return height; }
```

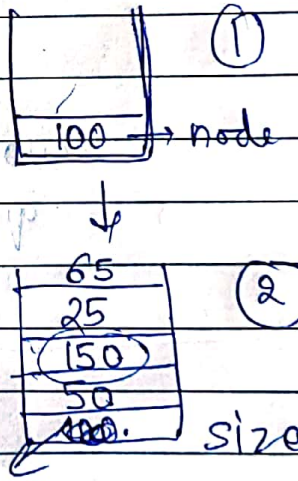
Input
1 2
1 2 3 4 5 6 7 8

Output
1
2
2
2
2
2
2
2



Draw all states of queue

queue



① ② height = 1

size = 2.

height = 2

size = 1.

Notebook
Date / /
Pg. No.

```
public int height (TreeNode node)
    { int height = -1;
      if (node == null)
          { return height;
          }
      else
          { Queue<TreeNode> queue = new LinkedList();
            queue.add(node);
            while (!queue.isEmpty ())
            {   int size = queue.size();
                height++;
                while (size>0) {
                TreeNode currentNode = queue.remove();
                if (currentNode.getLeft()!=null)
                  { queue.add (currentNode.
                                getLeft());
                  }
                if (currentNode.getRight()!=null)
                  { queue.add (currentNode.
                                getRight());
                  }
                size--;
            }
          }
          return height;
    }
```

AVL Trees :- Adelson, Velskii, Landi, you cannot guarantee order of log n in BST.

in BST, search ki complexity O(log n) → not guaranteed for eg in right skewed

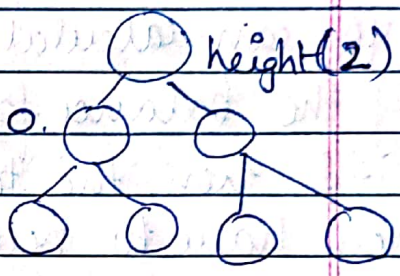
So form of

AVL → Balanced binary search tree

→ height balanced tree (left subtree = right subtree height)

height should be correspondent to log n

$h = \log_2 n$



null node = 0.

No of nodes $= 2^h - 1$

$= 2^3 - 1 = 7$

height = $\log_2$ (n) → no of nodes

$= \log_2(7) = 2.8$

For AVL you will calculate balance factor. This balance factor can have a value of {-1, 0, 1}

so after every insert (log n) check Balance. } complexity should be minimum.

then reconstruct a tree

$\log n$

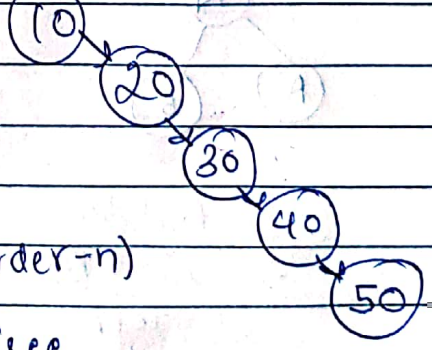
Create a balanced binary tree from given set of values.

10, 20, 30, 40, 50

construct a tree.

↓

unbalanced (right skewed).

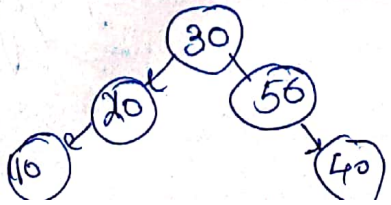


for balancing a tree.

traverse in inorder (order-n)

from midpoint create a new tree

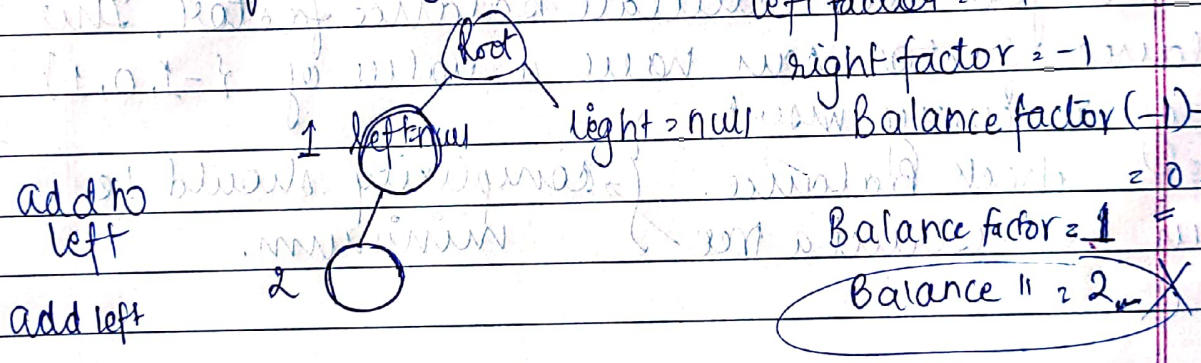
↓



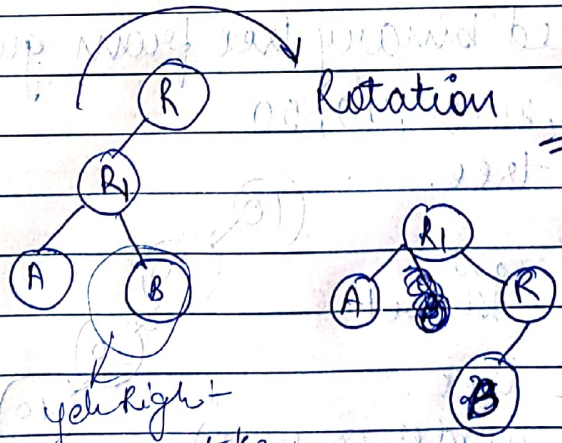
Complexity in reconstructing
= order(n) + order(n)
↓ ↓
construction traversal
= O(n).

How to calculate balance factor?
ans balance factor = height of left subtree − height of right subtree

You can calculate balance factor for every node.
If the balance factor value is greater than 1 or less than −1 then, the tree is unbalanced.
You have to reconstruct that particular tree in order to achieve balance.
For this, after every insert or delete, we will calculate balance factor of the tree



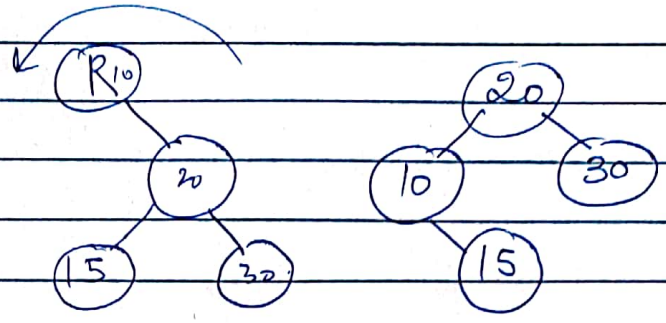
balance it



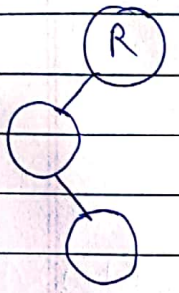
Notebook
Pg. No.
Date / /

Left Rotation

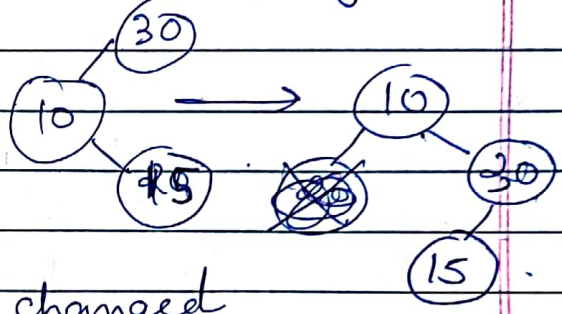
Right side pe unbalanced h to left rotation



Right and Left Rotation are known as single Rotations.



unbalanced on left side therefore right rotation



so now it has changed from unbalanced left to unbalanced right

so now we have to used double rotation