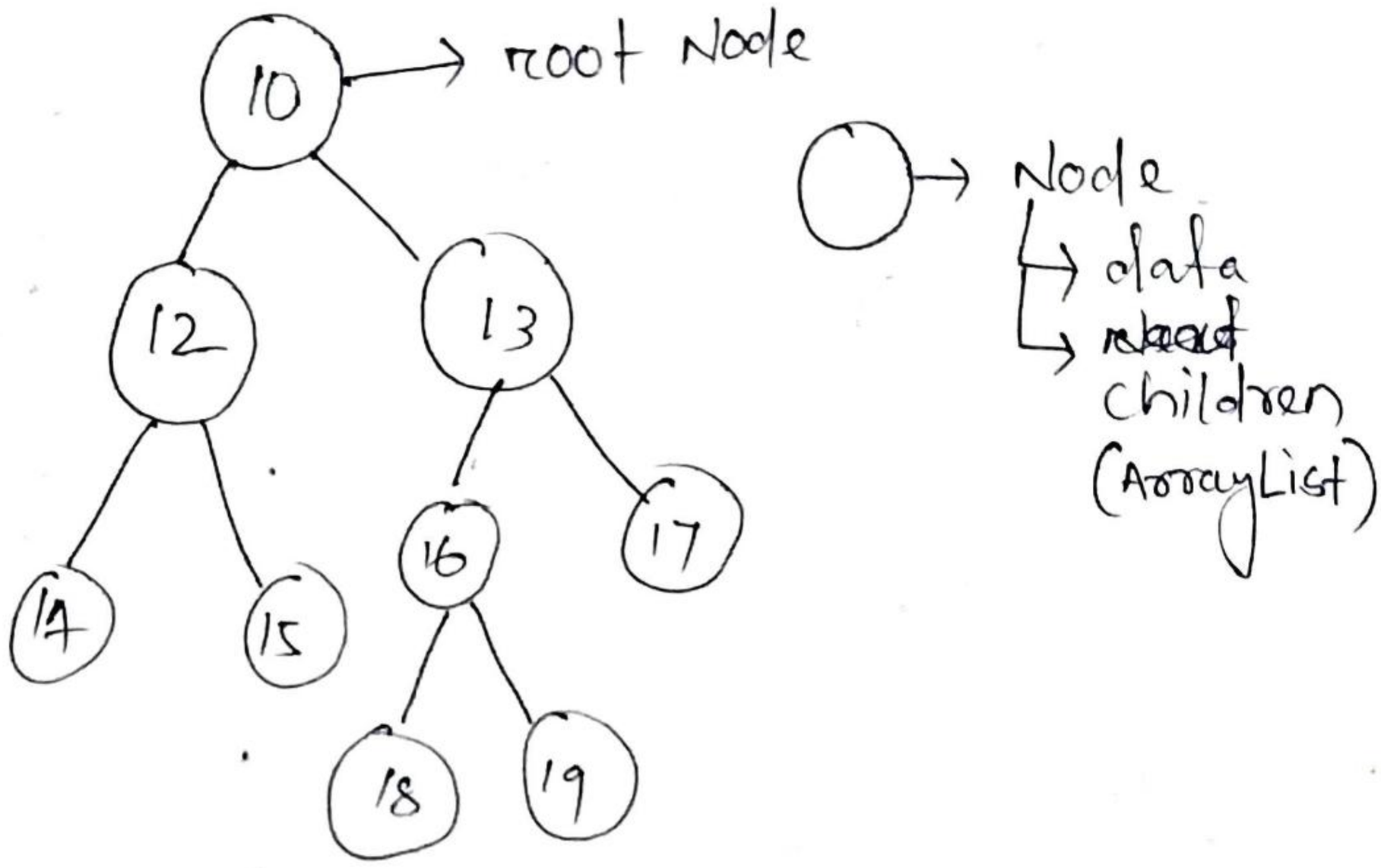
# Tree

## Generic Tree

### structure of Tree




### implementation of Generic Tree

① no. of nodes ⇒

② root

③ Ancestors

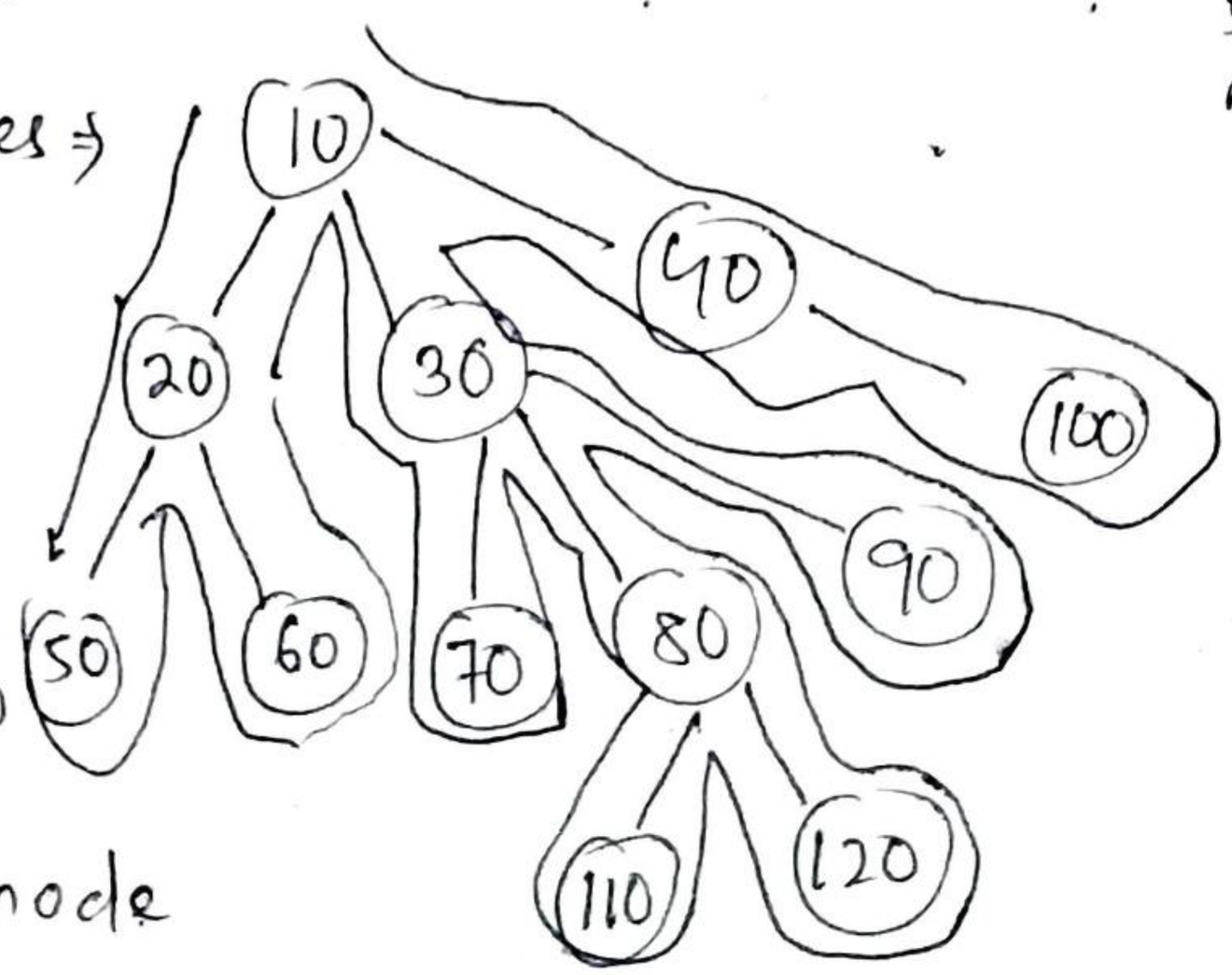
80 ⇒ 30, 10

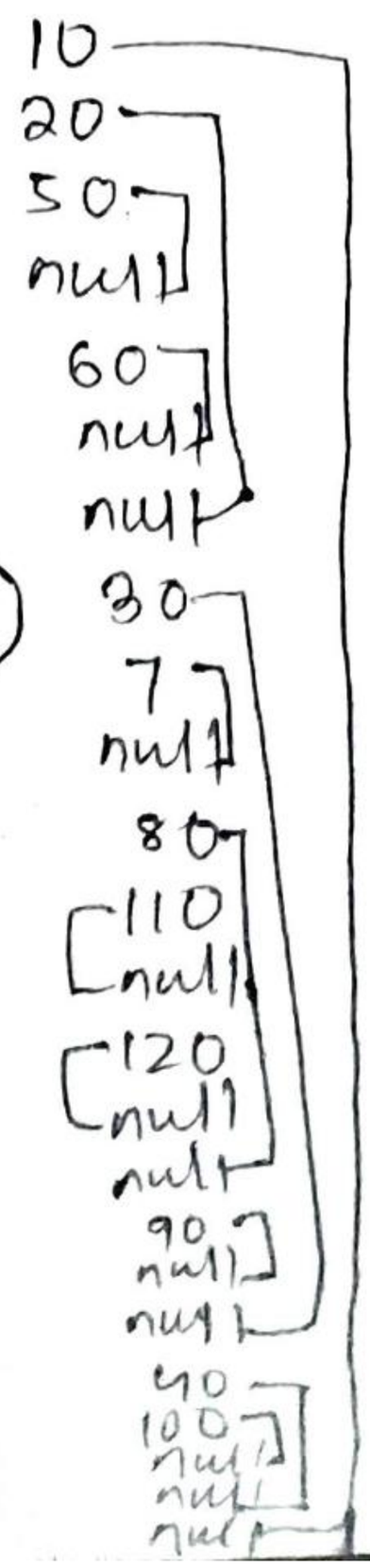
50 ⇒ 20, 10

④ parent node

↳ parent(10) = null

parent(110) = 80



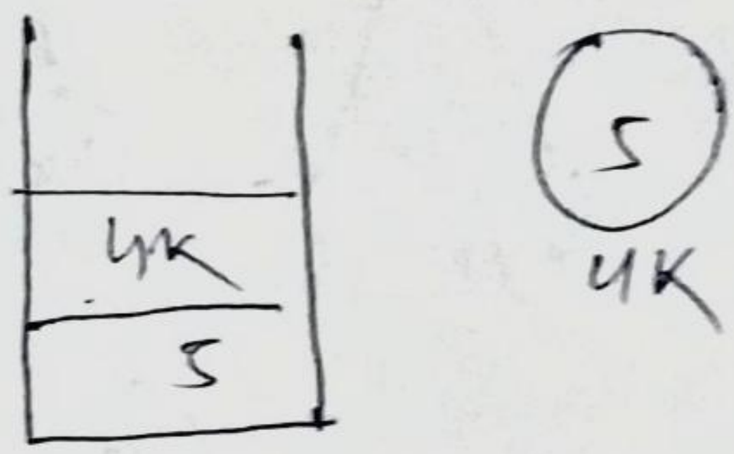

Scanned by TapScanner

To store this we have to create a array of Integer type (not int type)

→ because we want to store null, so in int type we can't store it, but in Integer type we can store null

→ In int the ~~mem~~ data is stored in stack memory but in Integer the data is stored in head memory

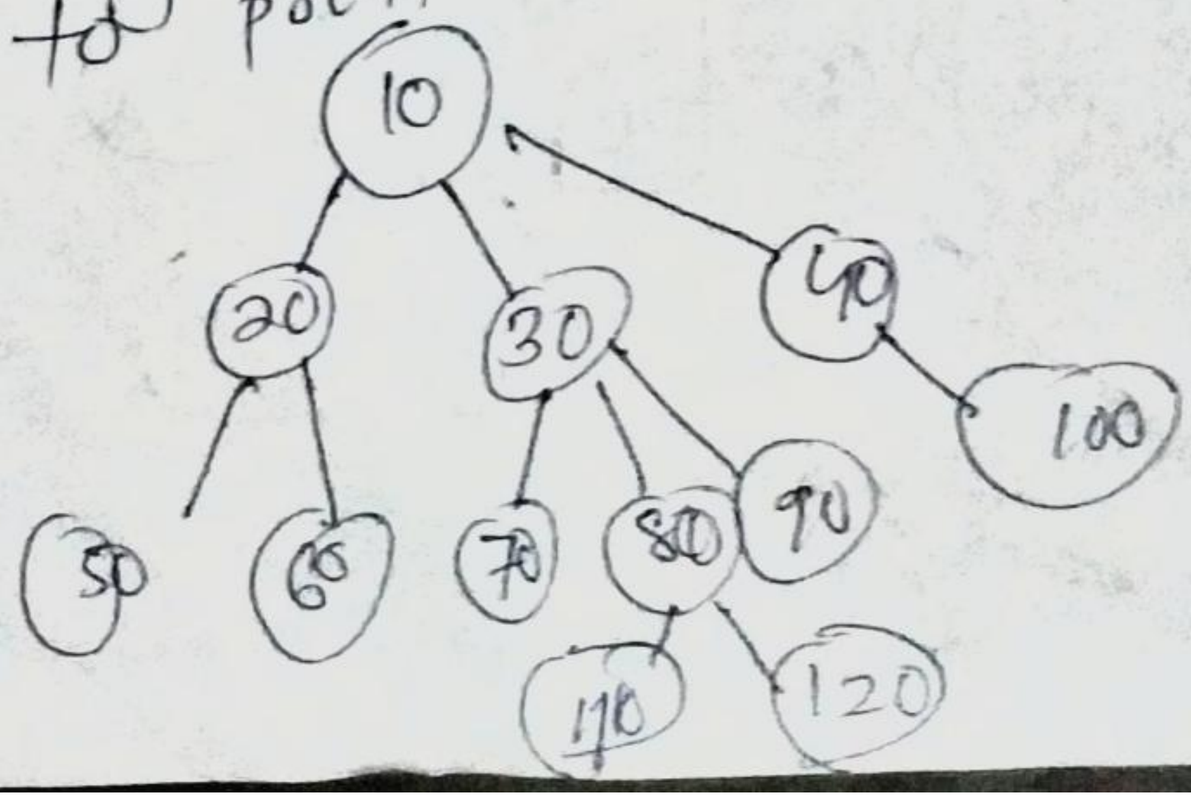
int s
└ Integer s




→ To take input from generic tree ⇄

Integer Input[] = {10, 20, 50, null, 50, null, null, 30, 70, null, 80, 110, null, 120, null, null, 90, null, null, 40, 100, null, null, null};

Q-1 :-
we are given a tree structure we have to print it on console




Scanned by TapScanner

implementation of a Tree

```
public class implementation {
public static class Node {
    int data;
// ArrayList<Node> children;
 Node (int data)
  {
 ArrayList<Node> children = new ArrayList<>();
    this. data = data
    }
 }
public static Node Construct (Integer input[])
 {
 Node root = new Node(input[0]);
 Stack <Node> st = new stack<>();
  st. push (root);
 for (int idx = 1; idx < input length; idx++)
   {
   Integer val = input [idx];
   if (val == null)
    {
    st. pop();
    }
```

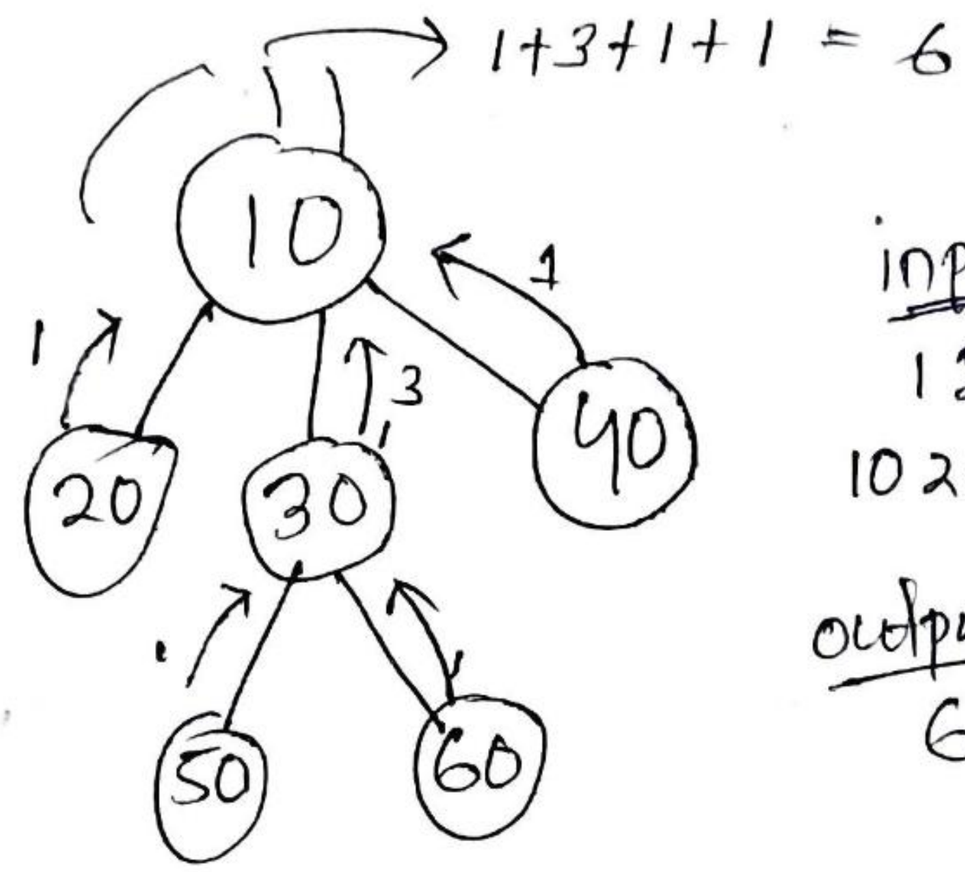
Scanned by TapScanner

```
else
 {
 Node node = new Node(val);
   st.peek().children.add(node);
    st.push(node);
    }
  }
 return root;
 }
public static void display(Node node)
   {System.out.print(node.data + "->");
   for(Node child : node.children)
    {System.out.print(child.data+ " ");
    }
   System.out.println(".");
   for(Node child : node.children)
    {display(child);
    }
  }
public static void main(String[] args)
 {
Integer input = {10, 20, 50, null, 60, null, null
30, 70, null, 80, 110, null, 120, null, null, 90
```

Scanned by TapScanner

```
null, null , 40, 100, null, null, null);
Node root = Construct (input);
  display(root);
}
}
```

## Size of a generic Tree




**input**

12
10 20 -1 30 50 -1 60 -1 -1
40 -1 -1

**output**

6

**Steps**

→ create a result variable and intialize it to 0.

→ If ~~there~~ is ~~no~~ children for a node ~~then return~~

→ Call the children for a child and store it in result

```
for (Node child : node.children
  {
    res = rest size (child);
  }
```

Scanned by TapScanner

* Intialy size is 0 then Call till last point of child where there is no children for a child then return size +1{(0+1) = 1}

* When Comming from a children node if there is another children exists then add the 1st result 1 & the size of 2nd children 1 = 2

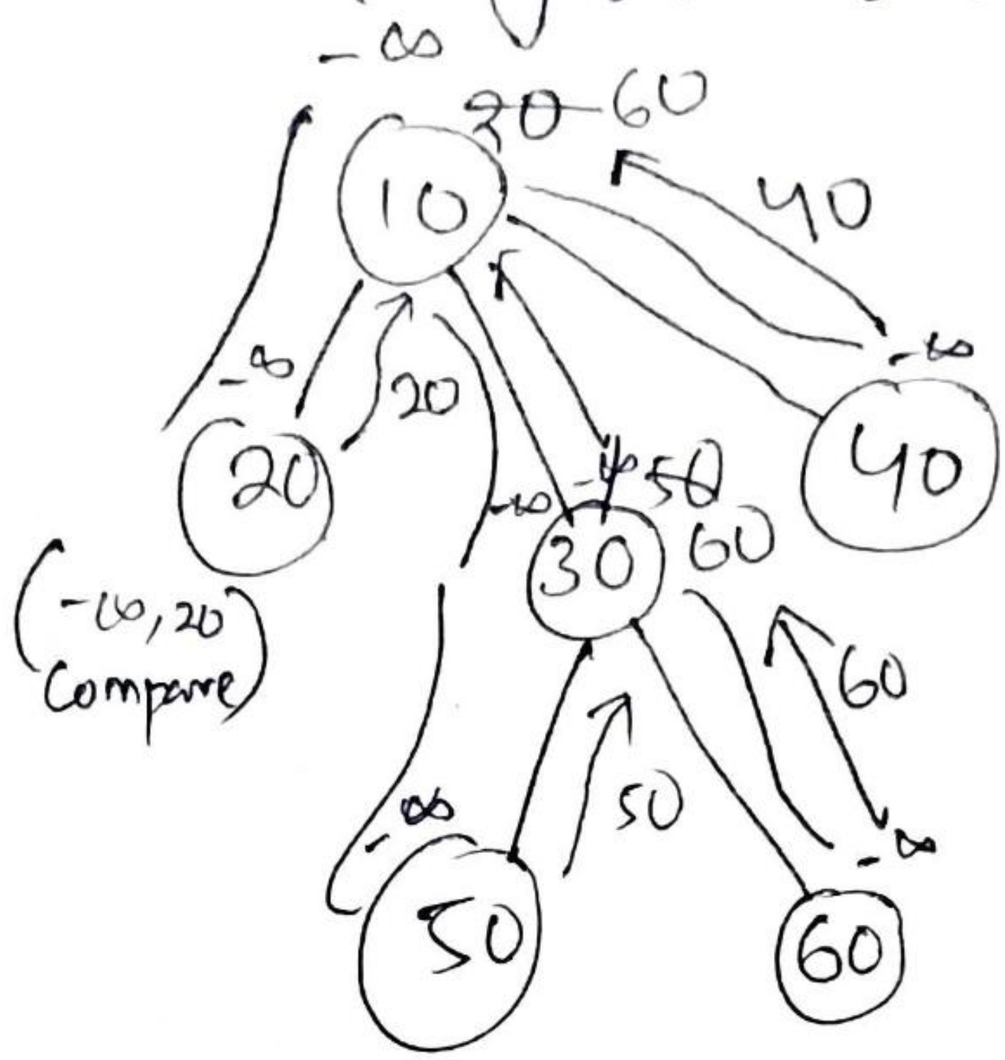
* Lastly return the res.

<u>Code</u>

```
public static int size (Node node)
{
int res = 0;
for (Node child : node. children)
{
res = res + size (child) ;
}
res = res + 1;
return res;
}
```

Scanned by TapScanner

# Maximum in a generic Tree




input

12

1020-13050-160-1-1

40-1-1

output

60

## Steps

* Consider a maximum value = -∞

int OMax = Integer.MIN_VALUE

* Call all the child and Compare the node data and OMax. and update the overall Maximum OMax.
* return the maximum betn node data and overall Maxim.

## Code

```
public static int max (Node node)
{
int oMax = Integer.MIN_VALUE;
for (Node child : node.children)
{
oMax = Math.max(oMax, max(child));
}
```

Scanned by TapScanner

return Math.max (oMax, node.data);

}

## Height of a generic Tree

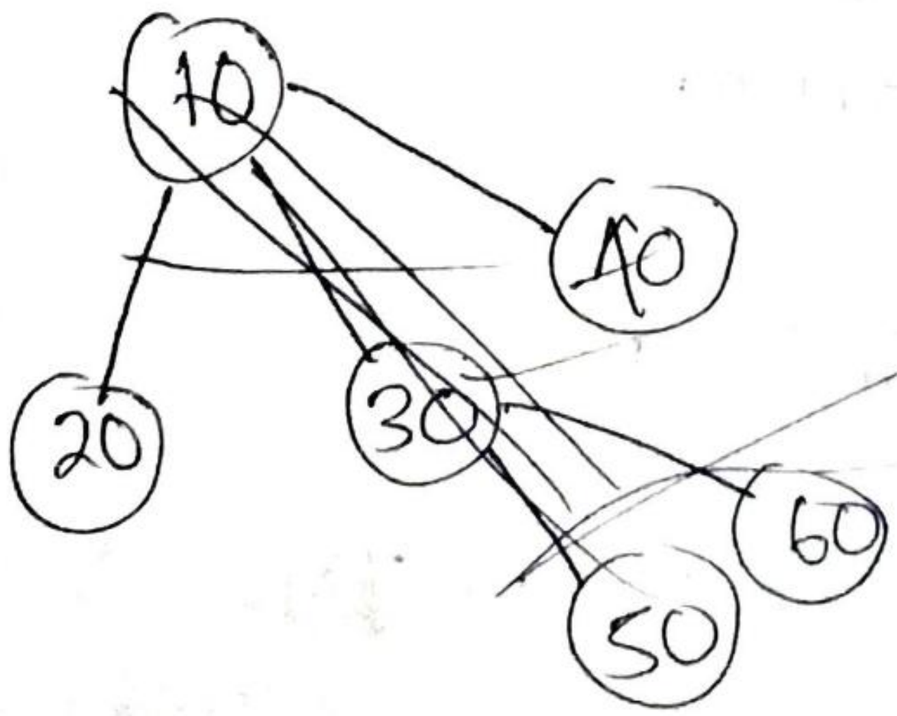
Given input of a generic tree we have to calculate the height of a generic-tree

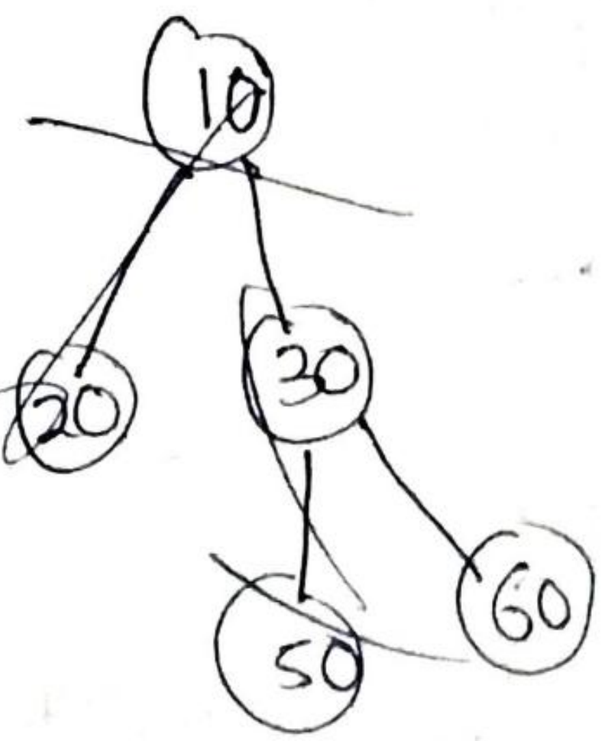
→ height can be calculated through edges & also through nodes

→ Here we are given to calculate through edges.

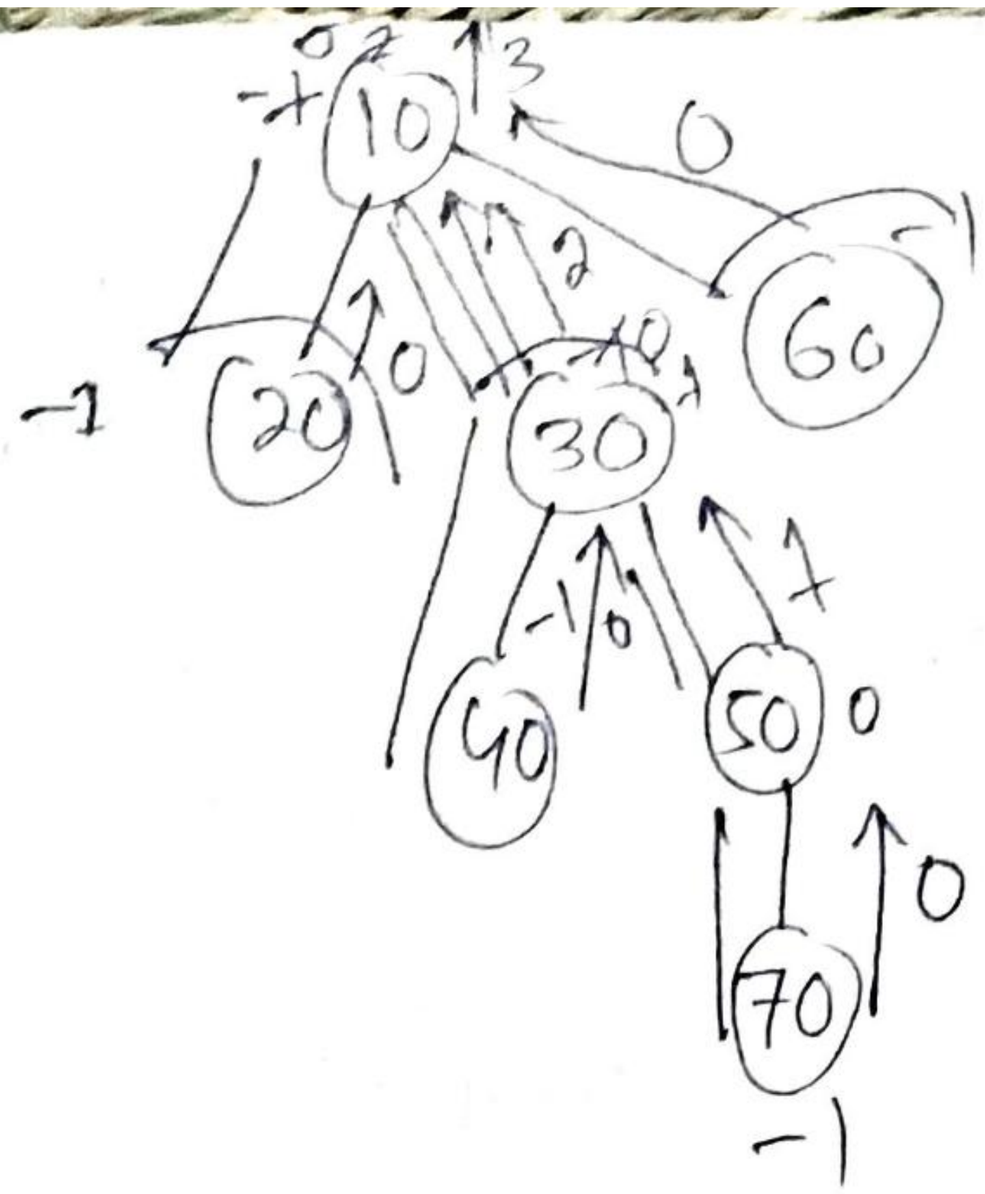
input 12

10 20 -1 30 50 -1 60 -1 -1 40 -1 -1




Scanned by TapScanner




input

12

10 20 -1 30 50 -1 60 -1 -1 40 -1 -1

output

2

steps

* set default height as -1.
* Call the children node and find the height of childs
  for (Node child : node.children)
* Now make res variable and store height of child.
* Now check res if it is greater than height then update height as res
  ht = res
* Now as we back from all children node now only root node left total height is equal to (ht + 1)

Scanned by TapScanner

## Code

```
public static int height (Node node)
{
    int ht = -1
  for (Node child : node.children)
    {
      int res = height (child);
      if (res > ht)
      {
        ht = res;
      }
    }
  return ht + 1;
}
```

## level order of a generic Tree

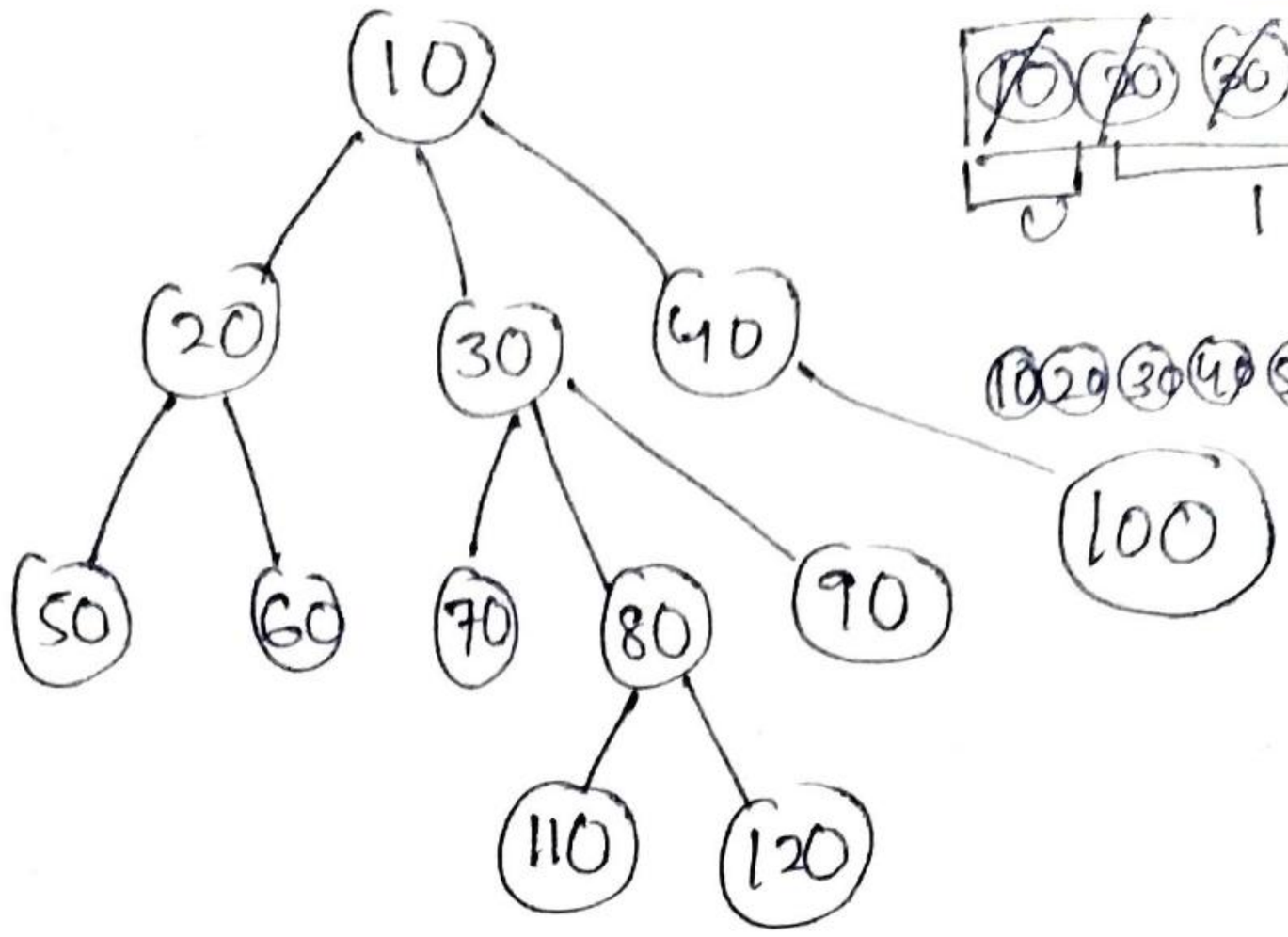
**input**

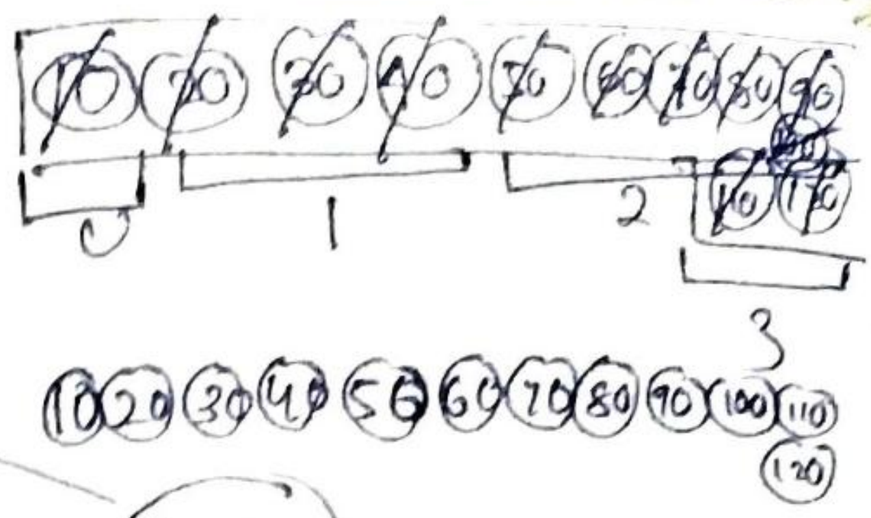
10 20 50 -1 60 -1 -1 30 70 -1 80 110 -1
120 -1 90 -1 40 100 -1 -1 -1

**output**

10 20 30 40 50 60 70 80 90 100 110 120

Scanned by TapScanner




Here we are given a tree and we have to print its node level by level means from root to every child and each children in a single line

10 20 30 40 50 60 70 80 90 100 110 120

## Steps

* We will solve it by the help of a Queue.
* First create a queue and put root node into the Queue first.
* Now again remove that node and print it's child, continue this process till the Queue becomes empty
* remove the node and print its child value then add the childs int queue.

Scanned by TapScanner

Code

```
public static void levelorder (Node node)
{
Queue <Node> queue = new ArrayDeque<> ();
queue.add (node);
while(queue.size() > 0)
 {
Node tmp = queue.remove ();
system.out.print (tmp.data + " ");
for(Node child : tmp.children)
  {
   queue.add (child);
  }
 }
 system.out.println (" . ");
}
```

## Generic Tree Traversal

input

10 20 -1 30 50 -1 60 -1 -1 40 -1 -1

O/P :- Node pre 10

Edge pre 10 -20

Node pre 20

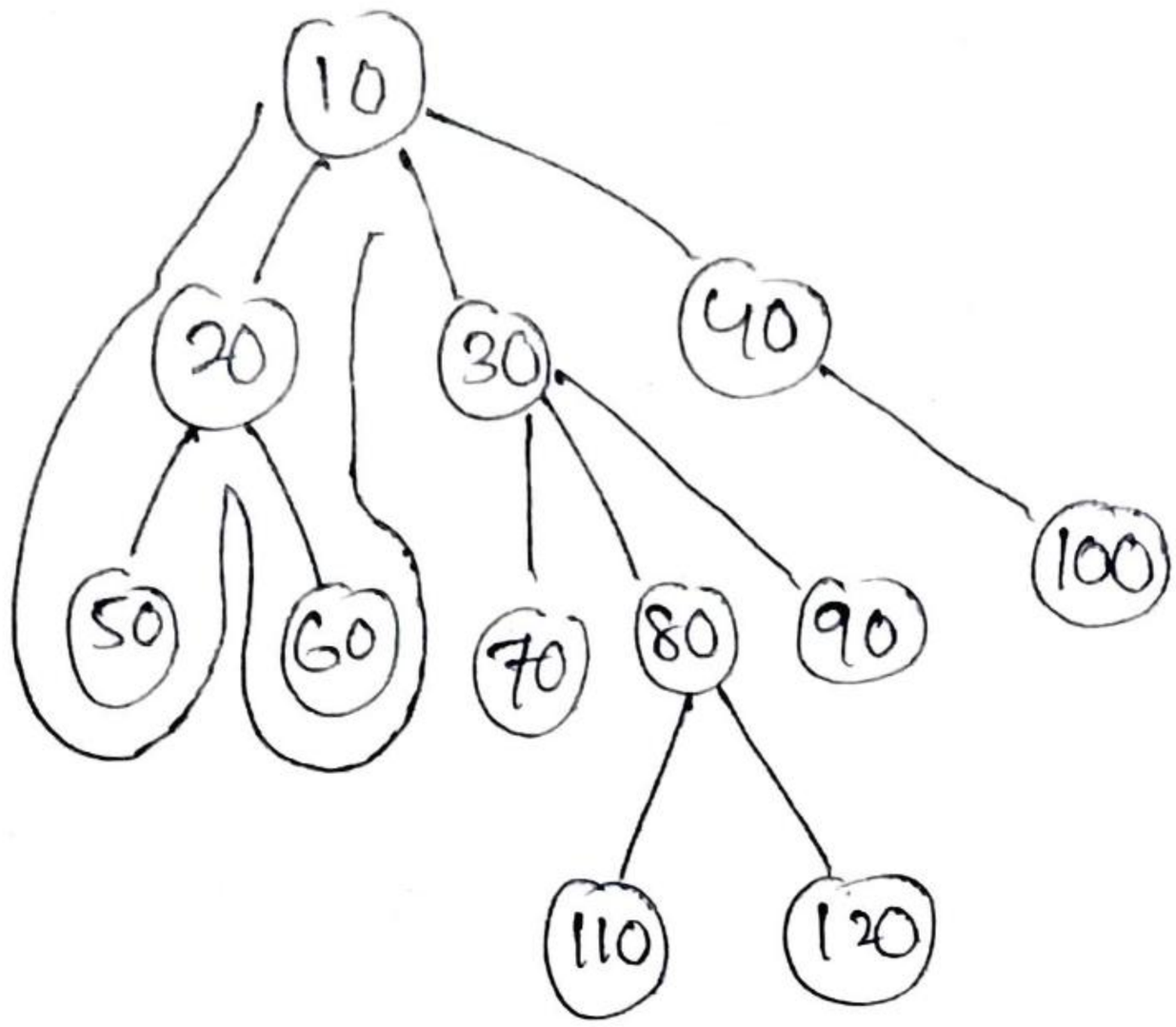
Node post 20

Edge post 10 -20

Edge pre 10-30

Node pre 30

Scanned by TapScanner




Node pre I 0
Edge pre 10-20
Node pre 20
Edge pre 20-50
Node pre 50
Node post 50
Edge post 20-50
Edge pre 20-60
Node pre 60
Node post 60

pre = when we are visiting 1st time
post = when we are visiting 2nd time

When we are visiting root node first it is called Node Pre and also called when we call the child then it is call Edge pre

* When we are leaving a Node called Node Post and in child it is called Edge post.

<u>steps</u> :-
→ Print data of nodes. Called Node pre
→ Make a for loop for every child and before calling the function Traversal for every child print the data of node is called Edge pre, when

* After the function called the print the data of node called Edge post

* After comming out for for loop that is when leaving the node print the data of node called ~~Edge post~~ Node post

Code

```
public static void Traversal (Node node)
  {
   System.out.println ("Node pre" + node.data);
   for (Node child : node.children)
   {
    System.out.println ("Edge pre" + node.data + "--",
                                        child.data);
    Traversal (child);
    System.out.println ("Edge Post" + node.data + "--"
                                        + child.data);
   }
   system.out.print en ("Node post" + node.data);
  }
```

Scanned by TapScanner

# Level order of a generic Tree Linewise

**input**

24

10 20 50 -1 60 -1 30 70 -1 80 110 -1 120 -1 -1

90 -1 -1 40 100 -1 -1 -1

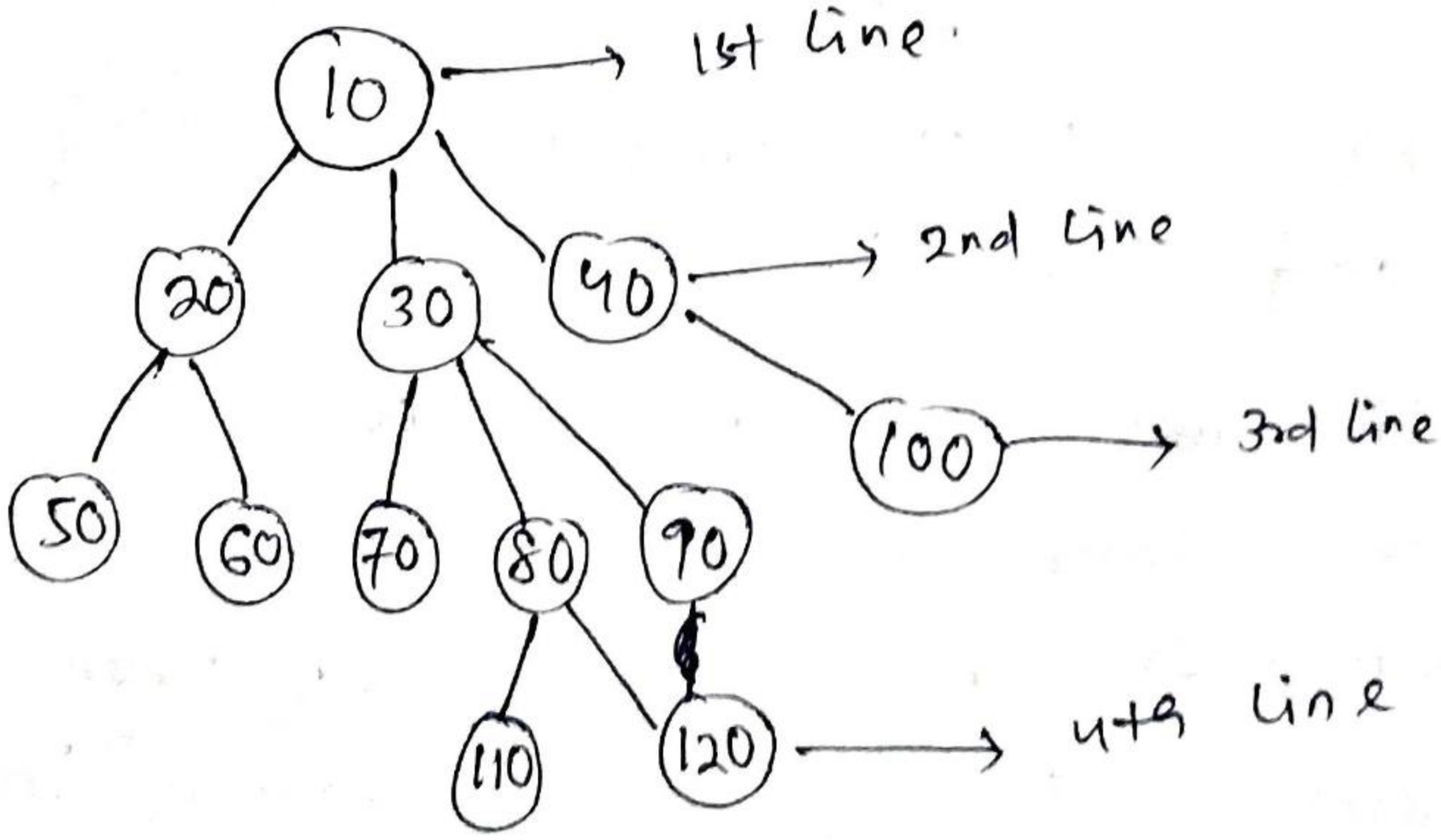
**output**

10

20 30 40

50 60 70 80 90 100

110 120




Here we have to print level order of tree line wise means (line by line)

10

20 30 40

50 60 70 80 90 100

110 120

Scanned by TapScanner

## Steps

* create two Queue one is main Q and other is helper Q.
* Now push the root node into mainQ (add)

  mainQ.add(root)
* While size of mainQ is greater than zero means element present then remove that element from main Q and print its data

```
while (mainQ.size() > 0)
{
    mainQ.remove();
}
```

* Now call the childrens of mainQ and add them to helper Q
* If size of mainQ become zero then swap the two queues means change references of two queues. (change main Q to helper Q & helper Q to main Q), move to next line

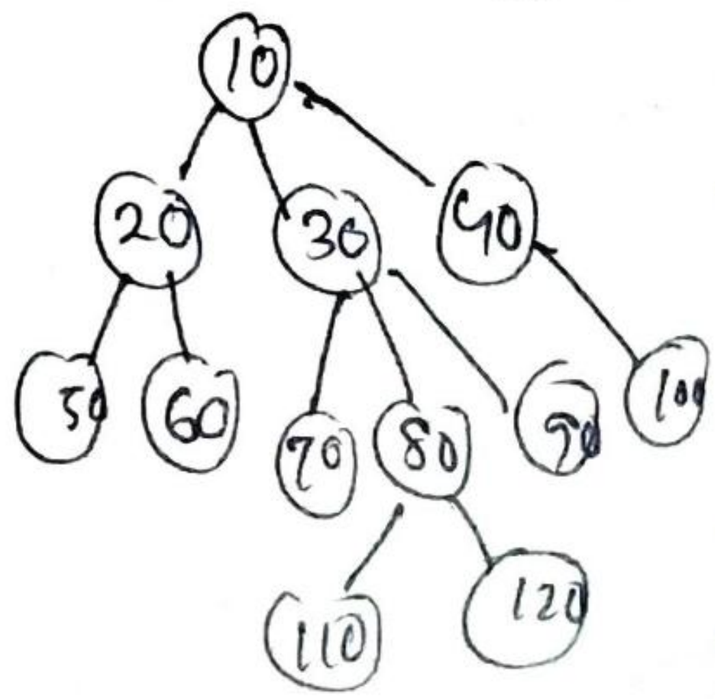
mainQ = 4k, helperQ = 5k — initially

| | | | | | | |
|---|---|---|---|---|---|---|
| 10 | 50 | 60 | 70 | 80 | 90 | 100 |

4k 5k 4k

| | | |
|---|---|---|
| 20 | 30 | 40 |

5k 4k 5k




Scanned by TapScanner

Code

```
public static void LevelOrderLinewise(Node node)
 {
 Queue<Node> mainQ = new ArrayDeque<>();
 Queue<Node> helperQ = new ArrayDeque<>();
  mainQ.add(node);
  while(mainQ.size() > 0)
   {
Node frontNode = mainQ.remove();
  System.out.print(frontNode.data + " ")
  For(Node child : frontNode.children)
   {
    helperQ.add(child)
   }
  if(mainQ.size() == 0)
   {
    System.out.println();
    Queue<Node> tempQ = mainQ
    mainQ = helperQ;
    helperQ = tempQ;
   }
  }
 }
```

Scanned by TapScanner

# Level order Linewise Zigzag

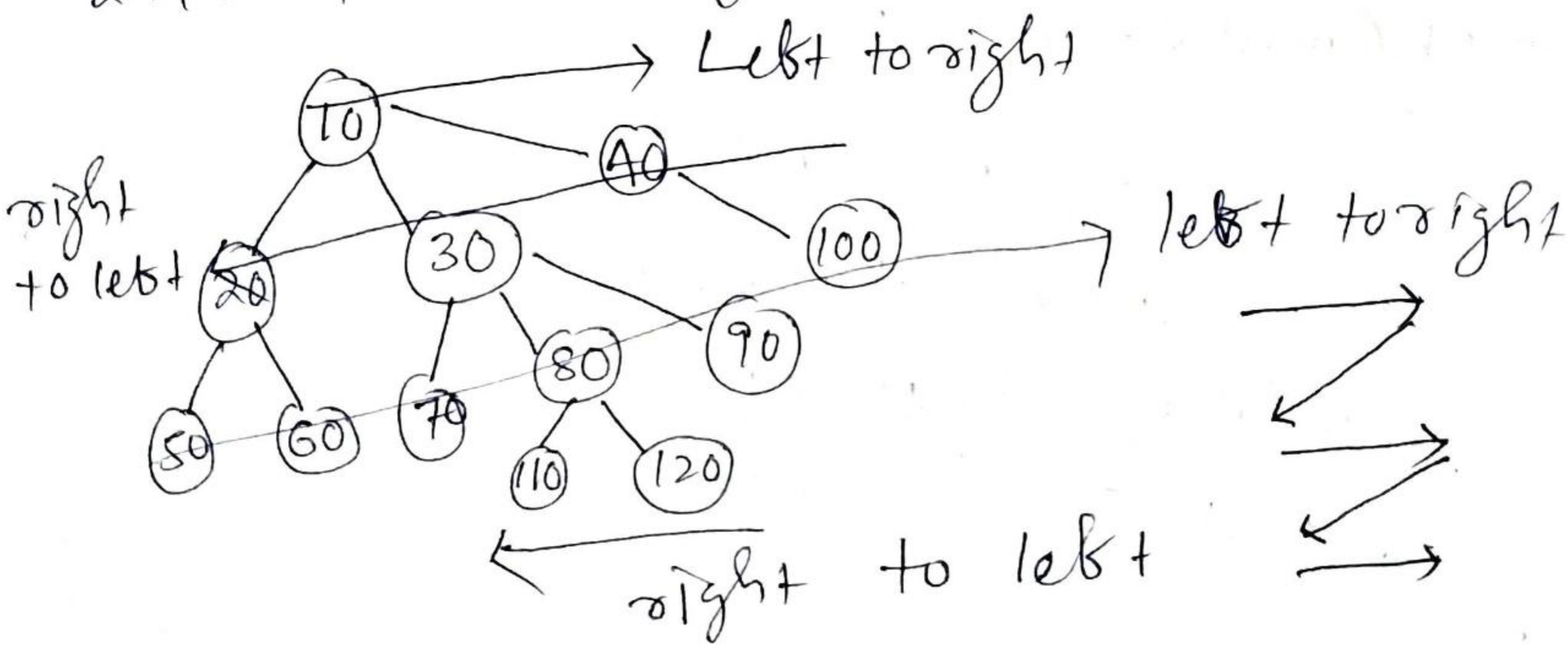
**input**

24
10 20 50 -1 60 -1 -1 30 70 -1 80 110 -1 120 -1 -1
90 -1 -1 40 100
-1 -1 -1

**output**

10
40 30 20
50 60 70 80 90 100
120 110

Here we have to travel all the level of a generic tree in zigzag manner means 1st level should from left to right and 2nd level from right to left so on




Here for levels 0 & 2 → Left → right (even)

for levels 1 & 3 → right → left (odd)

Scanned by TapScanner

## Steps

* Create two stack main stack and helper stack (mains, helpers) and maintain intial level as 0 (zero).

* Now push intial given root node to main stack. (mains.push(root))

* Now when size of mainstack is greater than or not zero then pop the element and print its data.

```
Node topNode = mains.pop();
Sys0(topNode.data+" ");
```

* We have to push the element in left to right manner for even level, so take modulus of level by 2 and if it is zero means even then add elements from left to right

```
for (int idx = 0 ; idx < topNode.children.size();
                                        idx++)
    {
      helpers.push(topNode.children.get(idx));
    }
```

* else if odd level then push from right to left

```
else {
    for (int idx = topNode.children.size()-1; idx
    {          idx >= 0; idx--)
  } helpers.push(topNode.children.get(idx));
```

Scanned by TapScanner

* When main stack become empty (or zero) then
→ increase the levels,
→ move to next line,
→ change the references of main stack & helper stack

Code

```
public static void levelOrderLinewiseZigZag
                                (Node node)
{
stack<Node> mains = new stack<>();
stack<Node> helpers = new stack<>();
int level = 0;
mains.push(node);
while (mains.size() > 0)
{
Node topNode = mains.pop();
System.out.print(topNode.data + " ");
if (level % 2 == 0)
{
for (int i = 0; i < topNode.children.size(); i++)
{
helpers.push(topNode.children.get(i));
}
else {
for (int i = topNode.children.size()-1; i >= 0;
                                        i--)
{
helpers.push(topNode.children.get(i));
}
```

Scanned by TapScanner

```
if (mains.size() ==0)
 {
  level++;
  System.out.printen();
 steek<Node> temps = mains;
   mains = helpers;
   helpers = temps;
 }
}
}
```

## Find in a generic Tree

**input**

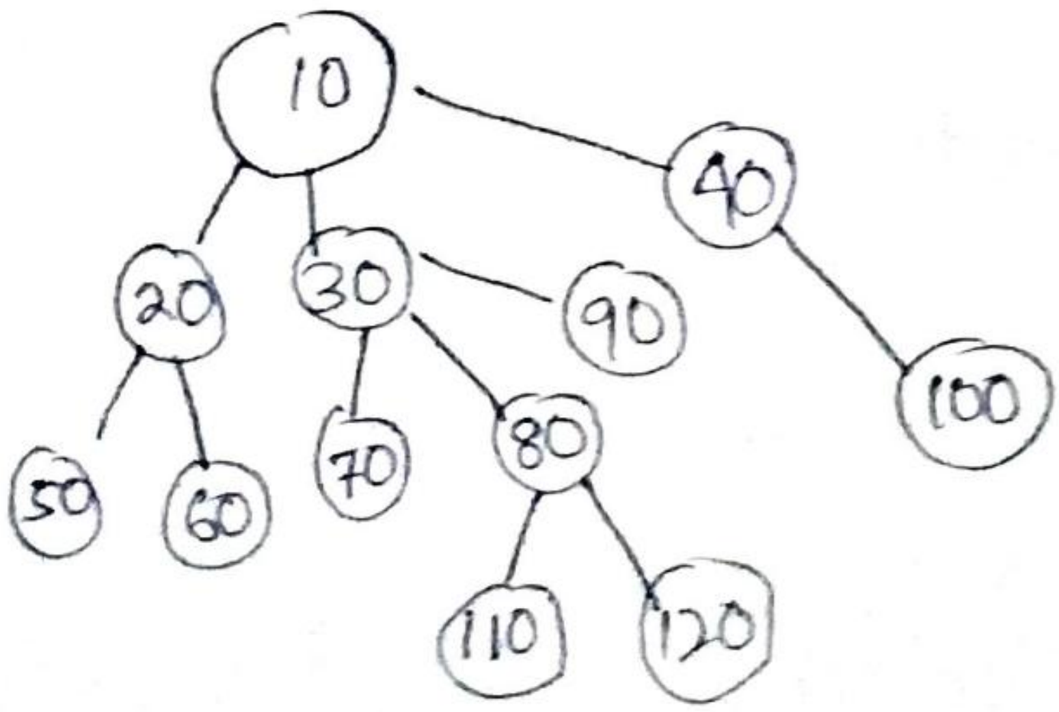
24

10 20 50 -1 60 -1 -1 30 70 -1 80 110 -1 120 -1 -1 90 -1 -1 40 100 -1 -1

120

**output**

true




We are given to find 120 exists in the ~~ct~~ tree or not.

Scanned by TapScanner

Call every children of tree if there we get the element then return true.

Steps

* check if the given node is our required element or not if it is the required element then return true
* Otherwise Call the child and find the element in their children node. If element will be get then return true
* If node is not the element or we didn't find in the child node then return false.

Code

```
public static boolean find (Node node, int ele)
  {
  if (node.data == ele)
   {
    return true;
   }
  for(Node child : node.children)
   {
    if (find(child, ele))
     {
      return true;
     }
   }
  return false;
}
```

Scanned by TapScanner

# Node to root path in a generic Tree

**input**

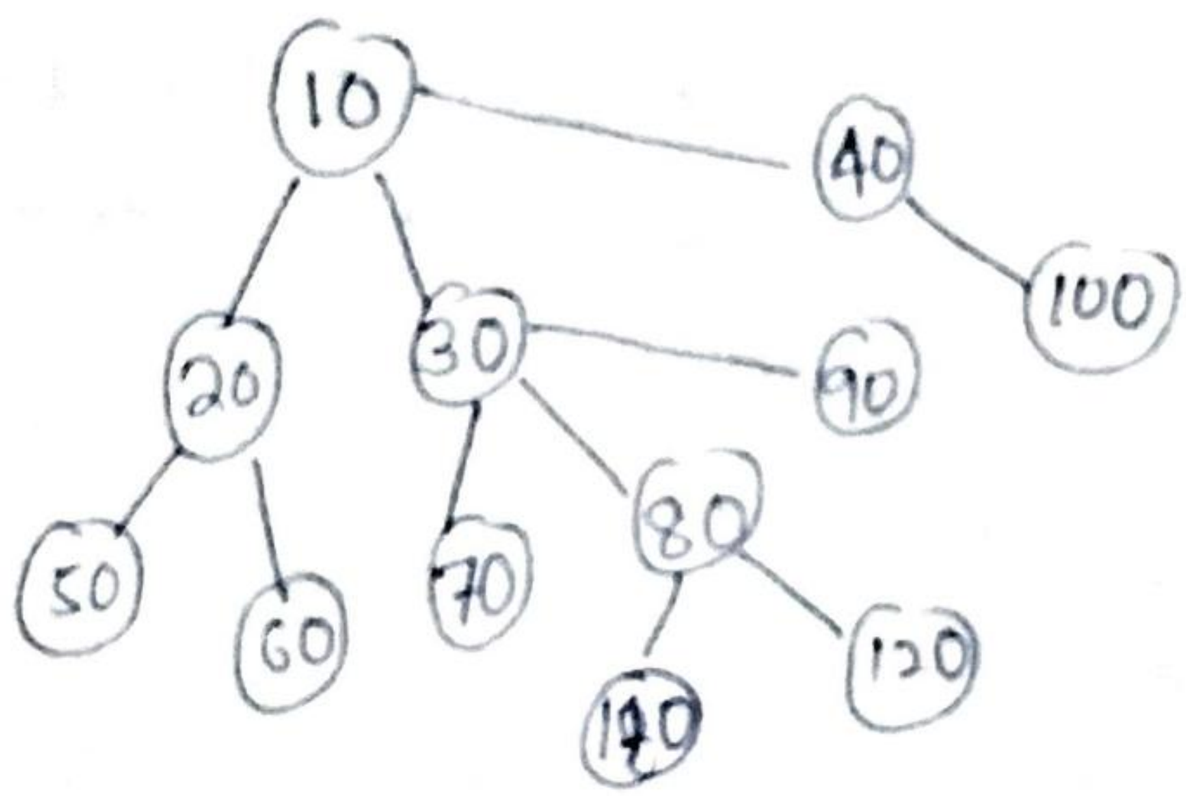
24

10 20 50 -1 60 -1 -1 30 70 -1 80 110 -1 120 -1 -1 90 -1 -1 40 100 -1 -1 -1 ;

120

**output**

[120, 80, 30, 10]




**Expectation**

to return node to root path in a ArrayList

| | | | |
|---|---|---|---|
| 120 | 80 | 30 | 10 |

**faith:-**

Every child will return node to root path in a arraylist

↳ if node not found return empty Arraylist

**Steps**

* First check if the Node's data (means root data) which is given to us is equal to the elements data which we are searching then
  → make a new Arraylist of Integer type
  → add the Node's data in the list
  → Now return the list.

Scanned by TapScanner

* Now execution check whether it's child having the element that we are searching Now call the child Node & its children Node and create a resultant list of Integer type where it will return function of Node to root path for all child

for (Node child : node.children)

{

ArrayList<Integer> rlist = Node to root path (child, data);

* Now check if the rlist is empty or not if size is greater than zero then ~~add~~ ~~the root (given no~~ means the required element is returning then add the current node's data and return the Arraylist.

* If we didn't get the element then return an empty ArrayList.

return new ArrayList<Integer>();

Code

public static ArrayList<Integer> nodeToRootPath (Node node, int data)

{

if (node.data = data)

{ ArrayList<Integer> list = new ArrayList<>();

```
list.add(data);
return list;
}
for(Node child : node.children)
{
ArrayList<Integer> rlist = nodeToRootPath(child, data);
    if(rlist.size() > 0)
    {
      rlist.add(data);
      return rlist;
    }
}
return new ArrayList<Integer>();
}
```

## Lowest-Common-ancestors of a Generic Tree

input

24
10 20 50 -1 60 -1 -1 30 70 -1 80 110 -1 120 -1 -1 90 -1 -1
40 100 -1 -1 -1

120
80

output

80

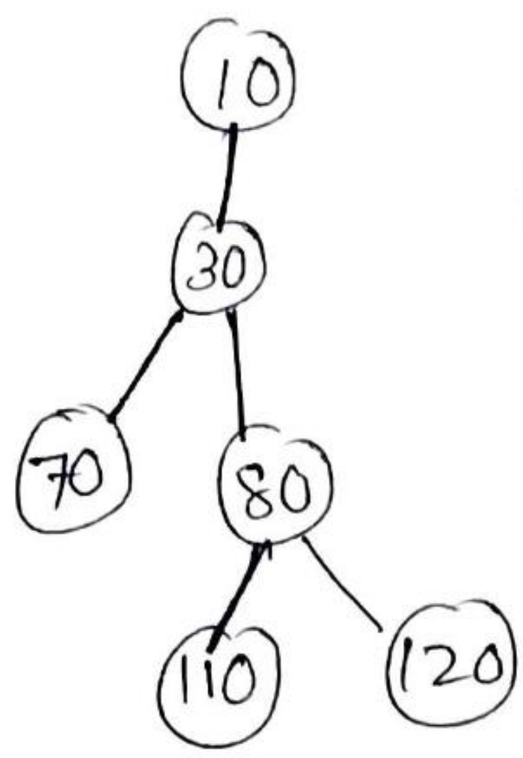
Scanned by TapScanner

We are given a tree as input and we are give two data $d_1 = 120$, $d_2 = 80$ and we have to find L-C-A (lowest common ancestors)

**LCA - first Common ancestore in path**

↳ while returing a Node which is given and to the Node which is given the Common element is called LCA

eg:-1



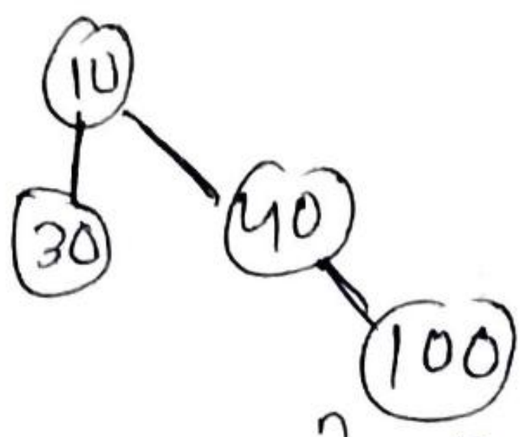

Here L-CA between 110 & 70 is is 30 becaus

→ 110's path to root Node [110, 80, 30, 10]

→ 70's path to root Node [70, 30, 10]

Lowest Common elemen betn two is 30 so LCA is 30.

eg:-2




LCA betn 30 & 100 is 10.

**Steps**

* We are given two data $d_1$, $d_2$, Now find node To root path of the two data which is given and store it in list1, and List 2.

ArrayList <Integer> list1 = node To root path (node, $d_1$)

ArrayList <Integer> list2 = node To root path (node, $d_2$)

Scanned by TapScanner

* Now create two pointer i & j and place them in last index of list1 & list 2

```
int i = list1.size() - 1
int j = list2.size() -1
```

* Now decrease the i and jth value till i & j becomes zero and till value at ith pos^n not equal to value at jth position

```
while(i>=0 && j>=0 && list1.get(i) == list2.get(j))
{
  i--;
  j--;
}
```

* Now when the loop stops the just before value is our required value. Now return (i+1) or (j+1)

list1.get(i+1) or list2.get(j+1);

Code

```
public static int lca(Node node, int d1, int d2)
{
ArrayList<Integer> list1 = nodeToRootpath(node, d1);
ArrayList<Integer> list2 = nodeToRootpath(node, d2);
int i = list1.size() - 1;
int j = list2.size() -1;
while(i>=0 && j>=0 && list1.get(i) == list2.get(j))
```

Scanned by TapScanner

```
{
    i--;
    j--;
  }
return list1.get(i+1);
}
```

## Remove Leaves in a generic Tree

We are given an tree and we have to remove ~~the tree~~ all the leaves from that tree

**input**

2 4

10 20 50 -1 60 -1 -1 30 70 -1 80 110 -1 120 -1 -1 90 -1 -1 40 100 -1 -1 -1

**output**

10 --> 20, 30, 40

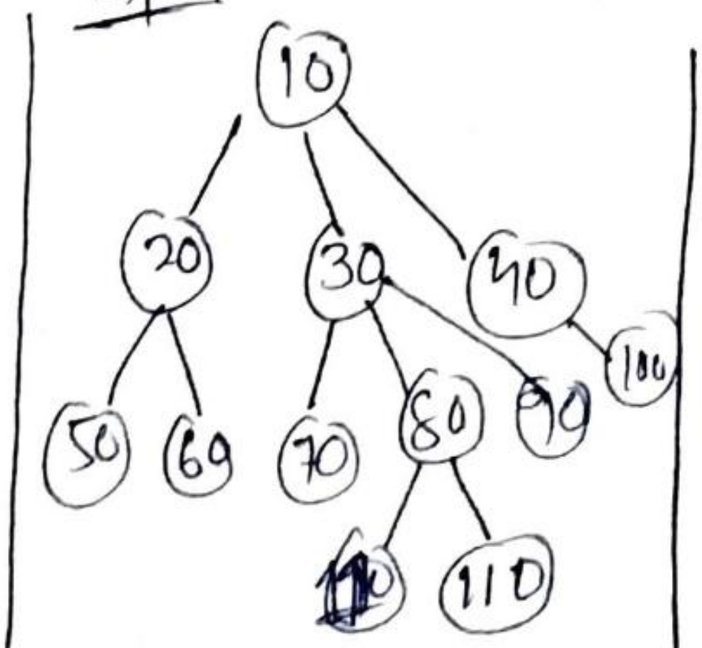
20 -->

30 --> 80

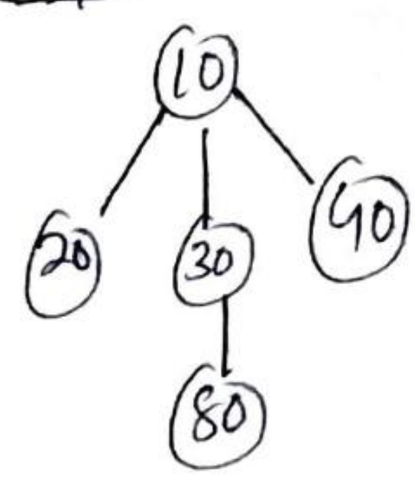
80 -->

40 -->

**input**




**o/p**




**steps**

* ~~Free remove~~ The children of a child in a tree are placed inside a Arraylist. In Arraylist we usually remove from last.

Scanned by TapScanner

because if we remove from last then order of index will be unchanged (If we done it from fast then will changed)

* get the index of the number of child ArrayList from it's children and call a loop which will strat from last index and will goes till idx >= 0 (last index).

* Now check if the size of child is zero or not if the size of child is zero then remove that idx).

* Now call every other child of that tree and call remove Leaves (child) function.

```
for (node child : node.children)
{ removeLeaves (child);
}
```

Code

```
public static void RemoveLeaves(Node node)
{
for (int idx = node.children.size()-1; idx=0; idx--){
{ Node child = node.children.get(idx);
```

Scanned by TapScanner

```
if (child.children.size() == 0)
  {
    node.children.remove(idx)
  }
}

for (Node child : node.children)
  {
    removeLeaves(child);
  }
}
```

Mirror of a generic-tree

input

24

10 20 50 -1 60 -1 -1 30 70 -1 80 110 -1 120 -1 90 -1 -1 40 100 -1 -1 -1

Output

10 -> 20, 30, 40

20 -> 50, 60

50 -> .

60 -> .

30 -> 70, 80, 90

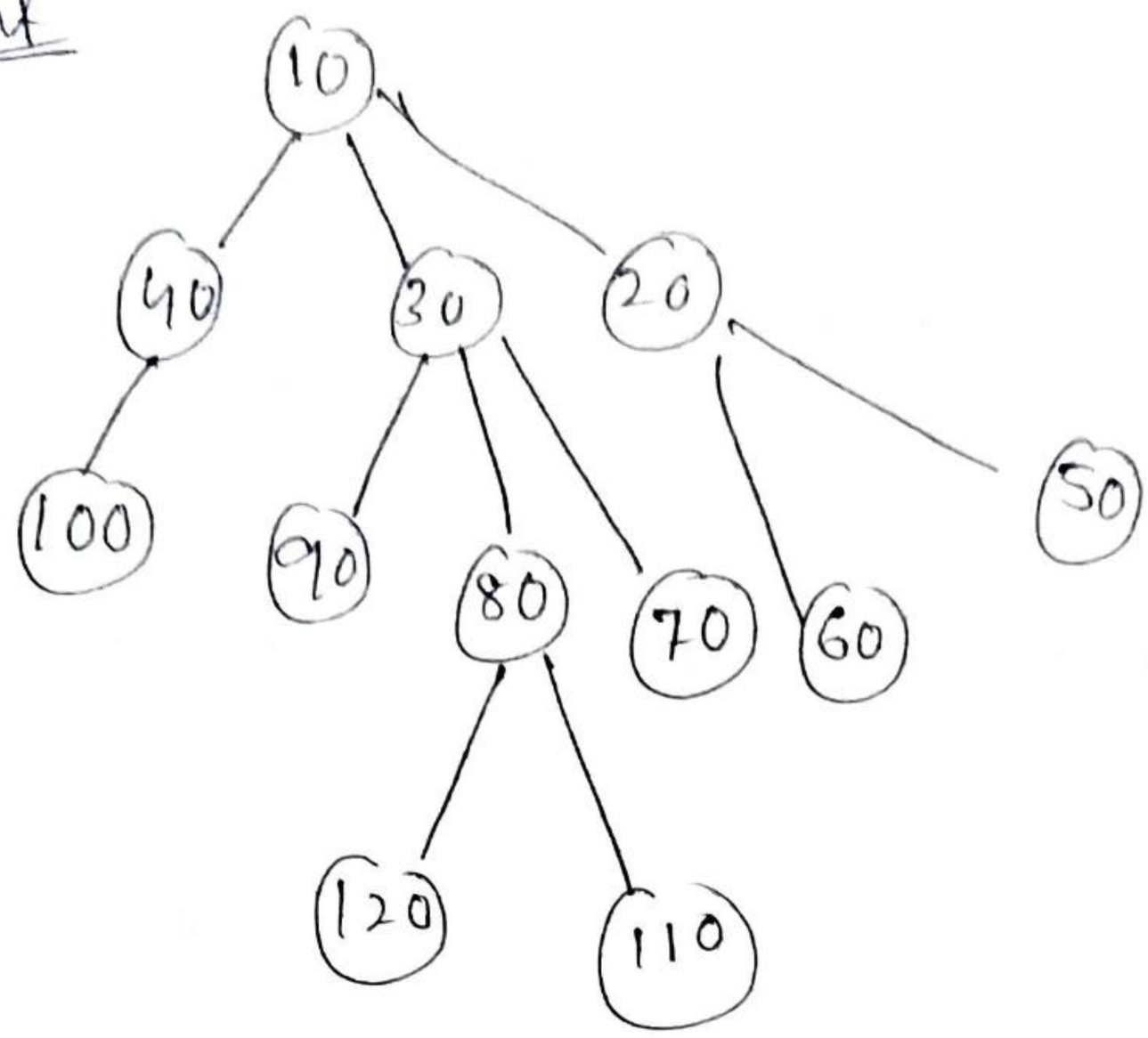
70 ->

80 -> 110, 120

Scanned by TapScanner

Here we have to find mirror of the given tree

output




Steps

* Reverse the children of root node first
  Collection.reverse (node.children);
* Now go to every children of the child node and reverse all the children node by calling mirror() function

Code

```
public static void mirror (Node node)
{
  Collections.reverse (node.children);
for (Node child: node.children)
    {
    mirror(child);
}  }
```

Scanned by TapScanner

# Are generic-Tree similar

**input**

2 4

10 20 50 -1 60 -1 -1 30 70 -1 80 110 -1 120 -1 -1 90 -1
-1 90 -1 -1 40 100 -1 -1 -1

2 4

1 2 5 -1 6 -1 -1 3 7 -1 8 11 -1 12 -1 -1 9 -1 -1 4 10 -1 -1 -1

**output**

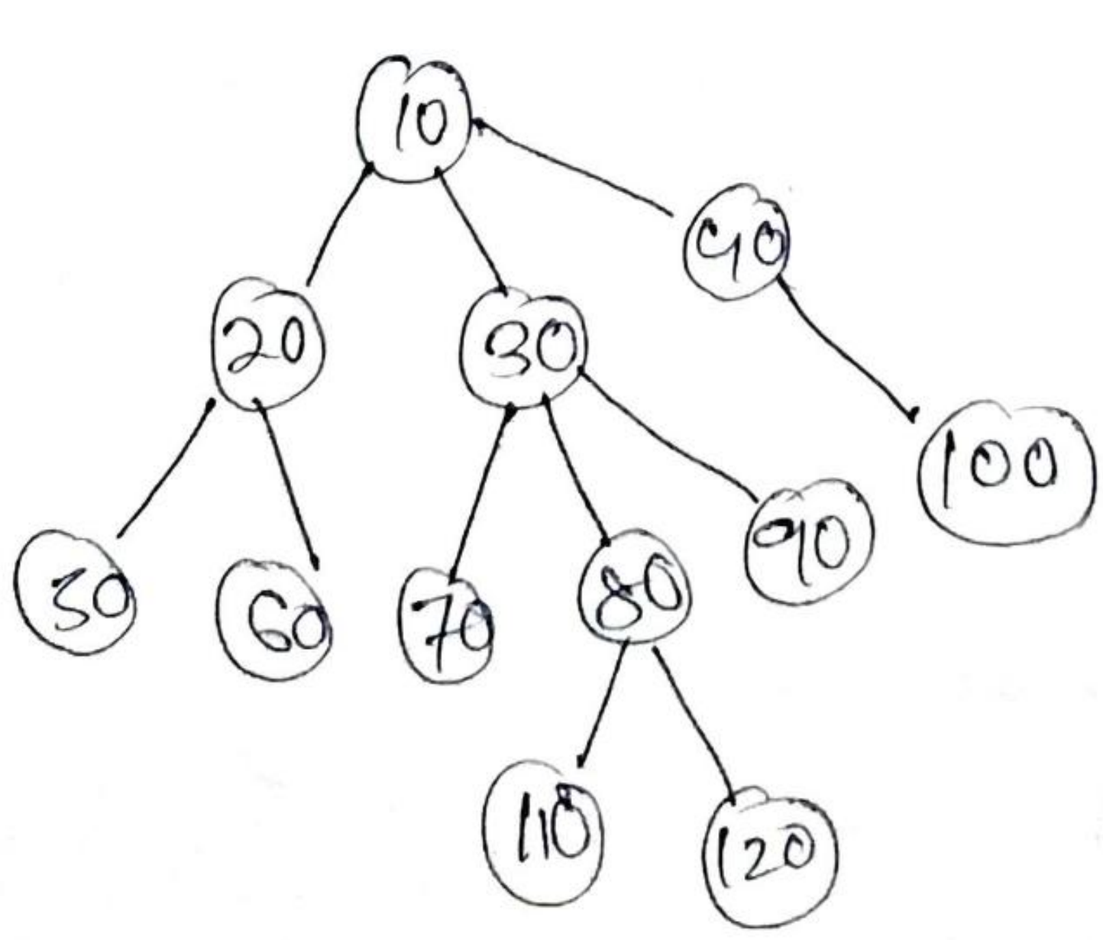
true

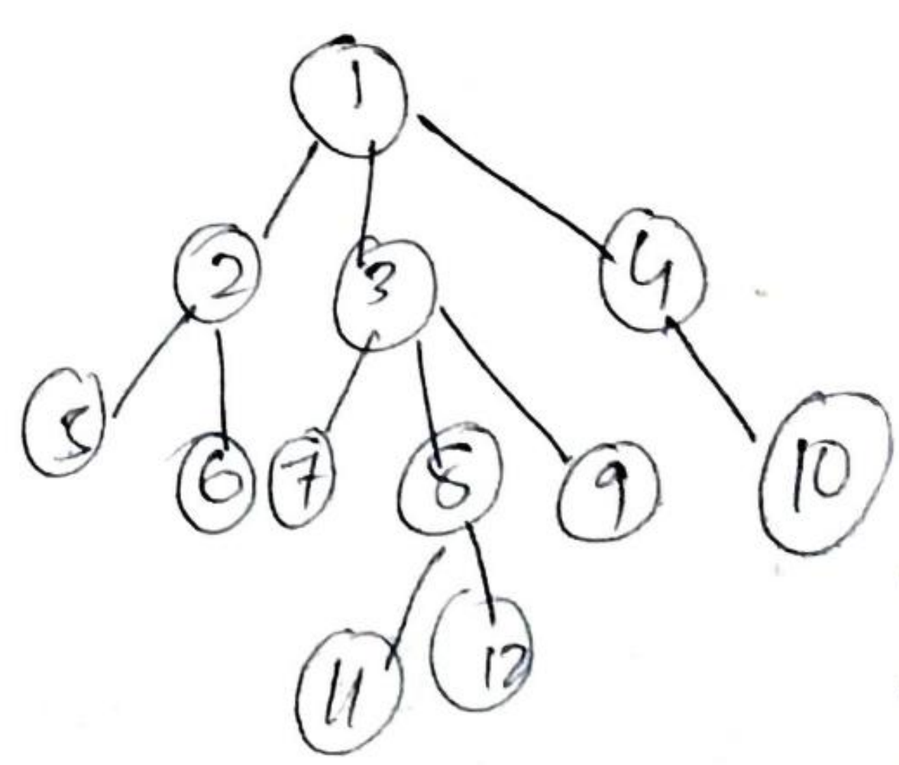
we are given two tree and we have to find is they are ~~equal or~~ similar or not.

In similarity basis we will check 2 things

1 - $n_1$.children.size() == $n_2$.children.size()

2 - ($n_1$.child, $n_2$.child) — similar




Scanned by TapScanner

we have to find if the no. of child of two tree are similar or not & and the child tree structure of two tree are similar or not.

Steps

→ 1st check if given node 1 ($n_1$) size and node 2 ($n_2$) size are not equal then return false

→ Now get the index of every children by starting a for loop from index '0' to size.

→ Now calculate two child node by calculating their index

Node child1 = n1.children.get(idx)
Node child2 = n2.children.get(idx)

and now start from 0th index and go till size and ~~find at~~ call the function areSimilar() child1, child2 are similar or not

→ Now After comming out from both if & for loop return true.

Scanned by TapScanner

Code

```
public static boolean areSimilar(Node n1, Node n2)
{
  if(n1.children.size() != n2.children.size())
  {
    return false;
  }
  for(int idx=0; idx<n1.children.size(); idx++)
  {
    Node child1 = n1.children.get(idx);
    Node child2 = n2.children.get(idx);
    if(areSimilar(child1, child2) == false)
    {
      return false;
    }
  }
  return true;
}
```

Are Trees mirror in shape

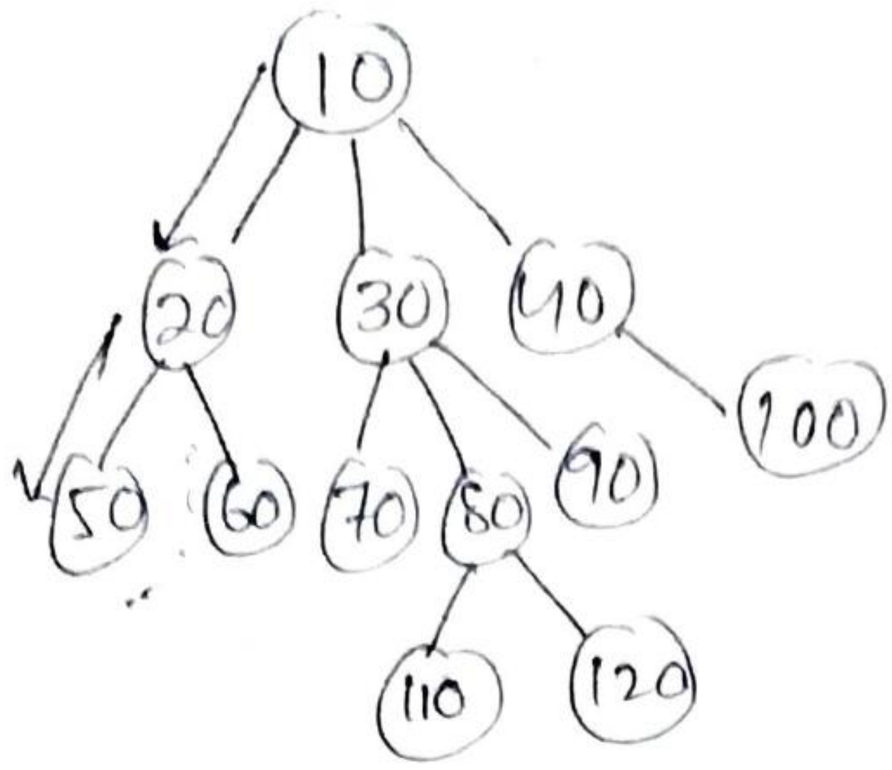
input

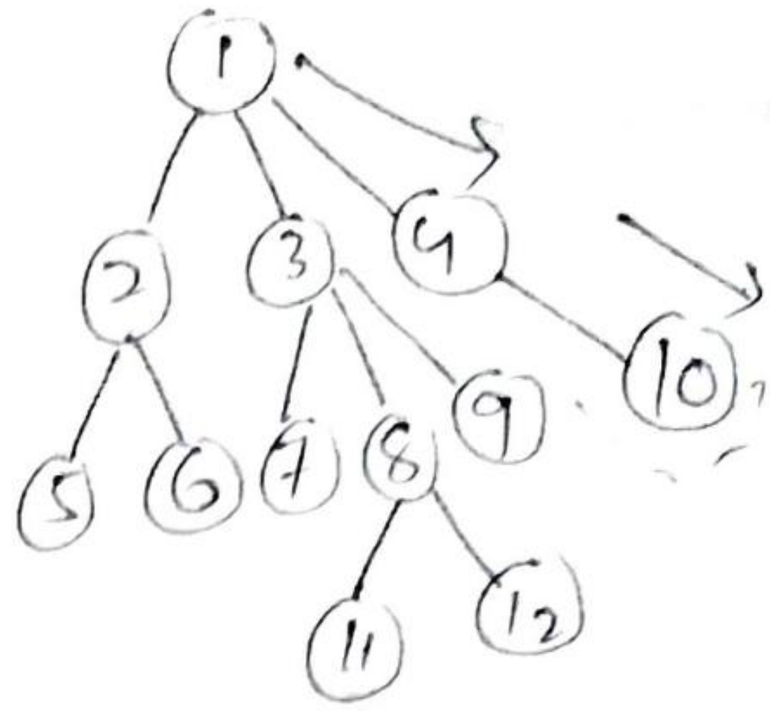
12
10 20 -1 30 50 -1 60 -1 -1 40 -1 -1

12
100 200 -1 300 500 -1 600 -1 -1 400 -1 -1

output

true




## Steps

* It is similar to trees are similar or not problem.
* Here also same 1st check if size of 1st tree is equal to 2nd or not.
* Now check ~~the~~ the child of two tree, ~~of~~ child of two trees ~~are~~ will be check by calculating index of child1 & child 2. (first idx)
* Test child1 from index = 0 and child 2 from from last index (~~n2.size().get(idx~~ ~~get~~ n2.children.idx-1)
  if false return false
* At last if two cond^n not satisfied return true.

## Code

```
public static boolean areMirror (Node n1, node n2)
  {
   if (n1.children.size()!= n2.children.size())
     {
      return false;
     }
```

Scanned by TapScanner

```
for (int idx = 0; idx < n1.children.size(); idx++)
  {
  Node child1 = n1.children.get(idx);
  Node child2 = n2.children.get(n2.children.size()-idx-1)

  if (areMirror(child1, child2) == false)
      {
        return false;
      }
  }
return true;
}
```

## Is-generic Tree Symmetric

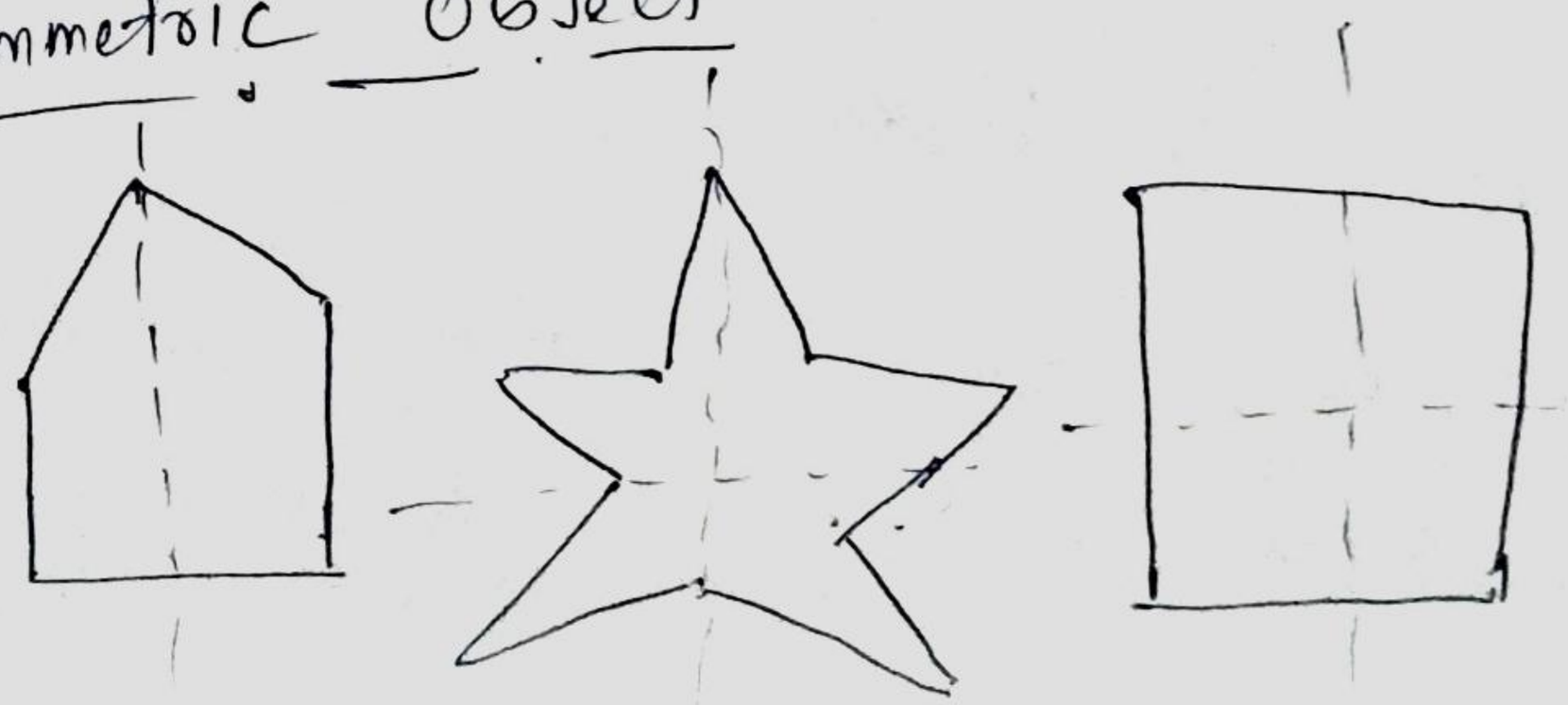
**input**

20
10 20 50 -1 60 -1 -1 30 70 -1 80 -1 90 -1 -1 40 100
-1 110 -1 -1 -1

**output**

true

## Symmetric object



Property of symmetry

(1) mirror image overlap

Scanned by TapScanner

Steps

* A tree will be called symmetric tree if it is mirror of itself.
* 1st find the mirror of two tree
* Now call the mirror function betn two root Node of same tree, pass the root node 2 times
* If they return true then the tree is symmetric

Code

```
public static boolean areMirror (Node n1, Node n2)
{
  if (n1.children.size() != n2.children.size())
    {
      return false;
    }
  for(int idx=0; idx< n1.children.size(); idx++)
    {
      Node child1 = n1.children.get(idx);
      Node child2 = n2.children.get(n2.children.size()
                                    - idx - 1);
      if (areMirror (child1, child2) == false)
        {
          return false
        }
    }
  return true;
```

Scanned by TapScanner

```
public static boolean IsSymmetric (Node root)
{
return areMirror(root, root);
}
```

N.B

A tree will be called symmetric if it is mirror of itself, so here we are passing root Node two times if two mirror are equal then it will be symmetric.

## Ceil and floor of a generic Tree

input

24
10 20 -50 -1 60 -1 -1 30 70 -1 -80 110 -1 -120
-1 -1 90 -1 -140 -100 -1 -1 -1

Ceil → 90
Floor → 70

output

CEIL = 90
FLOOR = 60

Ceil - Minimum among Larger
floor - Maximum among smaller

Ceil = ∞ → Integer. MAX_VALUE
Floor = -∞ → Integer. MIN_VALUE

Scanned by TapScanner

Steps

* 1st calculat it node's data is greater than the given value or smaller than the given value
* ceil is minimum among larger, for ceil travel from root node when the node's data is greater than the value then, update it's value and continue this process, ~~At last~~ when we find a value which is minimum from updated ceil value then update it
* for floor value, it is the largest among smaller, start from root node and if the value is ~~less~~ greater than the default floor value (-∞) but smaller than the given value update it.
* Lastly find the CeilAndFloor( ) function for all children Node and print it

Code

```
public static void CeilAndFloor (Node node,
                                 int val),
{
 if (node.data > val)
  {
   if (node.data < ceil)
    {
     ceil = node.data
    }
  }
```

Scanned by TapScanner

```
else if (node.data < val)
 {
    if (node.data > floor)
     {
      floor = node.data
     }
 }

for (Node child : node.children)
     {
       CeilAndFloor (child, val);
     }
}
```

## Kth Largest element in a generic Tree

input

2 4

10 20 -50 -1 60 -1 -1 30 70 -1 -80 110 -1 -120 -1 -1 90 -1 -1 40 -100 -1 -1 -1

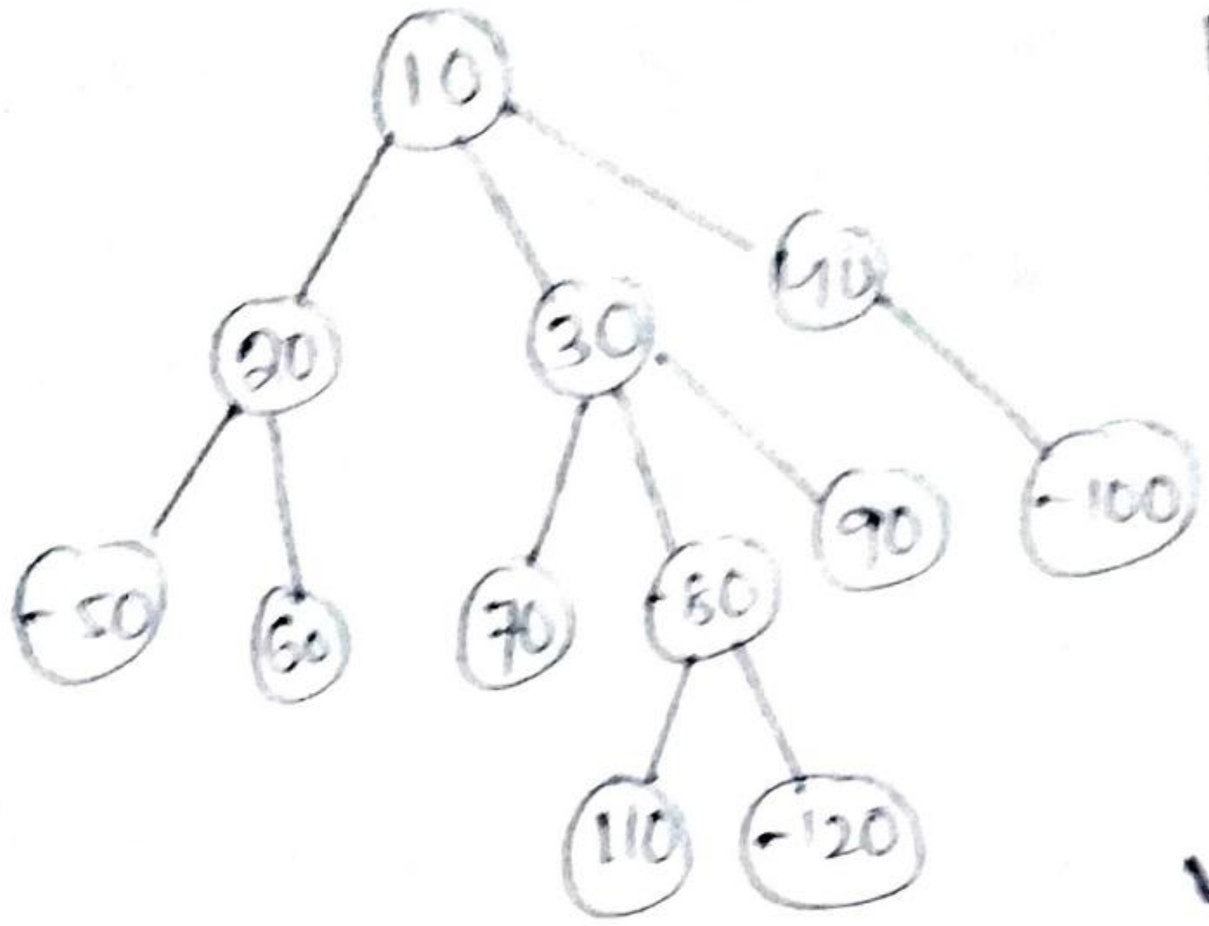
8

output

10

We are given a tree as input and also we are given to find the (Largest element), here given to find 8th largest element

HW




110 → 1st largest
90 → 2nd largest
70 → 3rd largest
60 → 4th largest
50 → 5th largest
30 → 6th largest
20 → 7th largest
10 → 8th largest
-50 → 9th largest
-80 → 10th largest
-100 → 11th largest
-120 → 12th largest

Here given to find 8th largest element, so we have to find 10

Steps

We will do it by calculating floor value from ∞ to the given index

(∞, 110) → floor is 110

| | floor | iteration |
|---|---|---|
| (∞, root, ∞) → | 110 | 1 |
| (root, 110) → | 90 | 2 |
| ⋮ | | ⋮ |
| (root, 20) → | $\boxed{10}$ | 8 |

* Create an ans variable and place it at ∞ value (Integer. MAX_VALUE)

int ans = Integer. MAX_VALUE

Scanned by TapScanner

* Now continue a for loop from i = 0 to given Kth value i.e 8.
* Now set floor value as $-\infty$ i.e

  floor = Integer.MIN_VALUE
* Now call the ceilAndFloor function and pass the root node and ans.
* Now update floor value with ans,

  lastly return ans.

Code

```
public static int KthLargest(Node root, int k)
{
int ans = Integer.MAX_VALUE;
for(int i=0; i<k; i++)
 {
  floor = Integer.MIN_VALUE;
  ceilAndFloor (root, ans);

  ans = floor;
 }
return ans;
}
```

Scanned by TapScanner

# Diameter of a generic Tree

**input**

20

10 20 -50 -1 60 -1 -1 30 -70 -1 80 -1 90 -1 -1 40 -100 -1 -1 -1

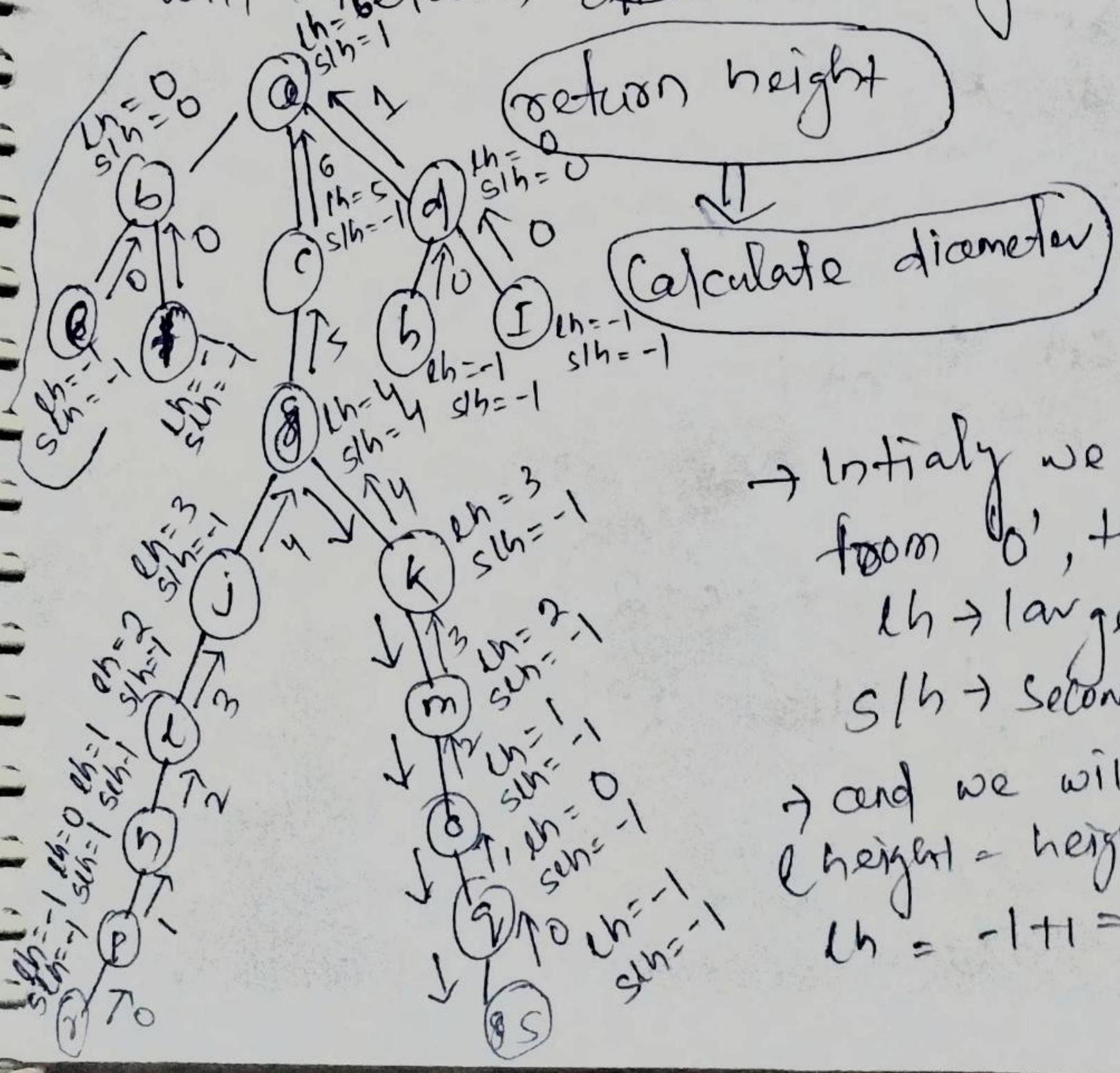
**output**

4

We are given to calculate diameter

$$\boxed{\text{diameter} = \text{largest height} + \text{Second largest height} + 2}$$

Here we will calculate diameter but we will return ~~diameter~~ height




→ Intialy we will start from '0', there lh → largest height = -1

s/h → Second largest height -1

→ and we will return height+1

height = height +1

lh = -1+1 = 0 ⇒ 0+1=1 ... so on

steps

* Create an int dia and set it static
  static int dia;
* Now create a function diameter pass the root node.
* Now set (largest height) lh = -1 and second lh = -1;
* Now call every child and ~~place the~~ Call the diameter function for every child.
* Now save the one in a child height variable
  int cht = diameter (child)
* check if cht > lh or cht > slh
  if cht > lh is true then
  set slh = lh;
  and lh = cht;
* if childheight is greater than second largest height then set
  slh = cht;
* Now check if the diameter is greater than (lh + slh + 2), if true
  then set dia = lh + slh + 2
* Now return height as largest height + 1
  return lh + 1

Scanned by TapScanner

* Lastly print the dia value

Code

```
static int dia;
public static int diameter(Node node){
  int lh = -1 , slh = -1;
  for (Node child: node.children)
  {
  int cht = diameter(child);

    if (cht > lh)
    {
      lh = slh;
      lh = cht;
    }
    else if (cht > slh)
      {
        slh = cht;
      }
  }
  if ( lh+slh+2 > dia )
  {
    dia = lh+slh+2;
  }
  return lh+1;
}
```

Scanned by TapScanner

```
P s v m (string[] avgs) throws Exception
{

}
dice = 0;
Node root = Construct (arr);
diameter(root);
System.out.println(dia);
```

## Predecessor and Successor of An Element

**input**

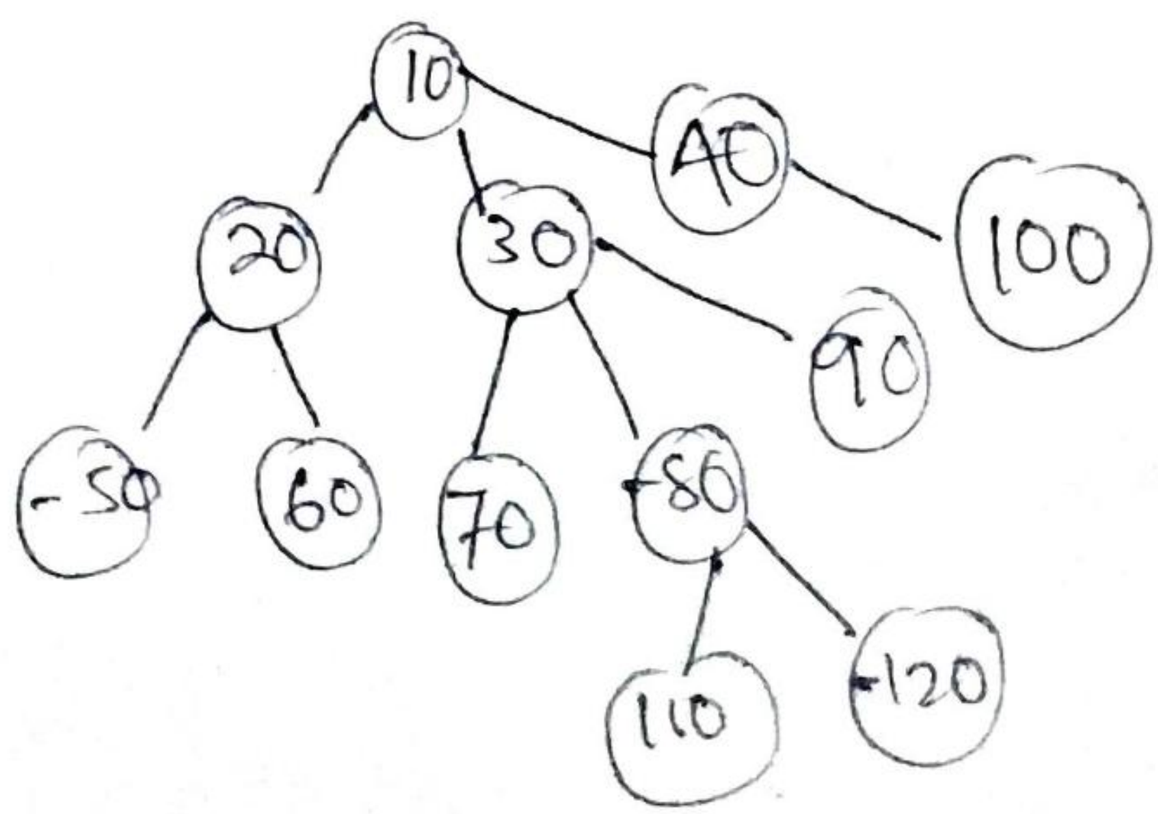
2 4
10 20 -50 -1 60 -1 -1 30 70 -1 -80 110 -1 -120
-1 -1 90 -1 -1 40 -100 -1 -1 -1

-120

**output**

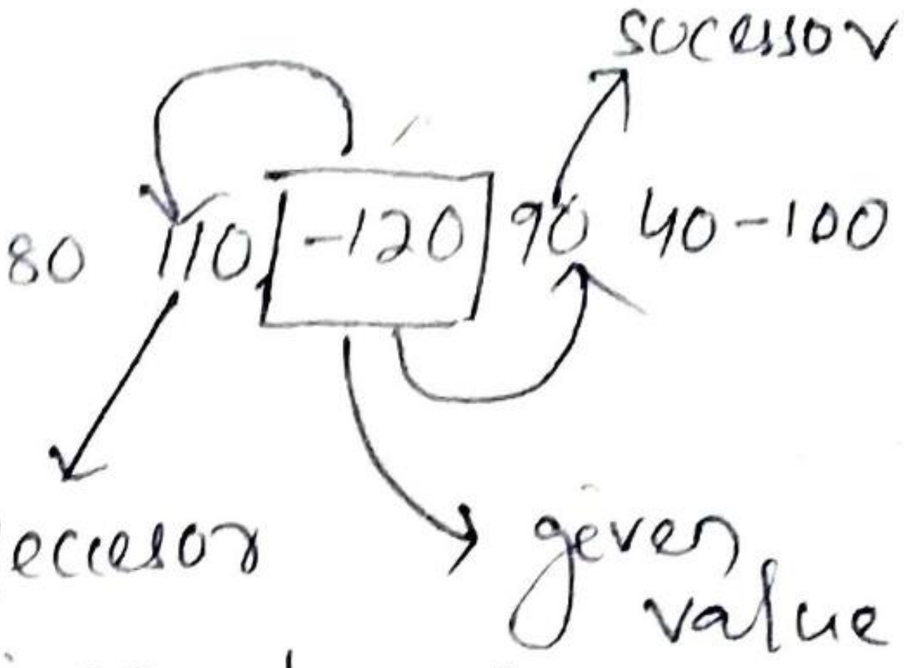
predcessor = 110
successor = 90

we are given val = -120 & a tree




Scanned by TapScanner

if we want to find predecessor and successor then we will move ~~bam~~ in pre order wise or pre order Traverse




predecessor → Previous of given element
successor → next of given element

Steps

* Intialy create ~~a state~~ 2 static Node variable → predecessor, successor, ~~state~~ & a static variable state.
  ↳ ~~gives~~ set predecessor, successor = null &
  state = 0
* Now create a predecessor AND successor function and pass Node.
* check if state is 0 then just check if node's data is equal to our required's data or not, if true then increase the state into 1.

```
if (node.data == data)
{ state = 1
}
```

else if node's data not equal to given data the ~~update~~ the ~~node~~ predecessor with node's data
* Now call every child and do the same process.
  ↳ { for (Node child: node.children)
  predecessorANDsuccessor(child);

Scanned by TapScanner

* if the node's data is equal to the the given data that is if state belome 1 then update sucessor with node's data and change the state to 2.

```
else if (state==1)
    {
      sucessor = node;
    } state = 2;
```

* For state 2 do nothing.

Code

```
static Node predecessor;
static Node sucessor;
static int state;
public static void predecessorAndsucessor(Node node)
  {  if (state==0)
     {
      if (node.data==data)
        {
          state = 1;
        }
      else {
          predecessor = node;
        }
     }
  }
else if (state==1)
      {
        sucessor = node;
      } state = 2;
```

Scanned by TapScanner

```
else if (state==2)
  {
    //do nothing
  }

for (Node child : node.children)
  {
    predecessorAndsucessor (child, data);
    }
  }

p s v m (string [] args)
  {
    }
  }
Node root = Construct (arr);
  predecessor = null;
  sucessor = null;
  state = 0
predecessorAndsucessor (root, data);
```

Scanned by TapScanner

# Node with maximum sub-tree sum

**input**

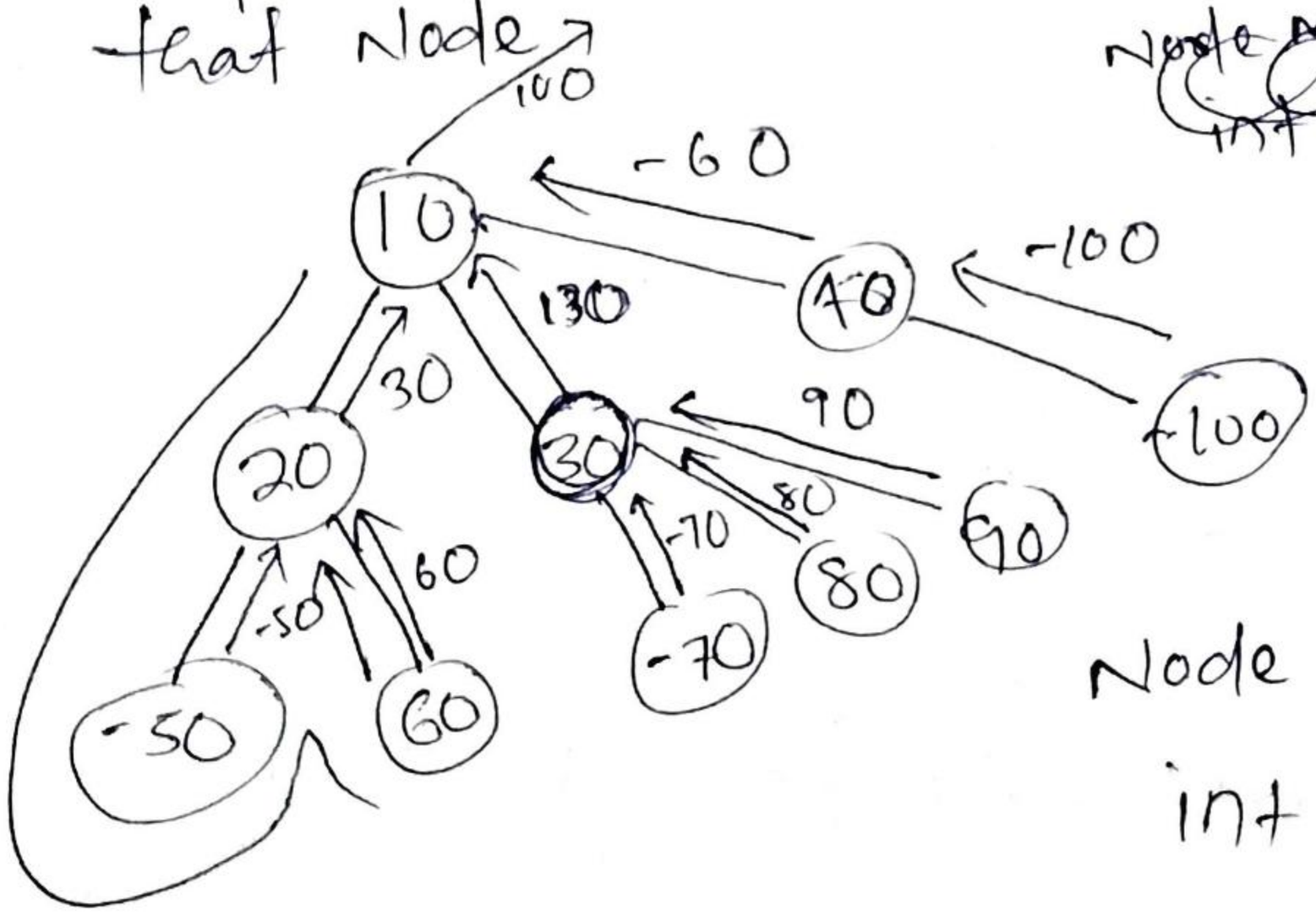
20
10 20 -50 -1 60 -1 -1 30 -70 -1 80 -1 90 -1 -1
40 -100 -1 -1 -1

**output**

30 @ 130

We are given a tree and we have to find the maximum sum of subtree and that Node

~~Node max sum Node: int sum =~~




Node max sum Node = 30

int sum = 130

* Here we are calculating all the node's data in pre, 1st we move to the last children then return its value as maximum value and do this for all children of a child, now add all childrens data with child's node's data and return it, do this for all child, lastly for root node

the ~~child~~ node where we get highest value return as sum & return node's data as highest sum node.

steps

* create a maximum sum node variable and intialize it to 0.

static int msn = 0;

* Also create maximum sum variable and intialize it to $-\infty$

static in ms = Integer.MIN_VALUE;

* Now call a function refsumAndCalculateMSST and pass the node

public static int refsumAndCalculateMSST()

* First create a sum variable and give its zero value

int sum = 0;

* Now travel to all children of child and ~~of~~ calculate child sum for all child by passing refsumAndCalculateMSST function, Now update sum = + csum (with all child's sum)

for (Node child : node.children)

{ int csum = refsumAndcalculateMSST (child);

sum = sum + csum;

}

Scanned by TapScanner

* Now add all node's data to sum variable update the sum

  sum = sum + node.data;

* check if sum greater than maximum sum or not if sum greater than maximum sum then update maximum sum with node's data and update maximum sum as sum

* Lastly return sum.

Code

```
static int msn = 0;
static in ms = Integer. MIN-VALUE;
public static in retsumAndCalculateMSST (Node node)
{
  int sum = 0;
  for (Node child : node.children)
  {
    int csum = retsumAndCalculateMSST (child);
    sum = sum + csum;
  }
  sum = sum + node.data;
  if (sum > ms)
  {
    msn = node.data;
    ms = sum;
  }
  return sum;
}
```

Scanned by TapScanner

# Linearize a generic tree

**input**

24

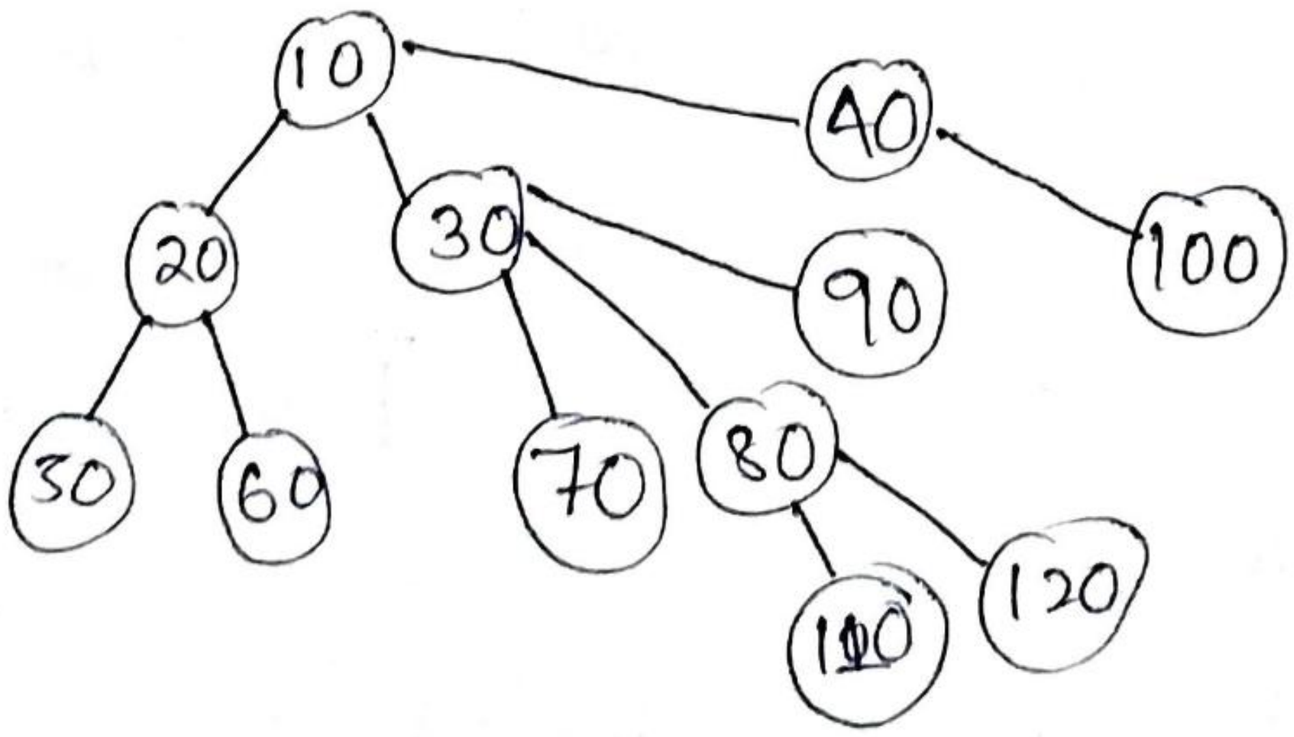
10 20 50 -1 60 -1 -1 30 70 -1 80 110 -1 120 -1 -1
90 -1 -1 40 100 -1 -1 -1

**output**

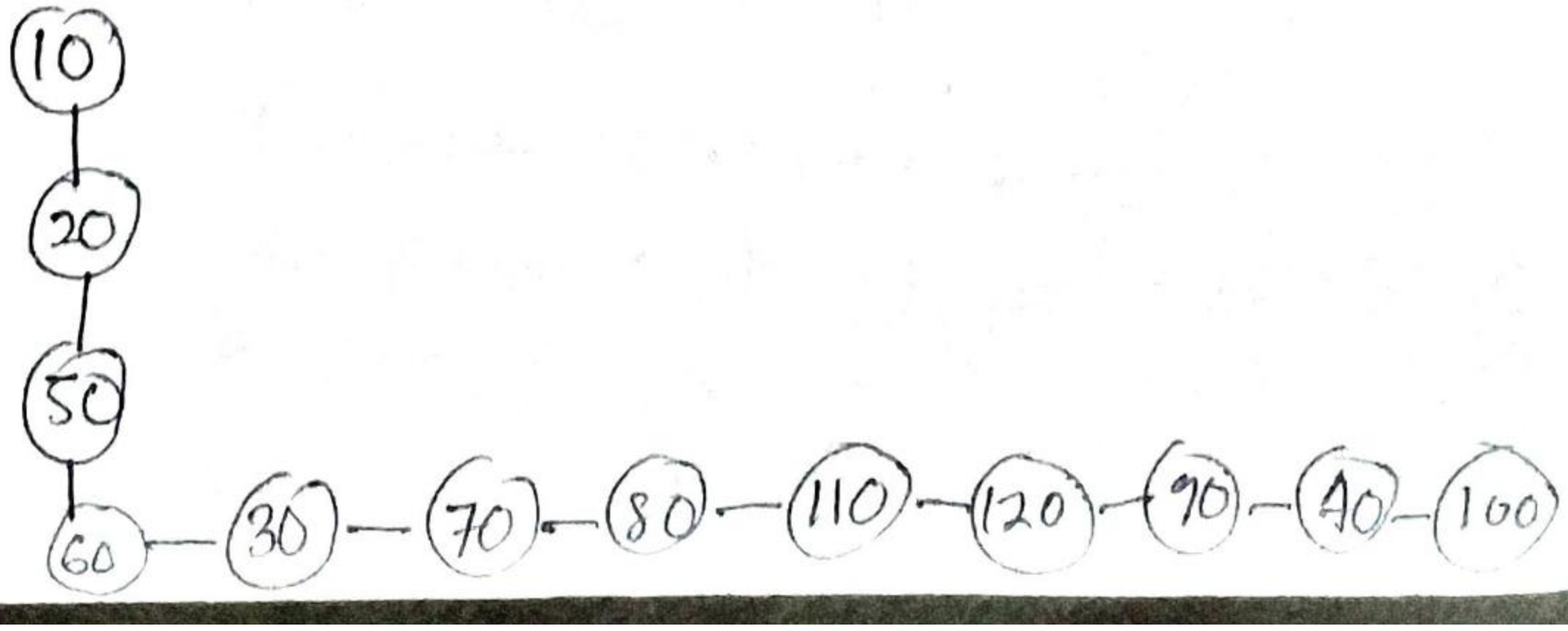
10 → 20, 120 → 90,
20 → 50, 40 → 100,
50 → 60,
60 → 30,
30 → 70,
80 → 110,

We we have given a tree and we are require to make it linnear by making all children as linnear.

**input**




**output**




Scanned by TapScanner

Steps

* First create ie Linearize function and pass node.
* Now move to every child of the node and call the Linearize function for every child.
* Till the size of node is not equal to 1 or greater than 1
  - ↳ Remove child get LastNode by removing last children

    Node lc = node.children.remove(node.children.size() -1)
  - ↳ get secondlastchildNode by just calculating index of SecondLast Node

    Node slc = node.children.get(node.children.size() -1)
  - ↳ Store the secondlastchildNode as tail node and call get TailNode function

    Node tail = getTailNode (slc)
  - ↳ Now add the lastchild in the tail of the tree

    tail.children.add(lc) at first
* Create a function getTailNode and pass the node, Here we have to get the oth children node so till the children's size is

Scanned by TapScanner

not greater than zero, get the node at index 0
lastly return that node.

Code

```
public static Node getTailNode(Node node)
{
  while(node.children.size() > 0)
    {
      node = node.children.get(0);
    }
  return node;
}
public static void Linearize(Node node)
{
  for(Node child : node.children)
    {
      Linearize(child);
    }

  while(node.children.size() > 1)
  {
  Node lc = node.children.remove(node.children.size()-1);
  Node slc = node.children.get(node.children.size()-1);

  Node tail = getTailNode(slc);
  tail.children.add(lc);
}
```

Scanned by TapScanner

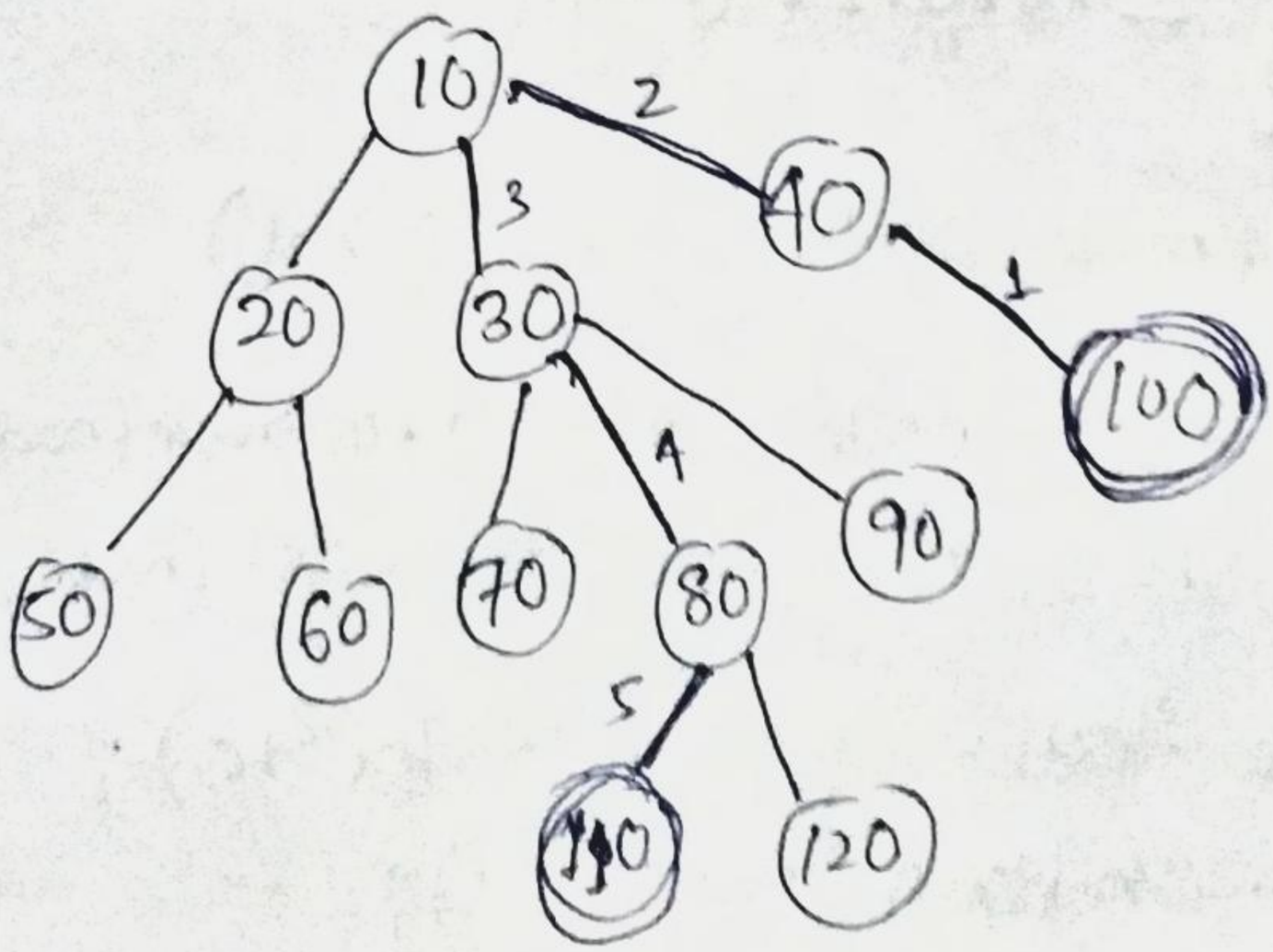
Distance between two Nodes in a generic Tree

input

24
10 20 50 -1 60 -1 -1 30 70 -1 80 110 -1 120 -1 -1
90 -1 -1 40 100 -1 -1 -1
100
110

output
5

Here we are given a tree and ~~two~~ we are given two Node's we have find out distance between then in the term of edges




Here we have to find distance betⁿ 100 & 110 in edges, the number of edges is 5, so we have to print 5 as output.

Scanned by TapScanner

Steps

* create a function distanceBetweenNodes() to calculate the distance betn nodes and pass the ~~root~~ root node and the given two value $d_1$, $d_2$

* Now calculate node to root path betn $d_1$ & $d_2$ from node and store it in $P_1$ & $P_2$ ArrayList $P_1$ & $P_2$.

* Now create ~~int~~ two integer i and j and put them on the last index of $P_1$ & $P_2$

```
int i = P1.size()-1;
int j = P2.size()-1;
```

* Now check the condn when i & j greater than and equal to zero also ~~P1 is~~ index of $P_1$ is equal to index of $P_2$, then decrease i and j value by 1.

* Here we calculated node to root path betn two ~~po~~ given point.

* Now the distance betn two point is the sum of edges betn them

* Now increase i by one and j by one so that we will get the common point.

* Now return i+j;

Scanned by TapScanner

Code

```
public static int distanceBetTwoNodes(Node node, int d1,
                                      int d2)
{
  ArrayList<Integer> P1 = nodeToRootPath(node, d1);
  ArrayList<Integer> P2 = nodeToRootPath(node, d2);

  int i = P1.size() - 1
  int j = P2.size() - 1

  while(i>=0 && j>=0 && P1.get(i) == P2.get(j))
  {
    i--;
    j--;
  }
  i++;
  j++;
}
return i+j;
}
```

Scanned by TapScanner

# iterative pre order and post order

**input**

24

10 20 -50 -1 60 -1 -1 30 70 -1 -80 110 -1 -120 -1
-1 90 -1 -1 40 -100 -1 -1 -1

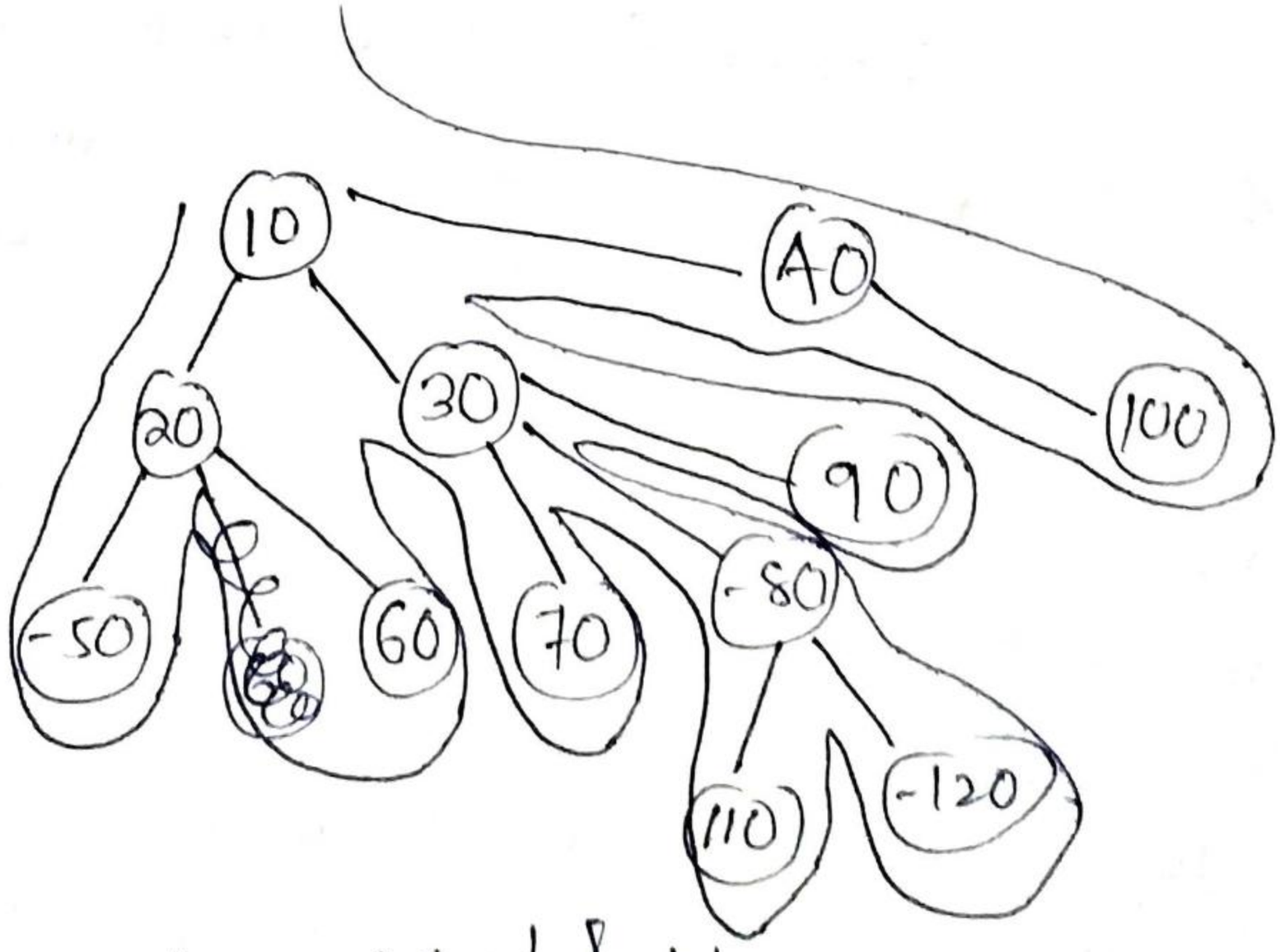
**output**

10 20 -50 60 30 70 -80 110 -120 90 40 -100

-50 60 20 70 110 -120 -80 90 30 -100 40 10

we are given a tree and we have to find it's pre order, post order in iterative approch

**Steps**




-1 → Pre order, state ++

children size → post order, pop

get^n 0 to children.size() - 1 → push child, state ++

Scanned by TapScanner

Co. Steps

* create a class pair and also create a Node & state variable, store node in Node and int state in state. create a
* ~~Call the~~ constructor pair with node & state data and get these data

~~this.data = data;~~
this.node = node;
this.state = state;

* create a function Iterative pre and post order pass the node, 1st create a stack of integer pair type * create two string pre and post;
* push the node with state -1

stack <pair> st = new stack<>();
st.push(new pair(node, -1));

* check if stack is not empty (>0) then check the top value

pair top = st.peek()

* Now check if top's state is equal to -1 then it is pre order add top node's data into string pre and increase the state.
* If the state is equal to top node's children size then add top node's data into post string.
* Else if the state is bet^n 0 to children size, then create a new pair ~~by calculating state of top node and~~ and pass the

child as top node's child and state -1 and push the child pair into stack. Increase the state of top.

Code

```
static class pair {
        Node node;
        int state;
  pair (Node node, int state)
    {
      this.node = node;
      this.state = state;
    }
  }
public static void IterativePreAndPostOrder(Node node)
  {
    statck<pair> st = new stack<>();
    st.push(new pair(node, -1));
    ~~while(st.size()>0)~~
      String pre = "";
      String post = "";
    while (st.size() > 0)
      {
        pair top = st.peek();
        if (top.state == -1)
        {
          pre = pre + top.node.data + " ";
        top.state++;
        }
    else if (top.state == top.children.size())
```

Scanned by TapScanner

```
post = post + top.node.data + " ";
  st.pop();
}
else {
    pair cp = new pair (top.node.children.get (top.state), -1);
    st.push (cp);
    top.state ++;
  }
}
system.out.println (pre);
system.out.println (post);
}
```

Scanned by TapScanner