Talking: Shreesh Trip

Data Structures -

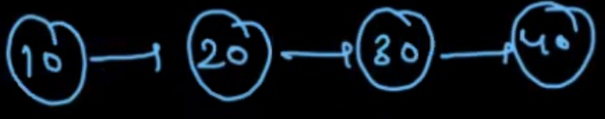
Linear Data Structure

1. Array & String
2. ArrayList
3. Stack
4. Queue
5. LinkedList

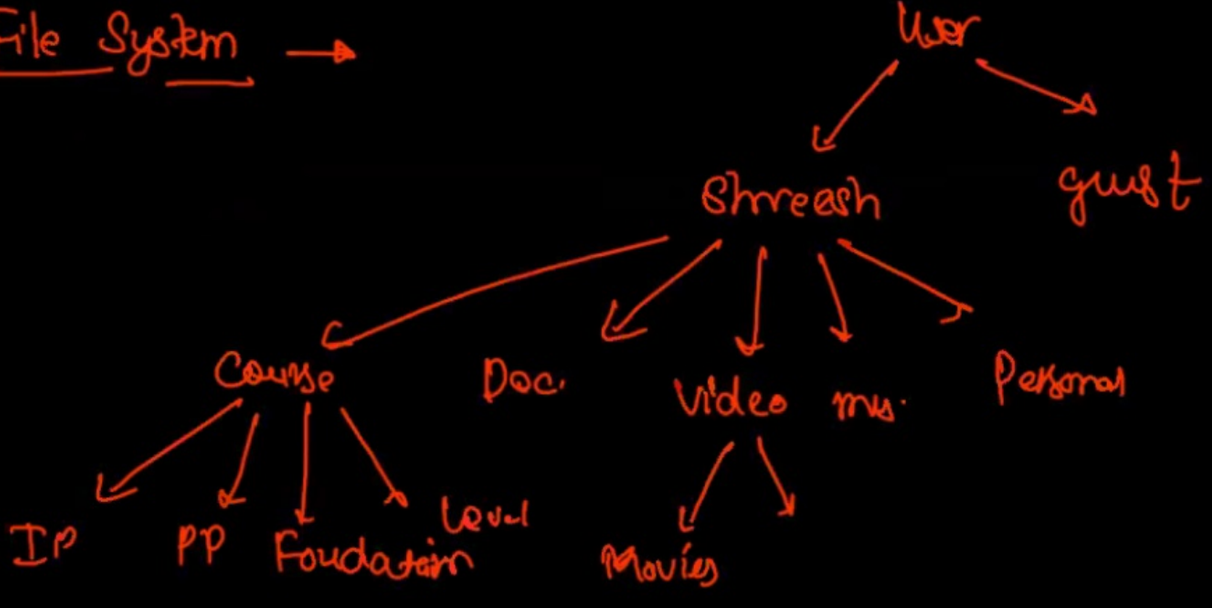
Memory continuous

StringBuilder.

Distributed



File System →



Heirarichal Data Structure -

Trees -

- Generic Tree
- Binary Tree
- Binary Search tree

Generic Tree →

Root ⇒ 10 having no parent

Parent ⇒ 20 → 10, 50 → 20

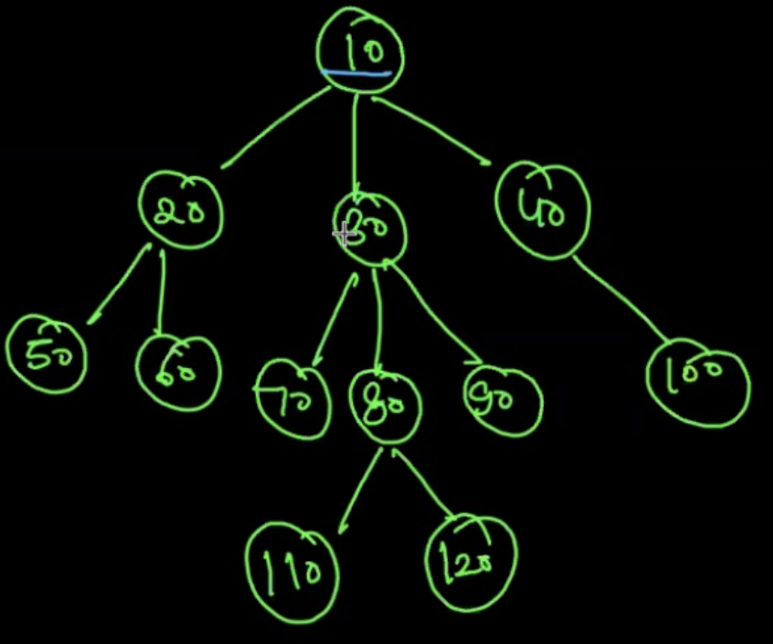
Child ⇒ 20 → 50, 60,

Ancestor ⇒ 50 → 20, 10

Descendent ⇒ 10 → all tree Except 10

Leaf ⇒ Node having no child.

Siblings ⇒ 50 — 60, 70 - 80 - 90



Node → int data;

next address → (Generic)

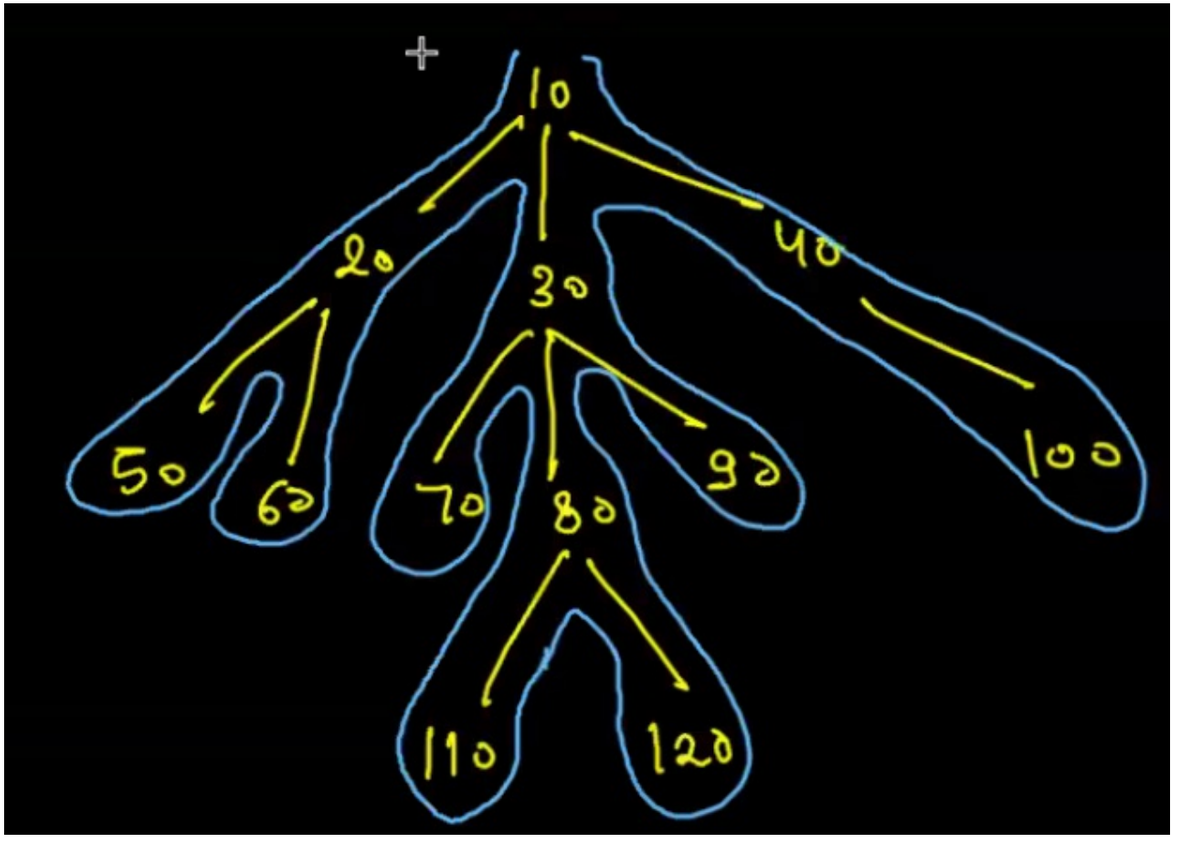
ArrayList<Node> children;

Construction : → Euler

PreArea = Value

PostArea = -1

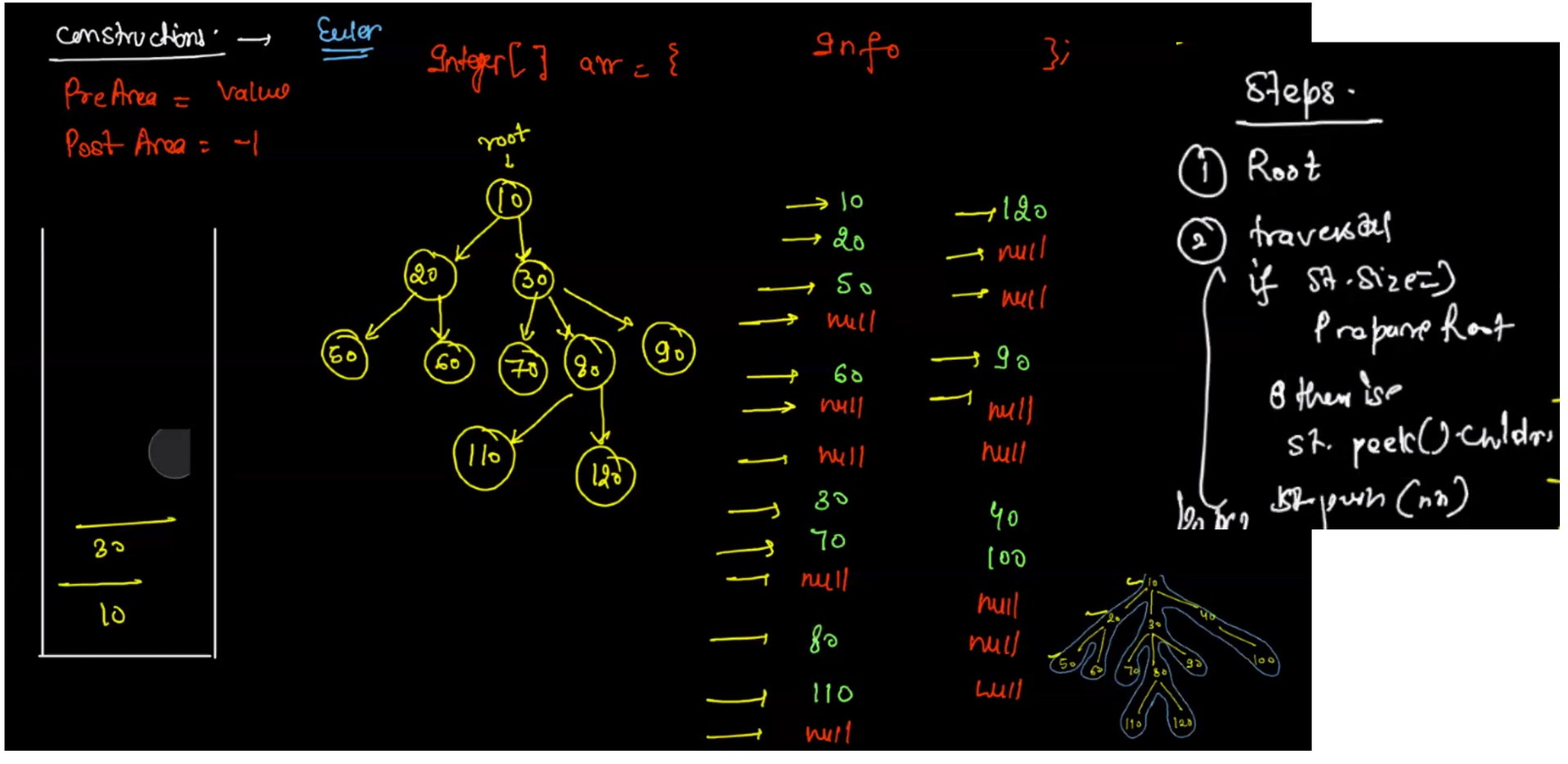
10	120
20	-1
50	-1
-1	90
60	-1
-1	-1
-1	30
40	70
100	-1
-1	80
110	-1
-1	



if you use int[] arr = new int[] --it will bydefault store 0

Integer[] data - new Integer[] -it will store null bydefault

we are constructing treee using data {10,20,50,null.....}



Approach:

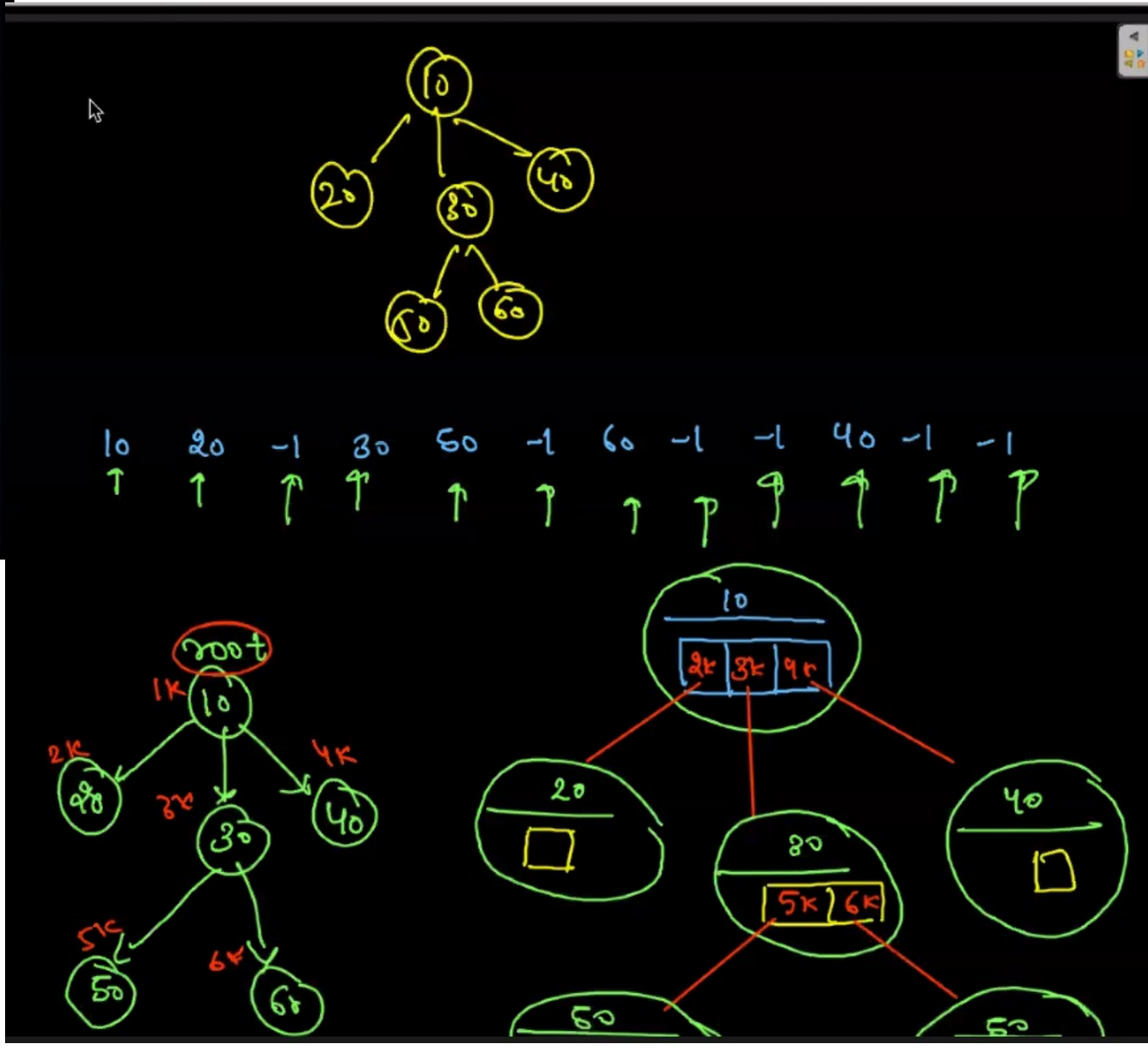
1. create new node;

2. stack k top me jo data he uske child me add kardia aur lkhudko push kar dia

2.agar null aya toh pop kardo

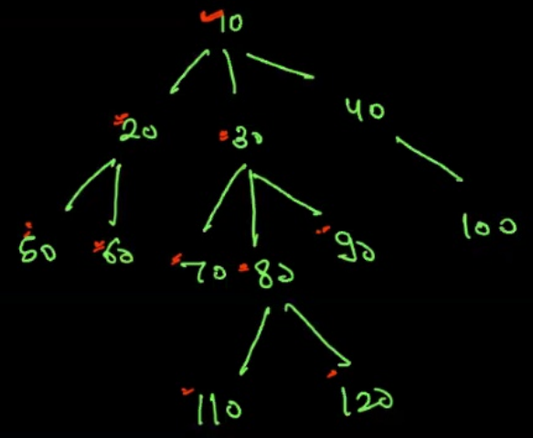
```java
public static Node construct(Integer[] arr) {
    Node root = null;
    Stack<Node> st = new Stack<>();

    for(int i = 0; i < arr.length; i++) {
        Integer data = arr[i];
        if(data != null) {
            Node nn = new Node(data);
            if(st.size() == 0) {
                root = nn;
                st.push(nn);
            } else {
                st.peek().children.add(nn);
                st.push(nn);
            }
        } else {
            st.pop();
        }
    }

    return root;
}
```

dryrun

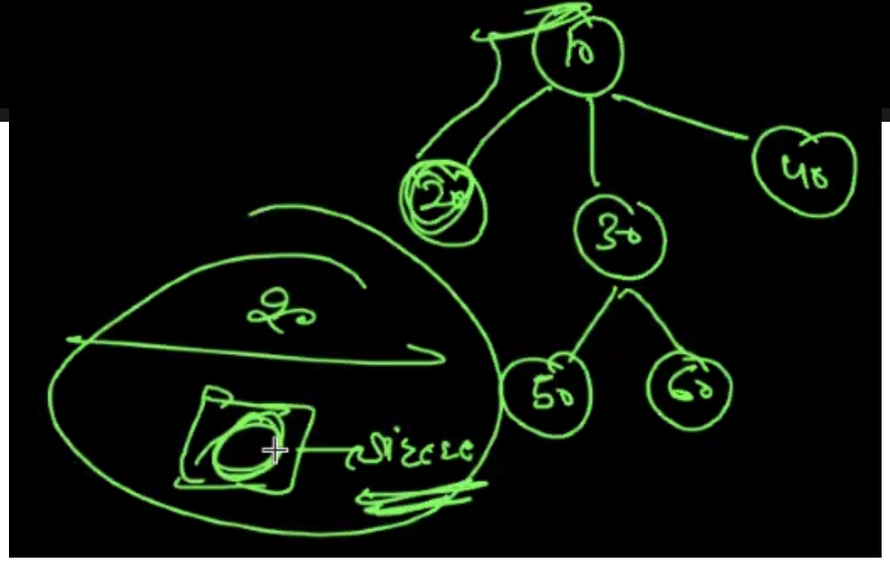


Display →

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



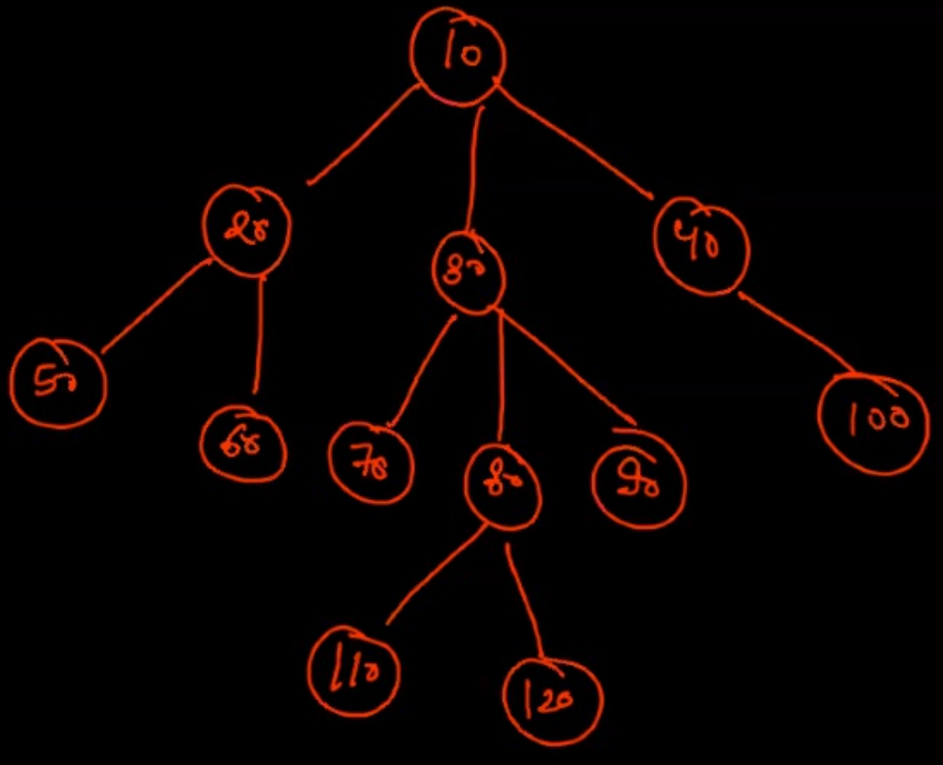

```
}
```

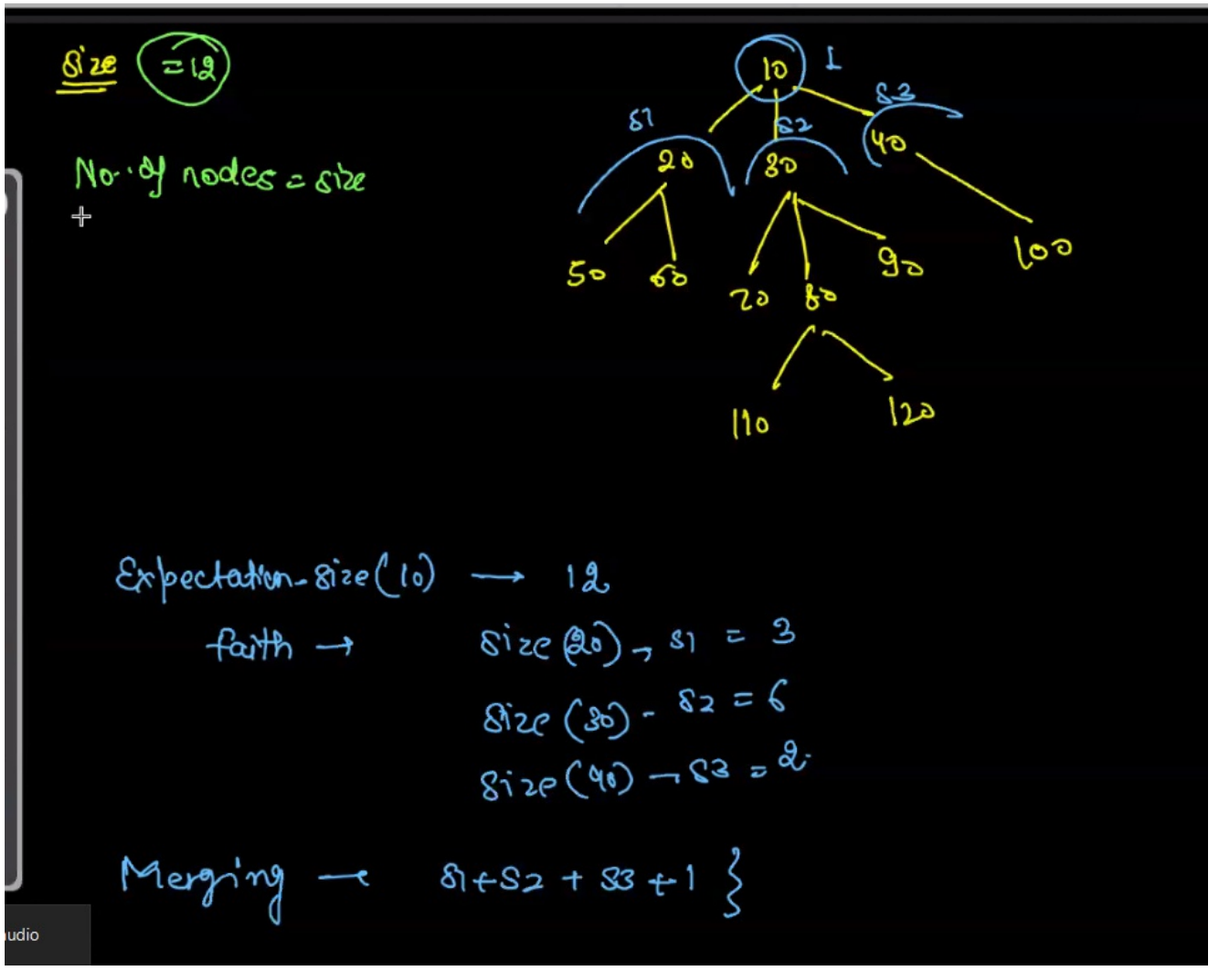



```java
public static void display(Node root) {

    String str = "[" + root.data + "] -> ";
    for(Node child : root.children) {
        str += child.data + ", ";
    }
    System.out.println(str + " .");

    for(int i = 0; i < root.children.size(); i++) {
        Node child = root.children.get(i);
        display(child);
    }
}
```

InOrder

```
1  [10] -> 20, 30, 40,  .
2  [20] -> 50, 60,  .
3  [50] ->  .
4  [60] ->  .
5  [30] -> 70, 80, 90,  .
6  [70] ->  .
7  [80] -> 110, 120,  .
8  [110] ->  .
9  [120] ->  .
10 [90] ->  .
11 [40] -> 100,  .
12 [100] ->  .
13
```





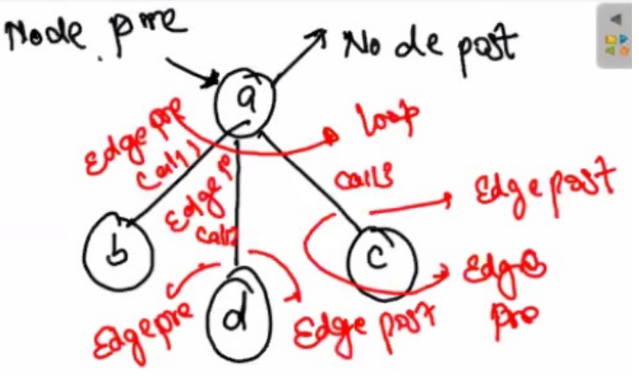

```java
public static int size(Node node) {
    // write your code here

    int s = 0;
    for (Node child : node.children) {
        int c = size(child);
        s = s + c; // 3 no child ka data add kardunga
    }
    s = s + 1; // usme ek vo khud add kardo
    return s;
}
```

Pre · Area → "Node pre" + node.data

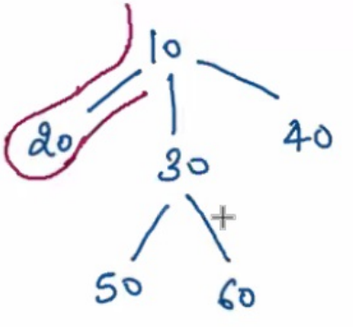
Before control leave → "Edge pre" + node.data -- child data

control back → "Edge post" + node.data -- child data

Post Area → "Node post" + node.data



node pre 10
Edge pre 10 -- 20
node pre 20
node post 20

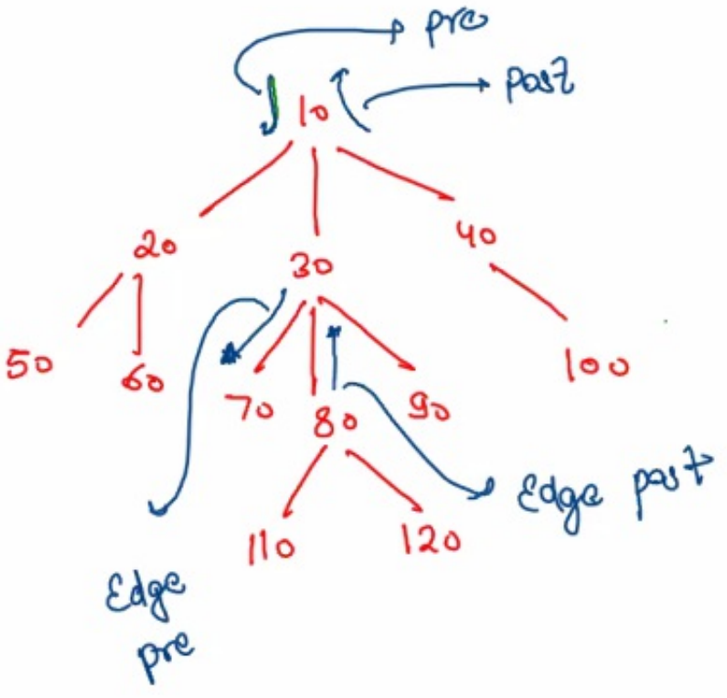


pre → just reach at level

Post → Before leaving current leaving

Edge-pre → make making a call inside loop

Edge post → After Making a call




```
node pre
    for all edges
        egde pre
        call
        edge post
node post
```

```java
public static void traversals(Node node) {
    // write your code here
    System.out.println("Node Pre " + node.data);
    for (int i = 0; i < node.children.size(); i++) {
        Node child = node.children.get(i);
        System.out.println("Edge Pre " + node.data + "--" + child.data);
        traversals(child);
        System.out.println("Edge Post " + node.data + "--" + child.data);
    }
    System.out.println("Node Post " + node.data);
}
```

Level Order of Generic Tree

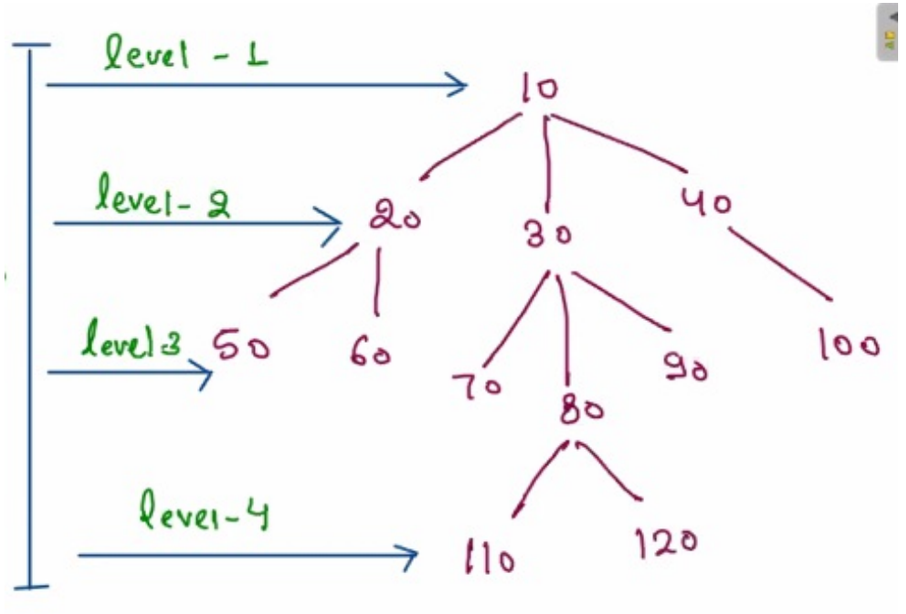
O/P → 10 20 30 40 50 60 70 80 90 100 110 120

Algo → Data Structure - Queue *Initially Queue have root

Steps - ① Remove

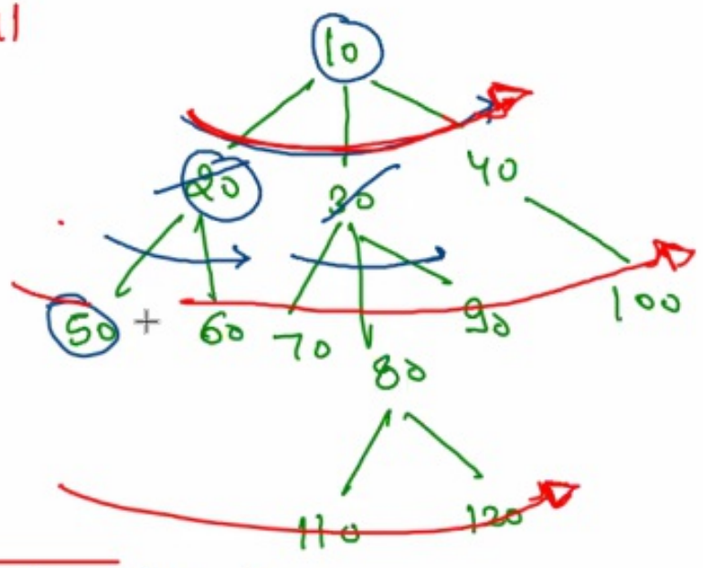
② print

③ Add children



FIFO - Queue

Radially traversal



10 20 30 40 50 60 70 80 90

100 110 120

Queue

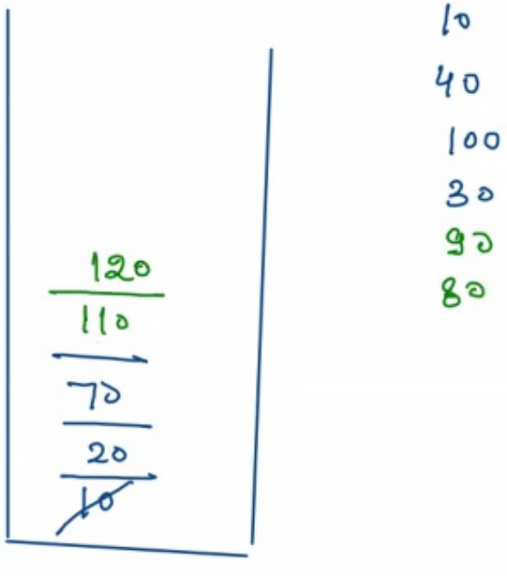
Queue → preference to siblings over children

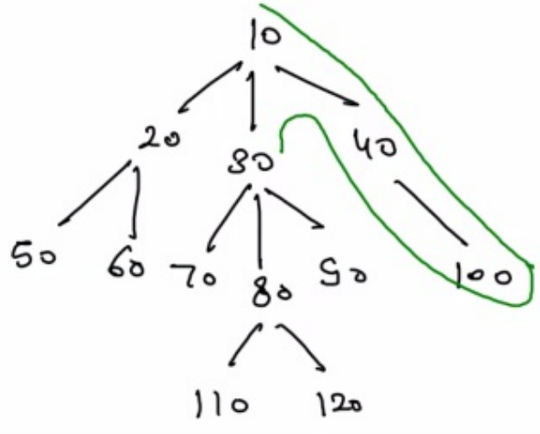
Steps:

1. Remove { Get + Remove
2. print
3. Add children

Note :if you use **stack** then there will be depth traversal from right to left and we are tring to **achieving recursion in iterative way**

ans in **recursion**: -itis also depth traversal from left to right






```java
// depth traversal using stack
public static void depthOrder(Node root) {

    Stack<Node> st = new Stack<>();

    st.push(root);
    while (st.size() > 0) {
        // RPA
        Node rem = st.pop();
        System.out.print(rem.data + " ");
        for (Node child : rem.children) {// traversal will be from right to left
            st.push(child);
        }
    }
}
```

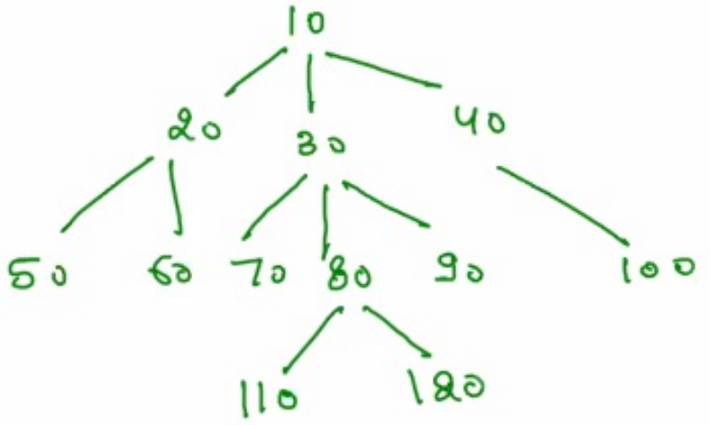
level order. line wise

Level-1 → 10

Level-2 → 20 30 40

Level-3 → 50 60 70 80 90 100

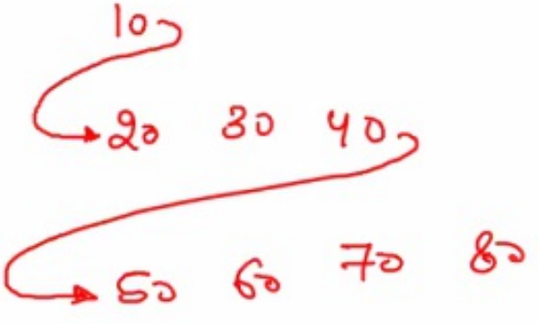
Level-4 → 110 120



Approach-1 → level order with 2 Queues

main Queue | ~~50~~ | ~~60~~ | ~~70~~ | ~~80~~ | ~~90~~ | 100 |

child Queue | 110 | 120 |



```java
// approach 1 - using 2 queue
public static void levelOrderLinewise(Node node) {
    // write your code here
    // you need to put extra line in levelorder traversal

    // approach1
    Queue<Node> mainQ = new ArrayDeque<>();
    Queue<Node> childQ = new ArrayDeque<>();

    mainQ.add(node);
    while (mainQ.size() > 0) {

        // RPA
        Node rem = mainQ.remove();
        System.out.print(rem.data + " ");
        childQ.addAll(rem.children);

        if (mainQ.isEmpty()) {// empty means level completed
            // hit enter
            System.out.println();

            // swap main and child
            Queue<Node> temp = mainQ;
            mainQ = childQ;
            childQ = temp;

        }
    }

}
```

approach 1

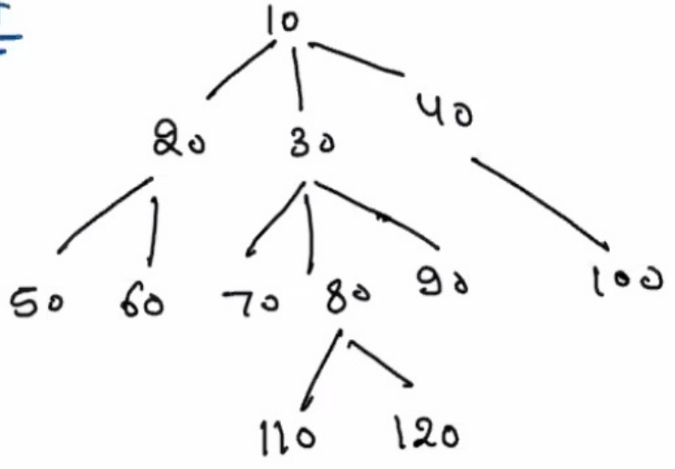
using 2 queue

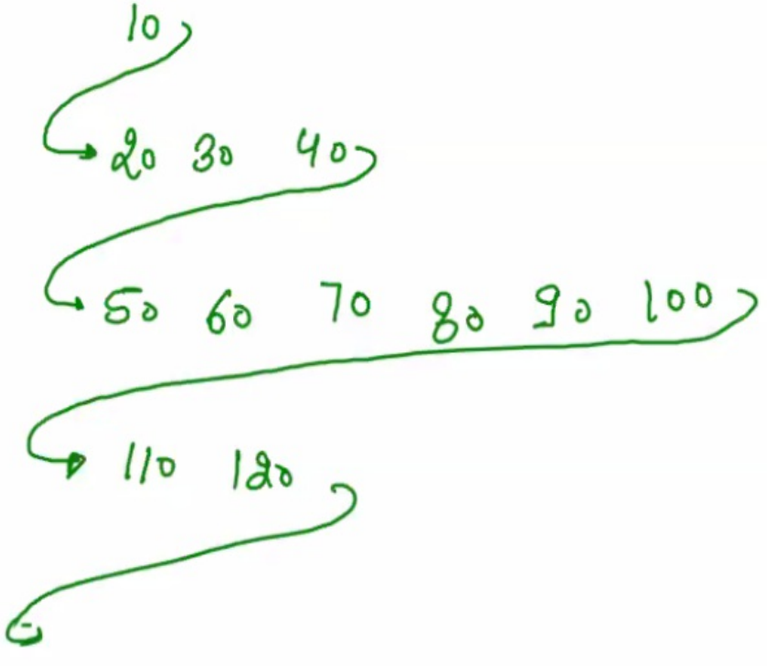
when \mainQ is empty l, we make sure 2 things
1. previous level is completed and
2.all the children of next level is added

Approach-2. Delimiter using single Queue ☆

Noted → Linked List

delimiter





→ null → Queue.size > 0

Steps: ① Get + Remove

② conditional → null → enter lit

conditional Addition of null

Size > 0

→ print

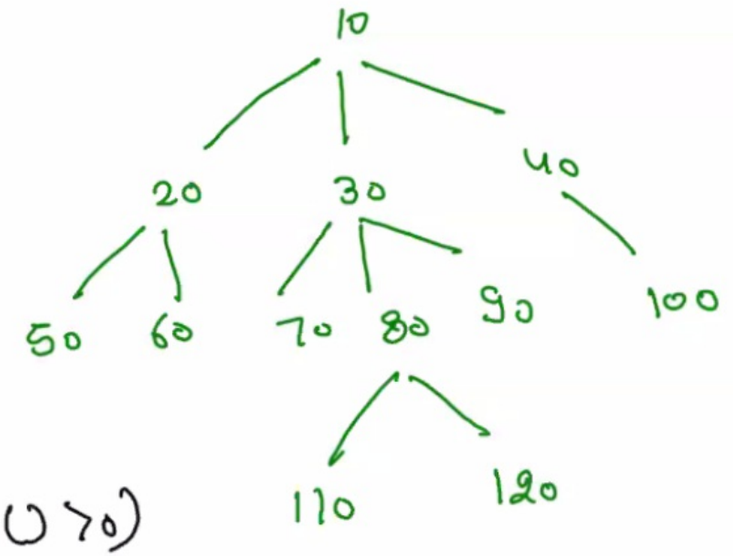
③ Add children

when you encounter nulll, we make sure 2 things
1. previous level is completed and
2.all the children of next level is added

approach 2: using delimiter

```java
// approach 2 using delimiter using single queue
public static void levelOrderLinewiseDelimiter(Node node) {

    // using linkedlist as queue
    // because arrayDequeue does not allow usto add null
    Queue<Node> qu = new LinkedList<>();

    qu.add(node);
    qu.add(null);
    while (qu.size() > 0) {

        // remove
        Node rem = qu.remove();
        if (rem == null) { // if delimiter encountered
            System.out.println();
            if (qu.size() > 0)
                qu.add(null);// only if qu size>0 else it will go to infinite
        } else {
            // print
            System.out.print(rem.data + " ");
            // add children
            qu.addAll(rem.children);
        }

    }

}
```

level order line wise - 3



size: ~~4~~ ~~3~~ ~~2~~ 2

10

20 30 40

50 60 70 80 90 100

110 120

while(qu. size() > 0)

size → of queue

Iterate size time on queue.

① get + remove

② print

③ children add

}

// level end . Enter hit

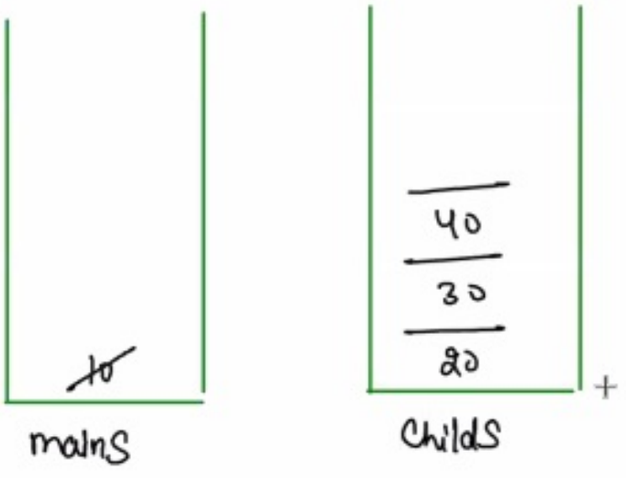
approach 3 : using single queue - - maintianing size

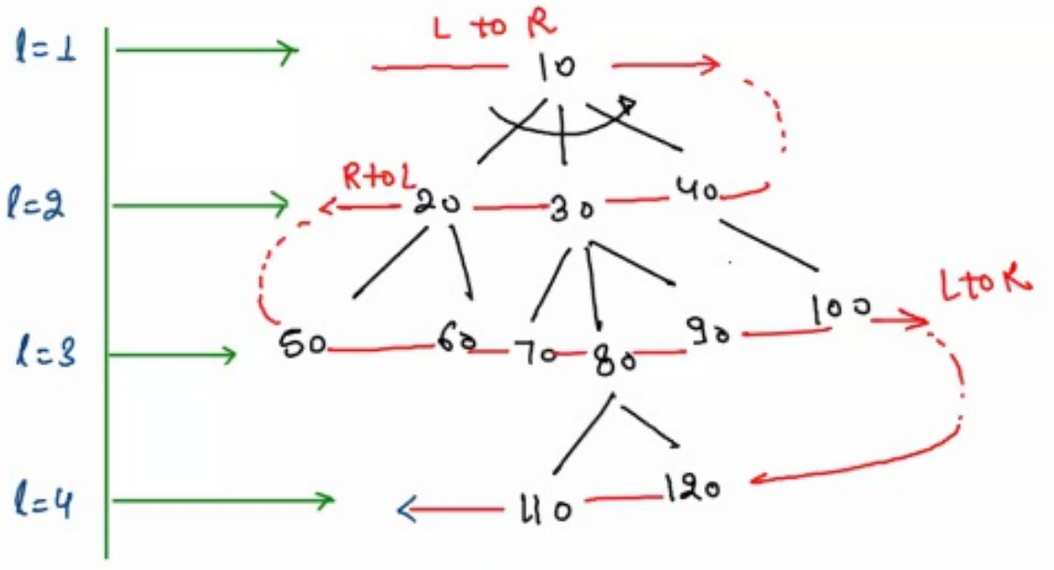
```java
// approach 3 using size of queue approach
public static void levelOrderLinewiseQueueSize(Node node) {

    Queue<Node> q = new ArrayDeque<>();
    q.add(node);
    int height = 0;
    while (q.size() > 0) {

        // find size
        int sz = q.size();
        while (sz-- > 0) {
            // RPA
            Node rem = q.remove();
            System.out.print(rem.data + " ");
            q.addAll(rem.children);
        } // at the end of this which loop
          // we can ensure that level is completed
        height++; // can be used for getting height of tree
        // hit enter
        System.out.println();

    }
    System.out.println(height);
}
```

aur ek particular level pe jyada freedom and ocntrol he

can be used for getting height of tree

Level order zigzag





evel 1

1 → left to Right

2 → Right to left

10

40 30 20

50 60 70 80 90 100

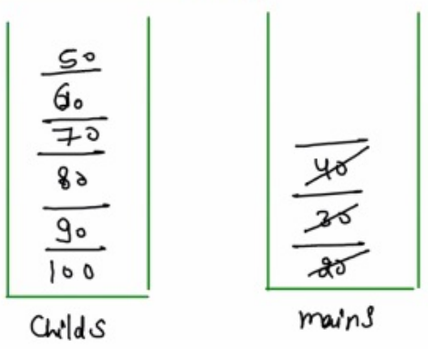
120 110

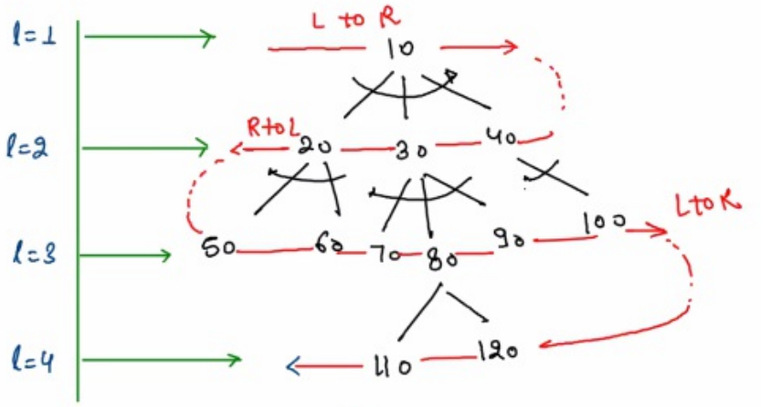
level - odd → left to Right

level - even → Right to left

JIS SIDE SE iterate kar rhe ho use side se child stack me add karna he

Level order zigzag

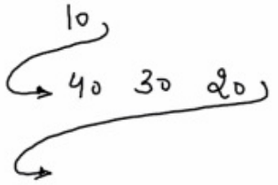




level 1 2

1 → left to Right

2 → Right to left

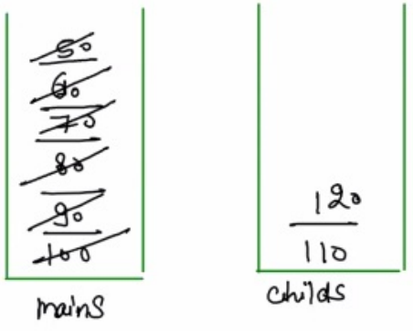


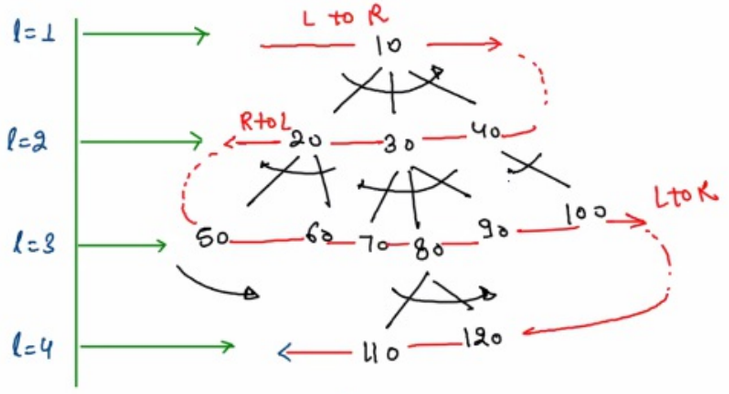
10
40 30 20
50 60 70 80 90 100
120 110

level - odd → left to Right

level - even → Right to left

Level order zigzag

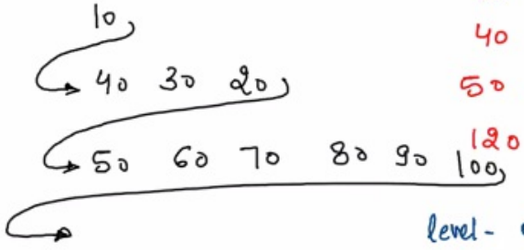




level 1 2 1 2

odd 1 → left to Right

even 2 → Right to left

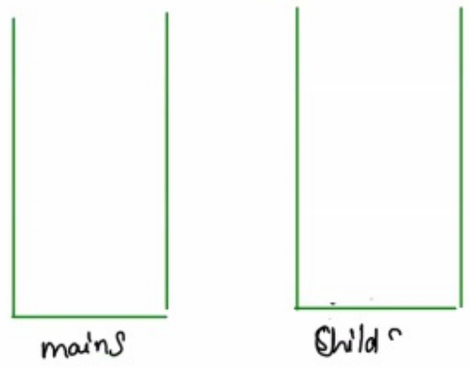


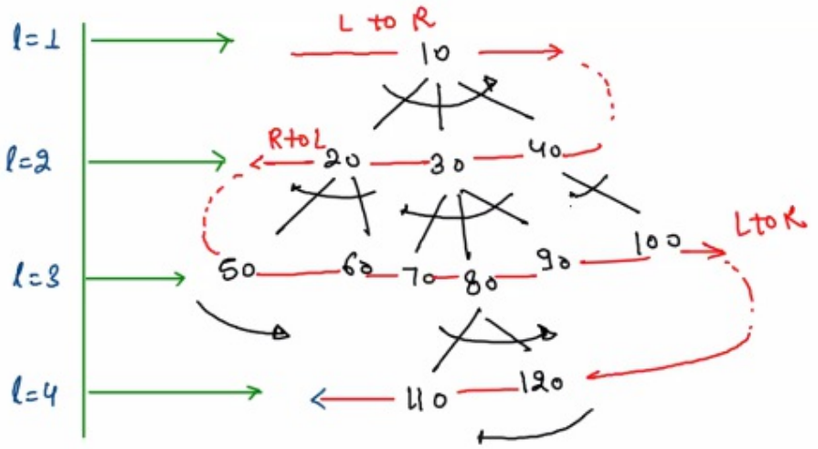
10
40 30 20
50 60 70 80 90 100
120 110

level - odd → left to Right

level - even → Right to left

Level Order Zigzag





level 1 2 1 2 1

odd 1 → left to Right

Even 2 → Right to left

10

40 30 20

50 60 70 80 90 100

120 110

10

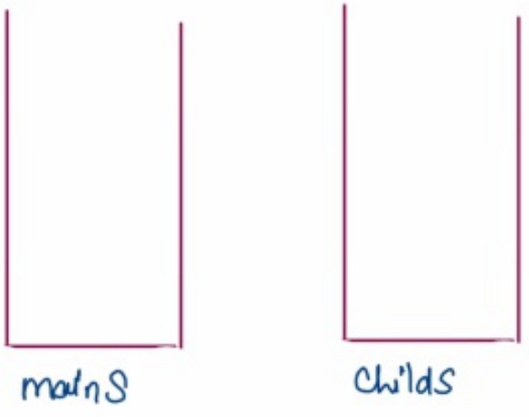
40 30 20

50 60 70 80 90 100

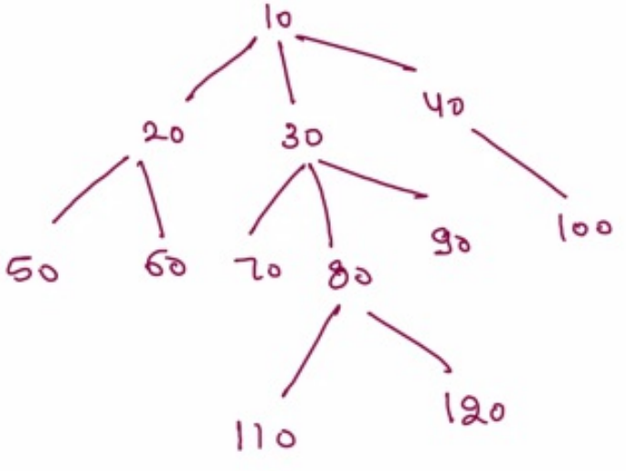
120 110

level - odd → left to Right

level - Even → Right to left

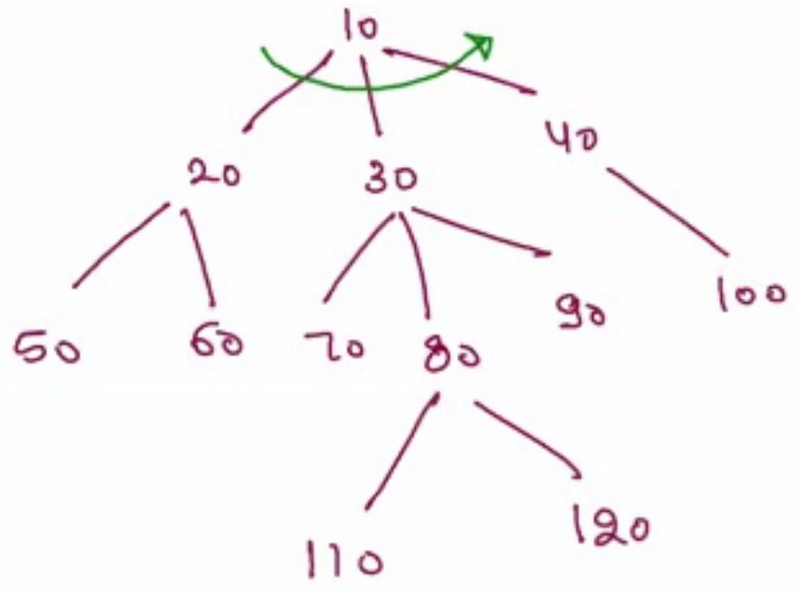


level is Helping for odd level and Even level



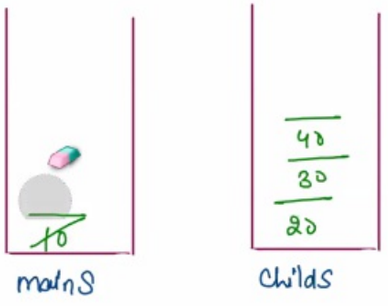
level = 1

level is Helping for odd level and Even level



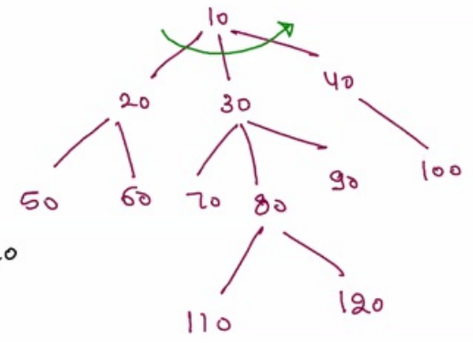
level - odd → addition of Child → left to Right

level - Even → addition of Child: Right to left



level = 1

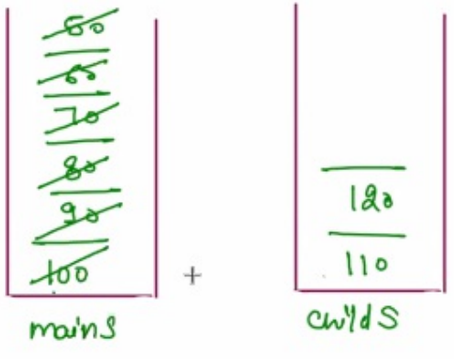
level is Helping for odd level and Even level



mains.size() == 0
level ++;
Hit Enter
swap mains and childs

level - odd → addition of Child → left to Right

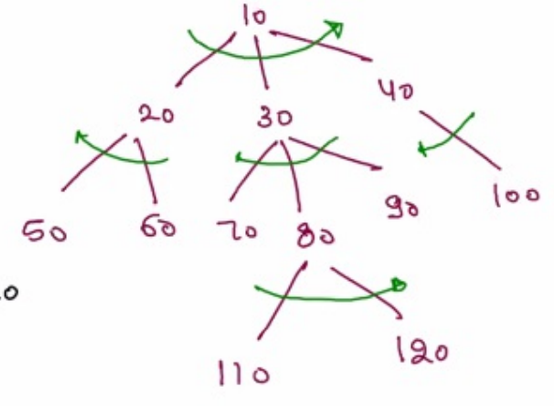
level - Even → addition of Child: Right to left



level = ~~1~~ 2 3

10
40 30 20
50 60 70 80 90 100

level is Helping for odd level and Even level



mains.size() == 0
level ++;
Hit Enter
swap mains and childs

level - odd → addition of Child → left to Right

level - Even → addition of Child: Right to left

think interms of having 2 queue and how it will behave

```java
public static void levelOrderLinewiseZZ2(Node node){
    Stack<Node> mainS = new Stack<>();
    Stack<Node> childS = new Stack<>();
    mainS.push(node);
    int lvl = 1;
    while(mainS.size() > 0) {
        Node rem = mainS.pop();
        System.out.print(rem.data + " ");

        if(lvl % 2 == 1) {
            // odd level -> left to right
            for(int i = 0; i < rem.children.size(); i++) {
                Node child = rem.children.get(i);
                childS.push(child);
            }
        } else {
            // even level -> right to left
            for(int i = rem.children.size() - 1; i >= 0; i--) {
                Node child = rem.children.get(i);
                childS.push(child);
            }
        }

        if(mainS.size() == 0) {
            System.out.println();
            lvl++;
            Stack<Node> temp = mainS;
            mainS = childS;
            childS = temp;
        }
    }
}
```

```java
// using double stack
public static void levelOrderLinewiseZZ(Node node){
    Stack<Node> mainS = new Stack<>();
    Stack<Node> childS = new Stack<>();
    mainS.push(node);
    int lvl = 1;
    while(mainS.size() > 0) {
        while(mainS.size() > 0) {
            Node rem = mainS.pop();
            System.out.print(rem.data + " ");

            if(lvl % 2 == 1) {
                // odd level -> left to right
                for(int i = 0; i < rem.children.size(); i++) {
                    Node child = rem.children.get(i);
                    childS.push(child);
                }
            } else {
                // even level -> right to left
                for(int i = rem.children.size() - 1; i >= 0; i--) {
                    Node child = rem.children.get(i);
                    childS.push(child);
                }
            }
        }
        System.out.println();
        lvl++;
        Stack<Node> temp = mainS;
        mainS = childS;
        childS = temp;
    }
```

more control on level