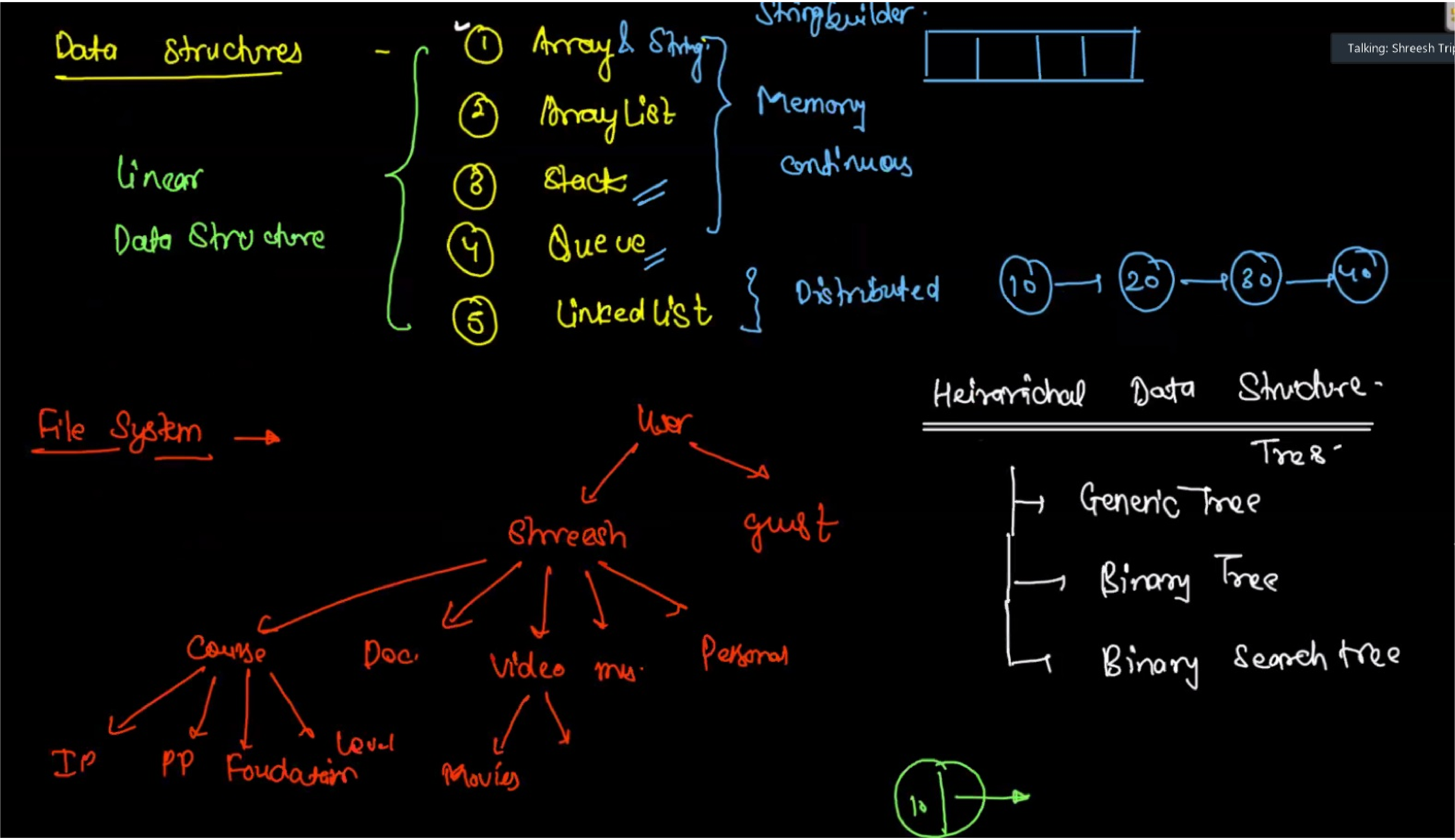


Data Structures -

Linear Data Structure

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

StringBuilder.

Memory continuous

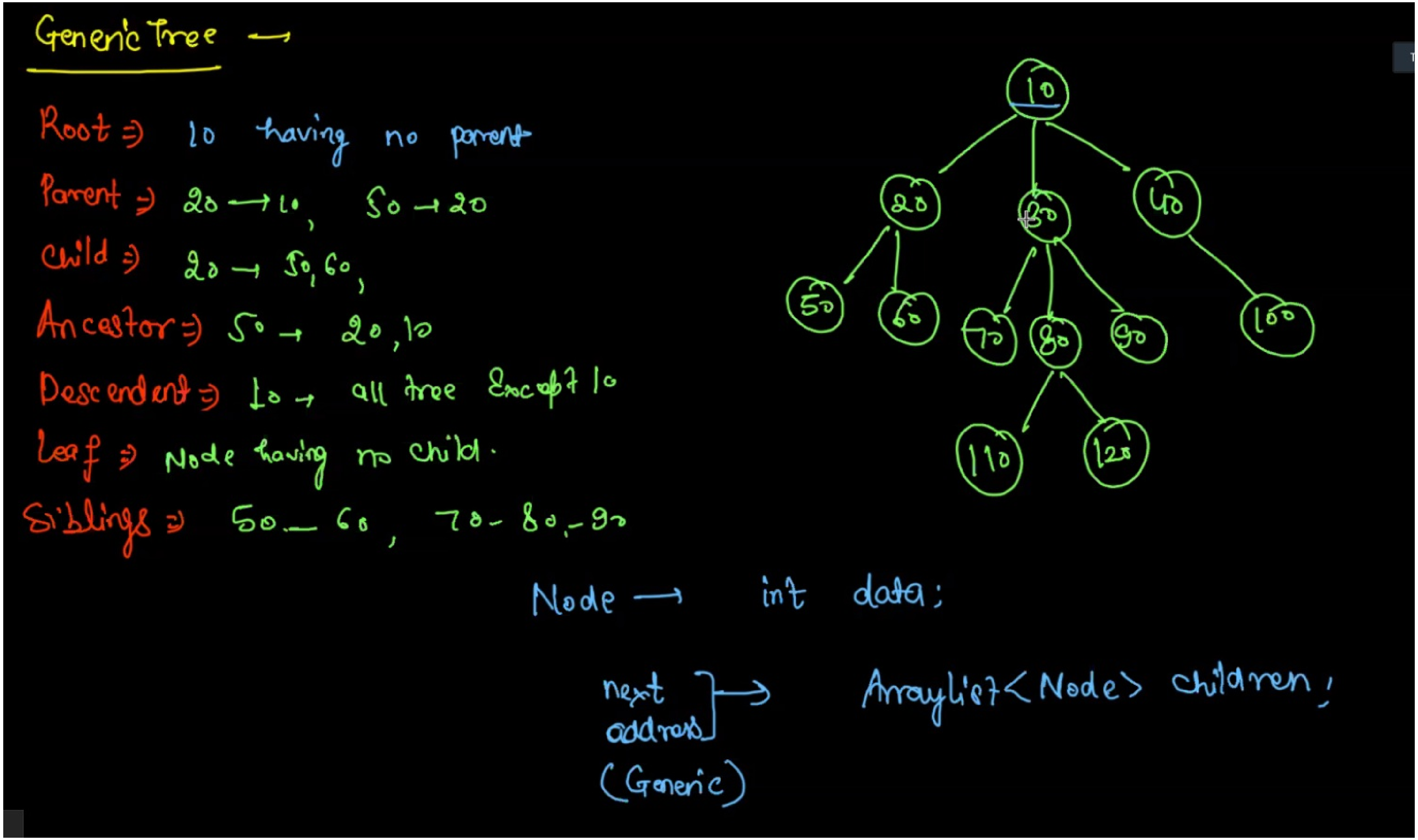
Distributed

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

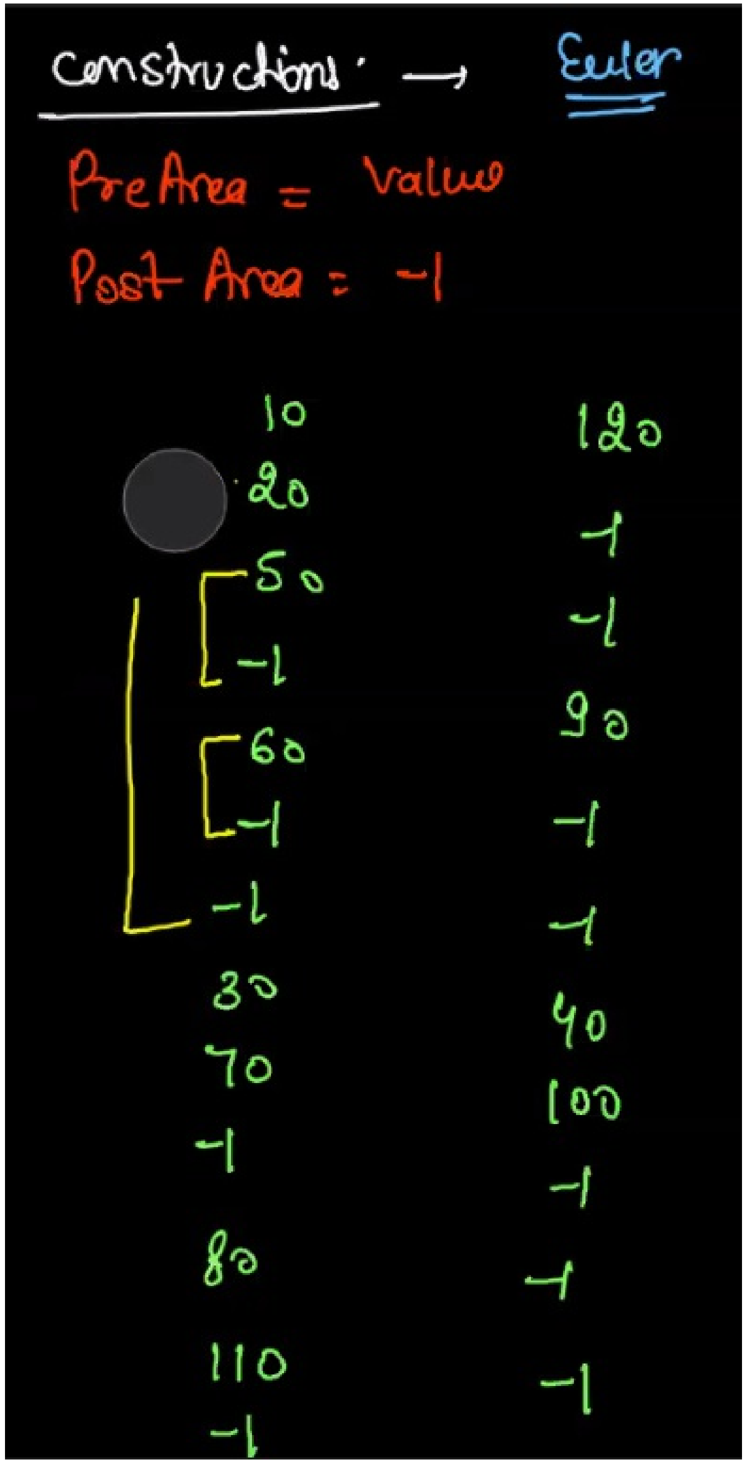
Descendent ⇒ 10 → all tree Except 10

Leaf ⇒ Node having no child.

Siblings ⇒ 50 — 60, 70 - 80 - 90

Node → int data;

next address (Generic)] → Arraylist<Node> children;



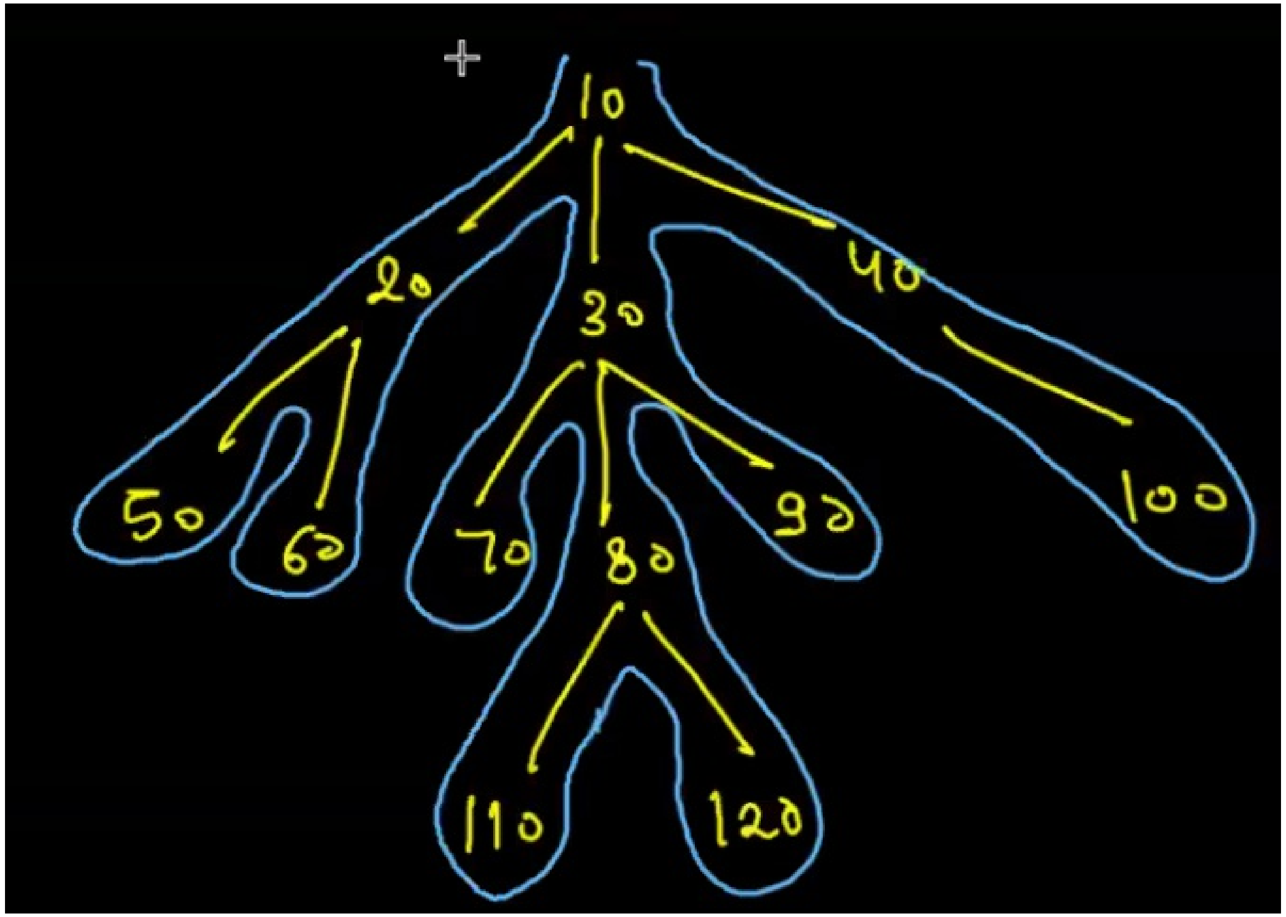
constructions: → Euler

PreArea = Value

Post Area = -1

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

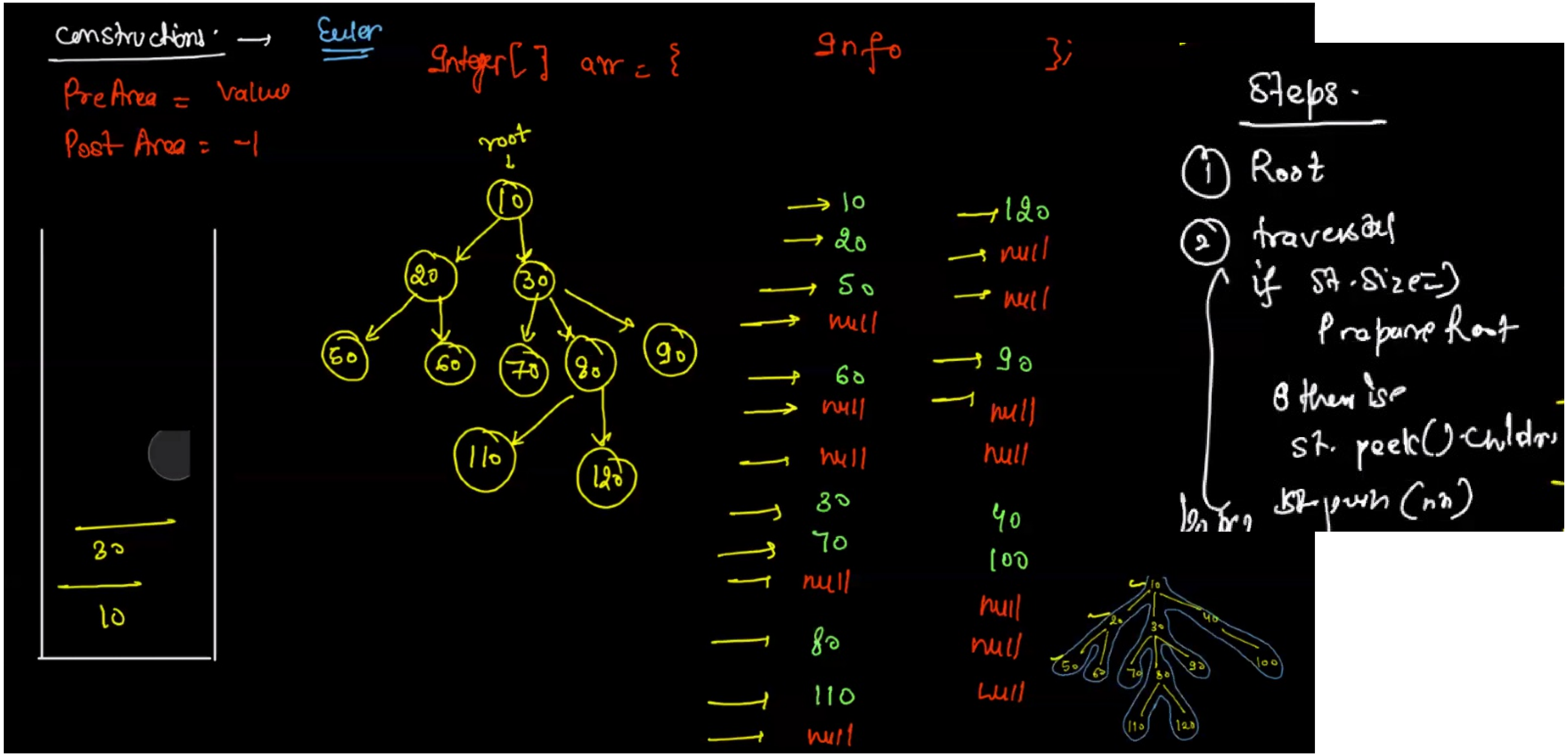
120
-1
-1
90
-1
-1
40
100
-1
-1
-1



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.....}



Construction: → Euler

PreArea = Value

Post Area = -1

Integer[] arr = { info };

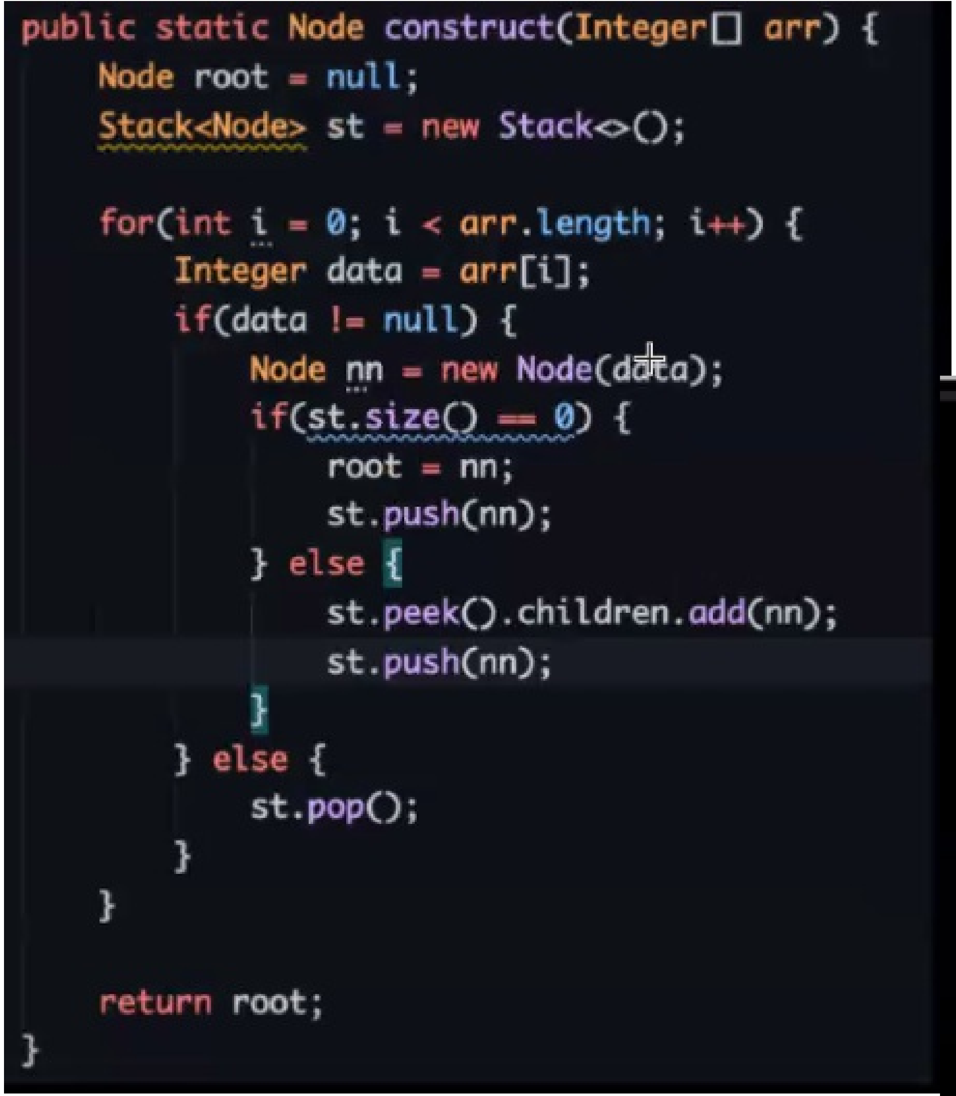
10, 20, 50, null, 60, null, null, 30, 70, null, 80, 110, null, 120, null, null, 90, null, null, 40, 100, null, null, null

Steps.

1. Root
2. traversal
   if st.size==)
   Prepare Root
   & then use
   st. peek().childr
   st.push(nn)

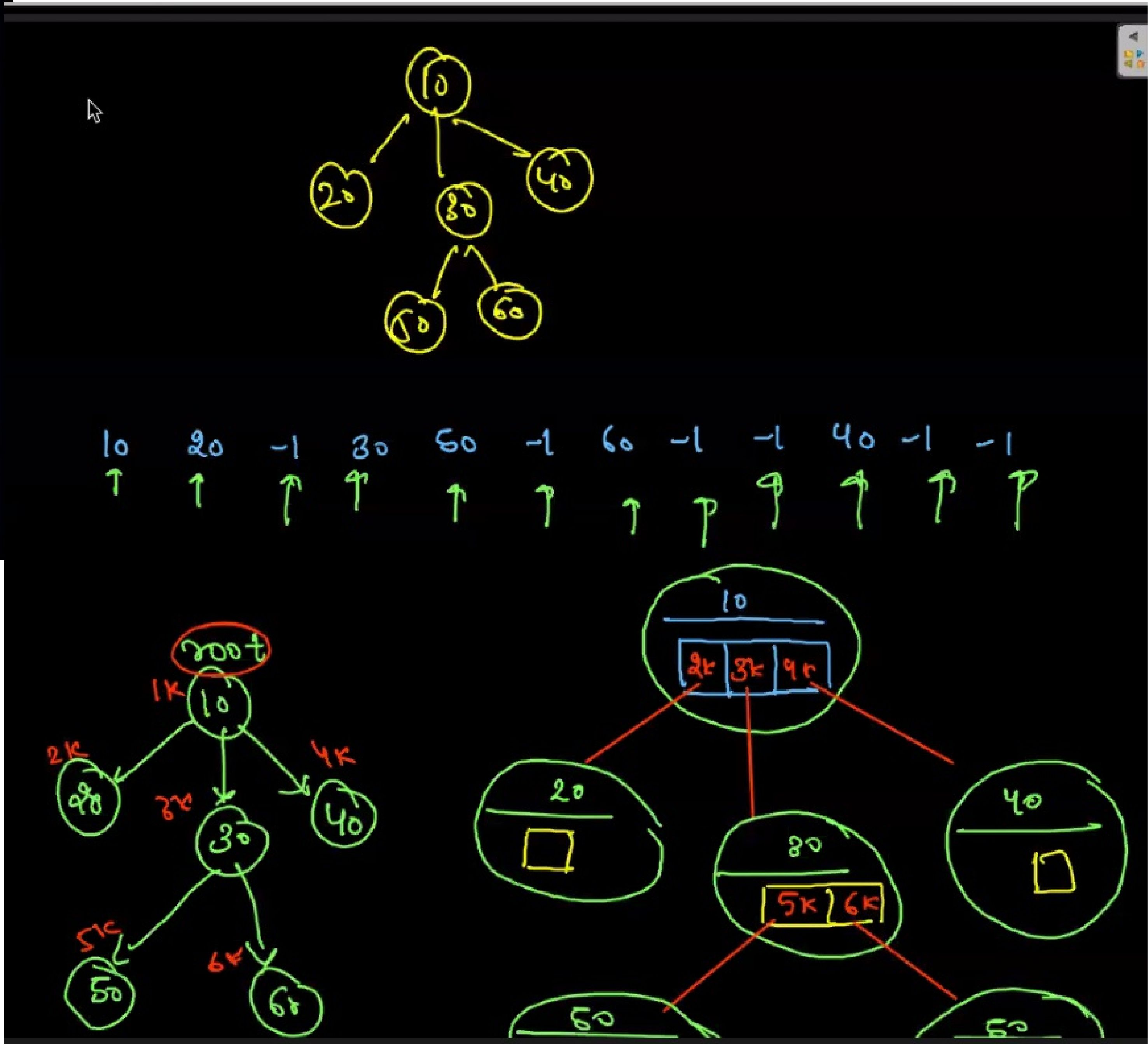
Approach:
1. create new node;
2. stack k top me jo data he uske child me add kardia aur lkhudko push kar dia

2.agar null aya toh pop kardo



```
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] → .

din

[60] → .

[30] → 70, 80, 90, .

[70] → .

[80] → 110, 120

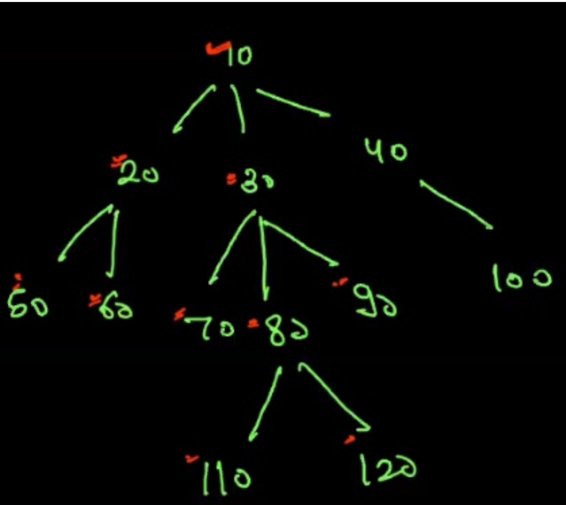
[110] → .

[120] → .

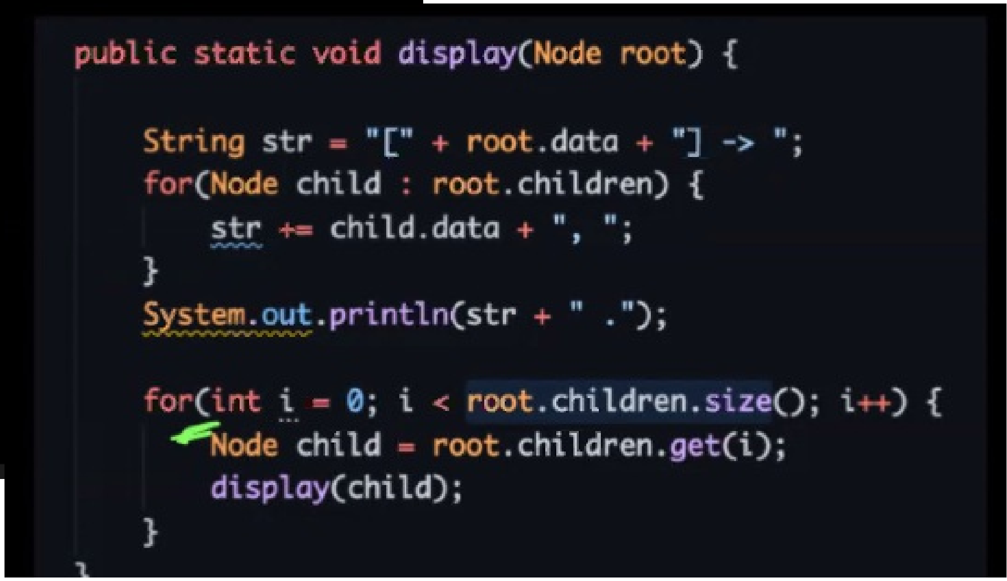
[90] → .

[40] → 100

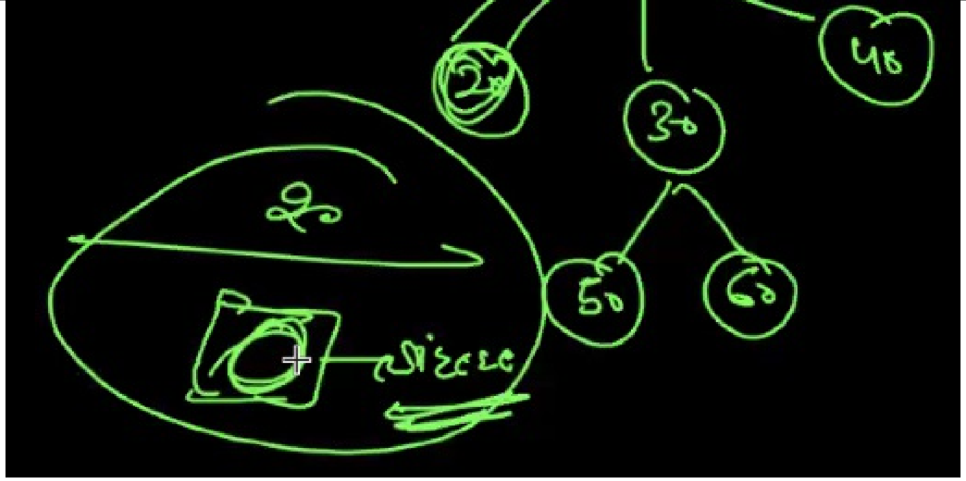
[100]



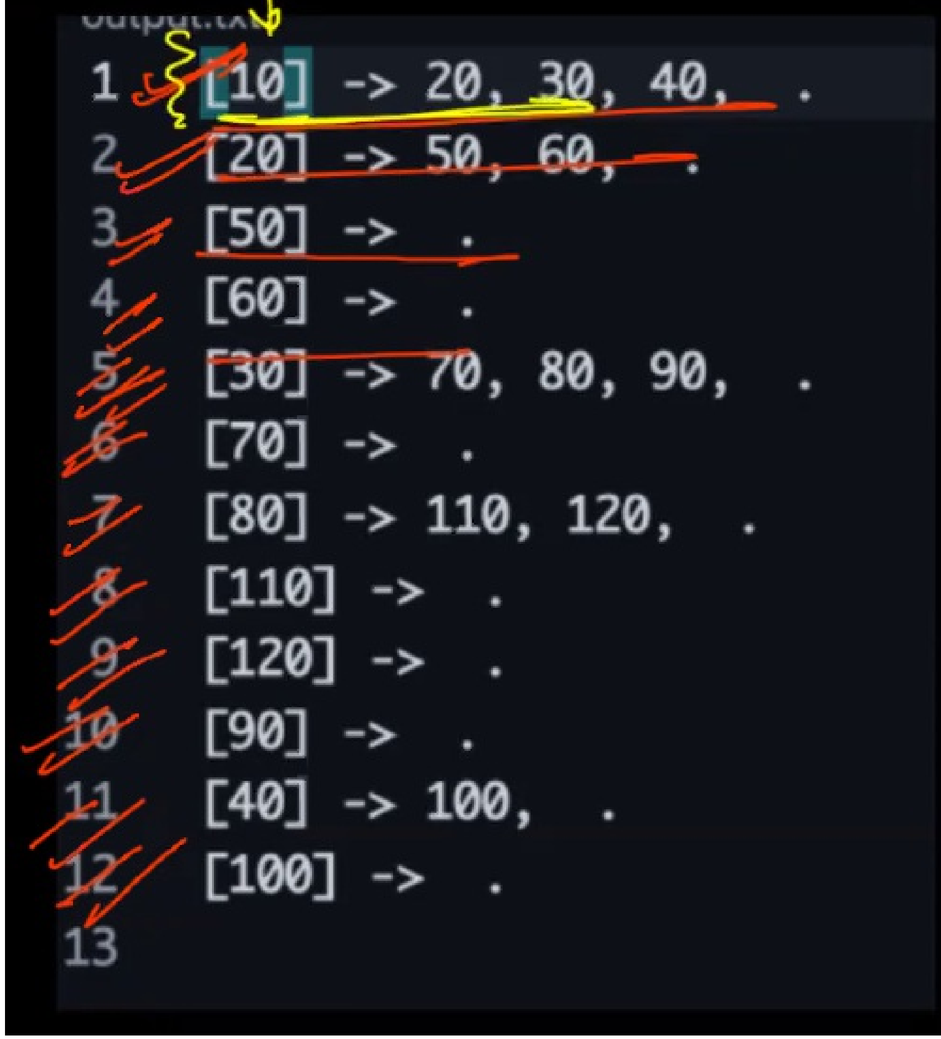
```
}
```



```
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);
    }
}
```

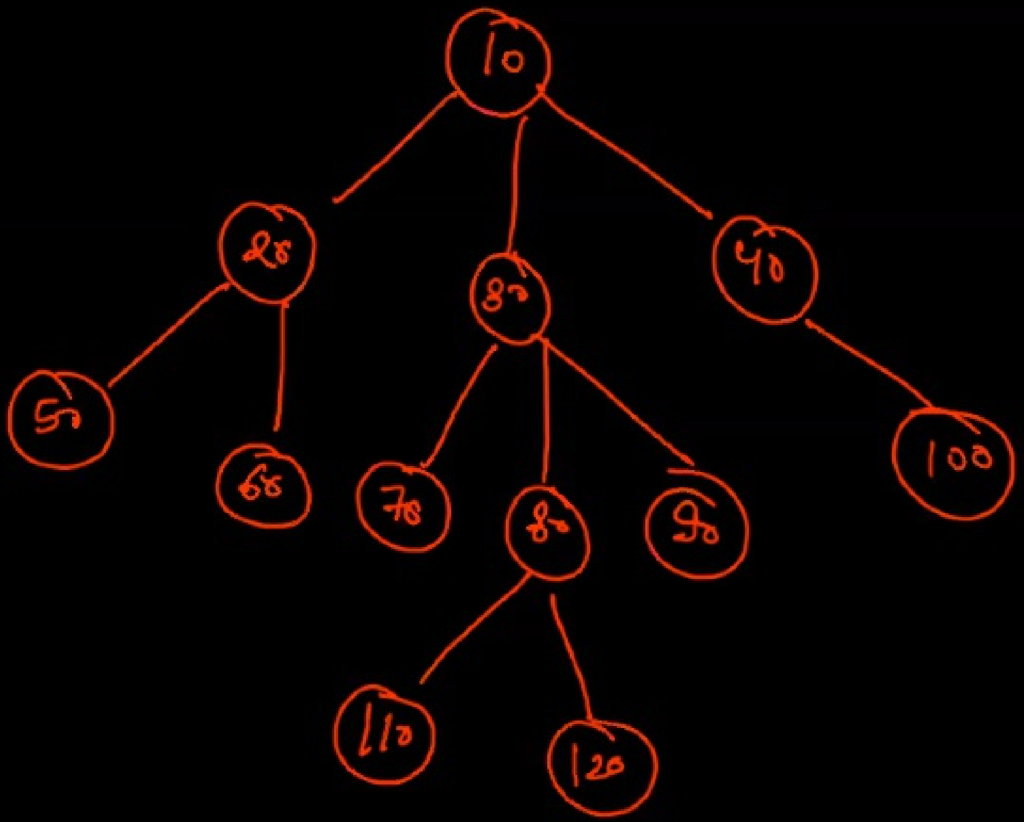


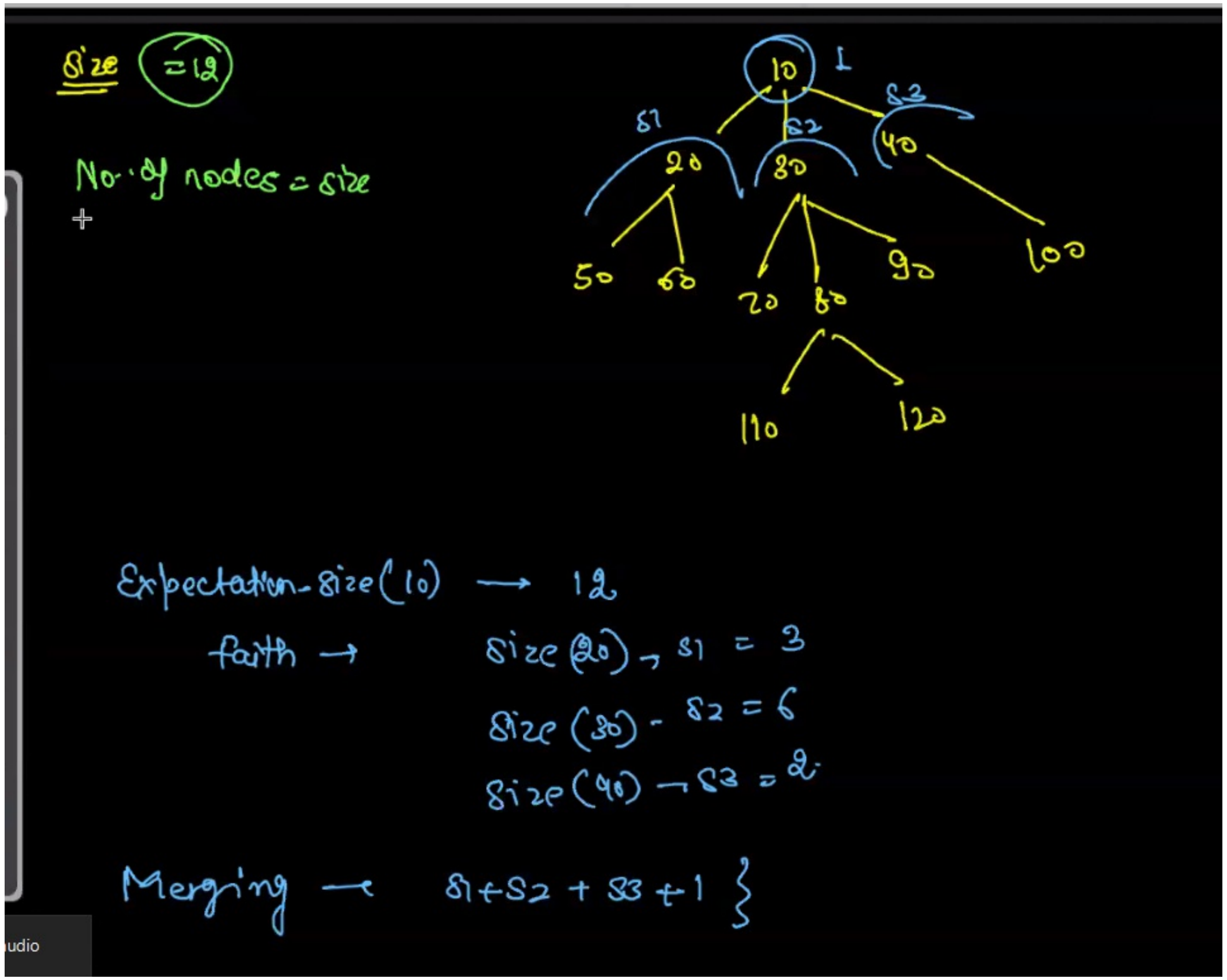
InOrder



output.txt

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





Size (=12)

No. of nodes = size

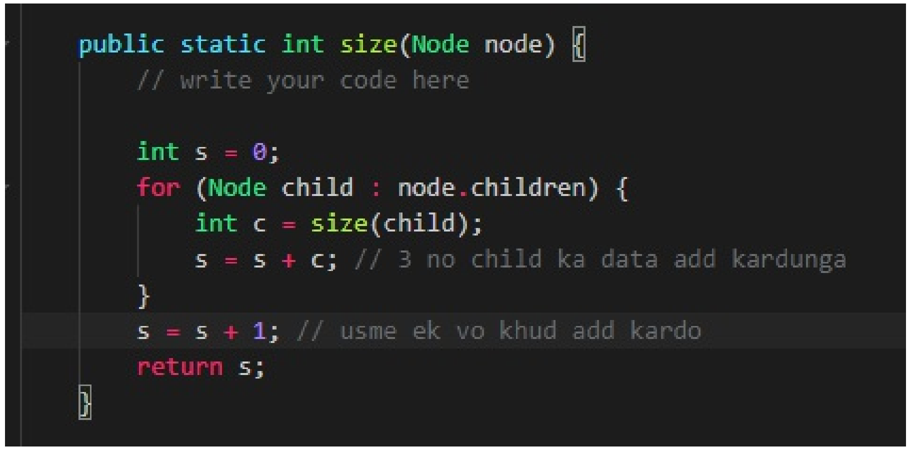
Expectation - size(10) ⟶ 12

faith → size(20) → s1 = 3

size (30) - s2 = 6

size(40) → s3 = 2

Merging → s1+s2 + s3 +1 }



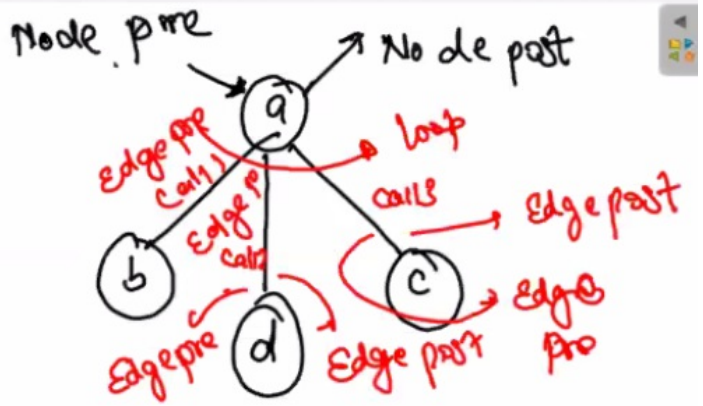
```java
public static int size(Node node) {
    // write your code here

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

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

Post Area → "Node post" + node.data

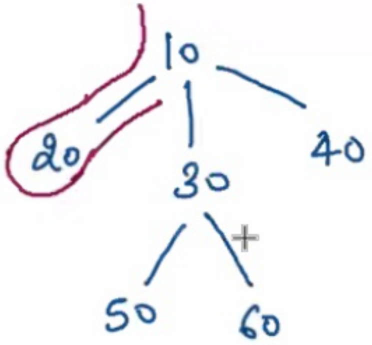


node pre 10

Edge pre 10 -- 20

node pre 20

node post 20

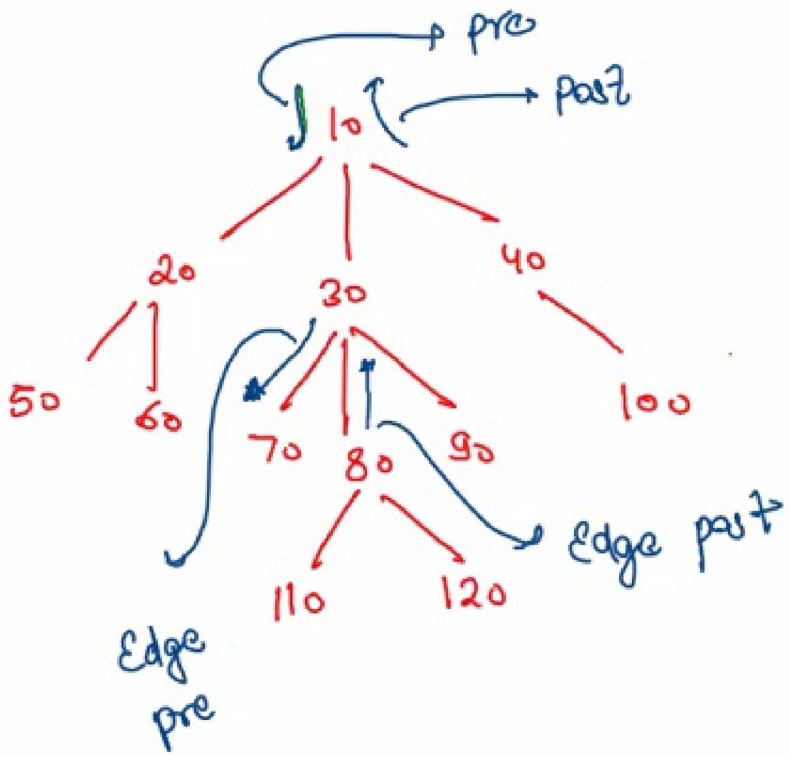


pre → just reach at level

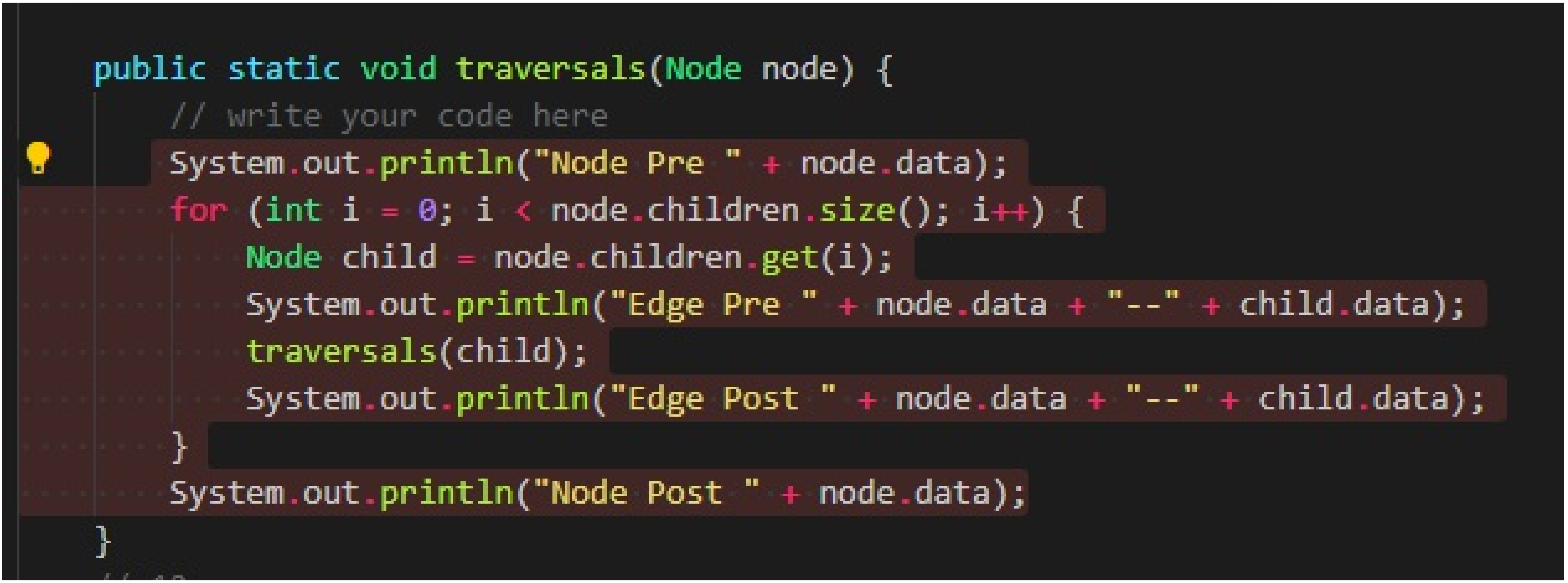
post → Before leaving current leaving

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
```



```
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

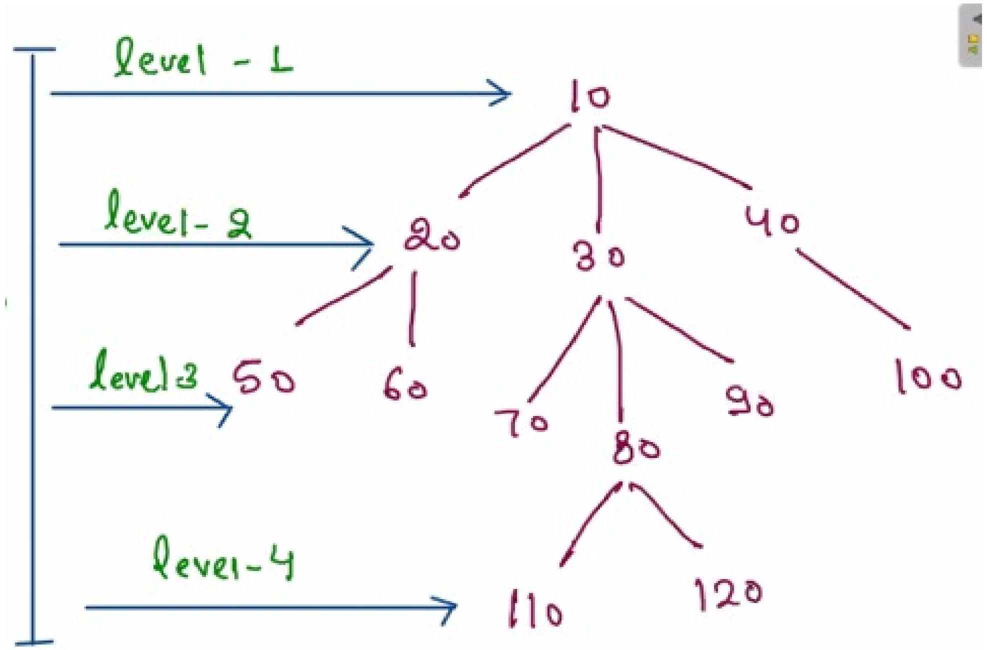
O/p → 10 20 30 40 50 60 70 80 90 100 110 120

Algo → Data Structure - Queue *Initially Queue have root

Steps - ① Remove

② print

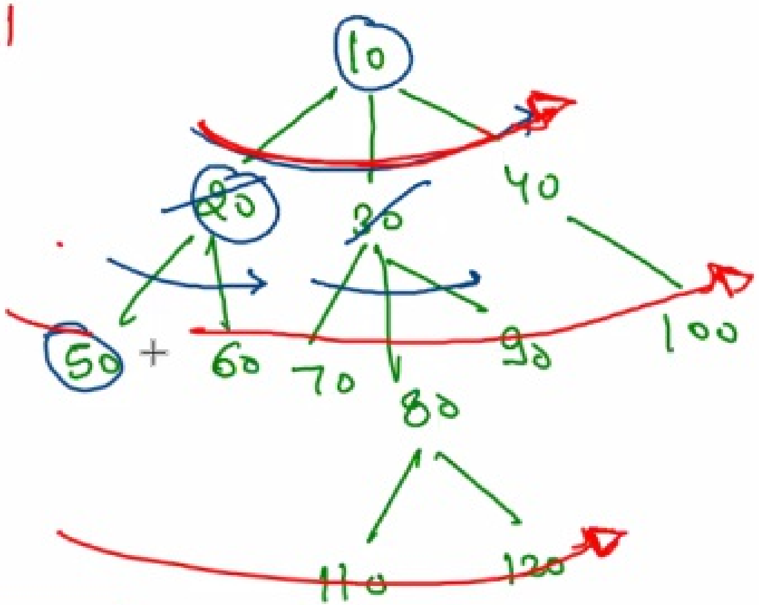
③ Add children



FIFO - Queue

Radially traversal

~~80~~ | ~~90~~ | ~~100~~ | ~~110~~ | ~~120~~



10 20 30 40 50 60 70 80 90

100 110 120

Quew ~~10~~ | ~~20~~ | ~~30~~ | ~~40~~ | ~~50~~ | ~~60~~ | ~~70~~ | ~~80~~ | 90 | 10 | 110 | 120

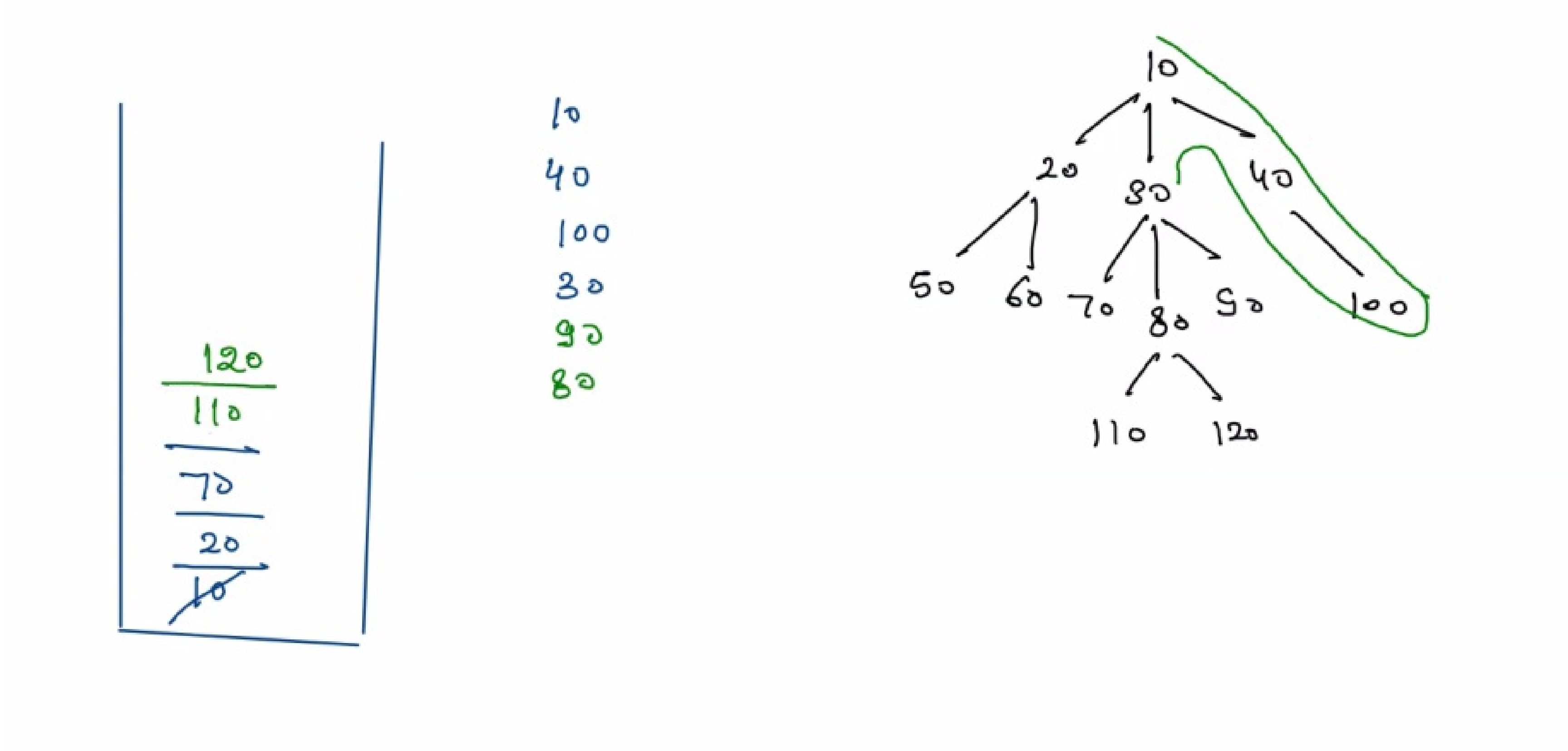
Queue → preference to siblings over children

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



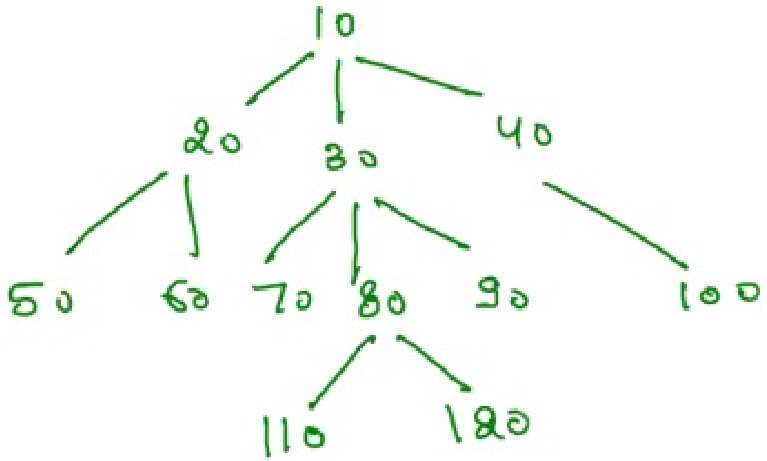
level order- line wise

level-1 → 10

level-2 → 20 30 40

level-3 → 50 60 70 80 90 100

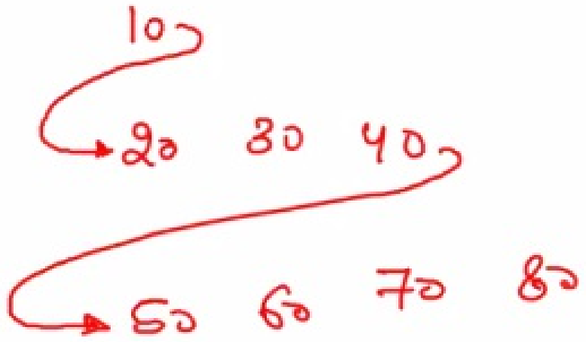
level-4 → 110 120

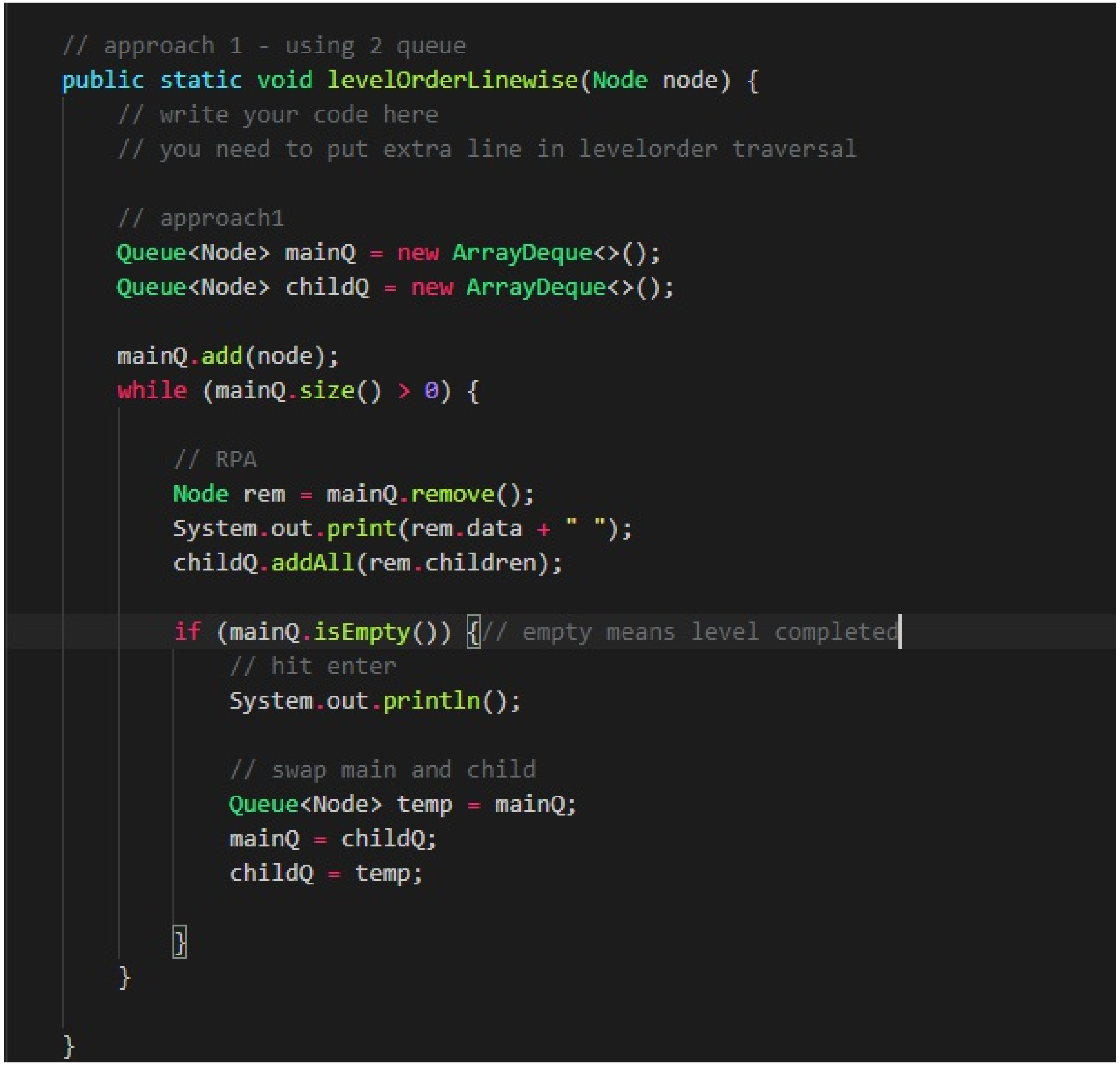


Approach-1 → level Order with 2 Queues

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

Child Queue 110 | 120



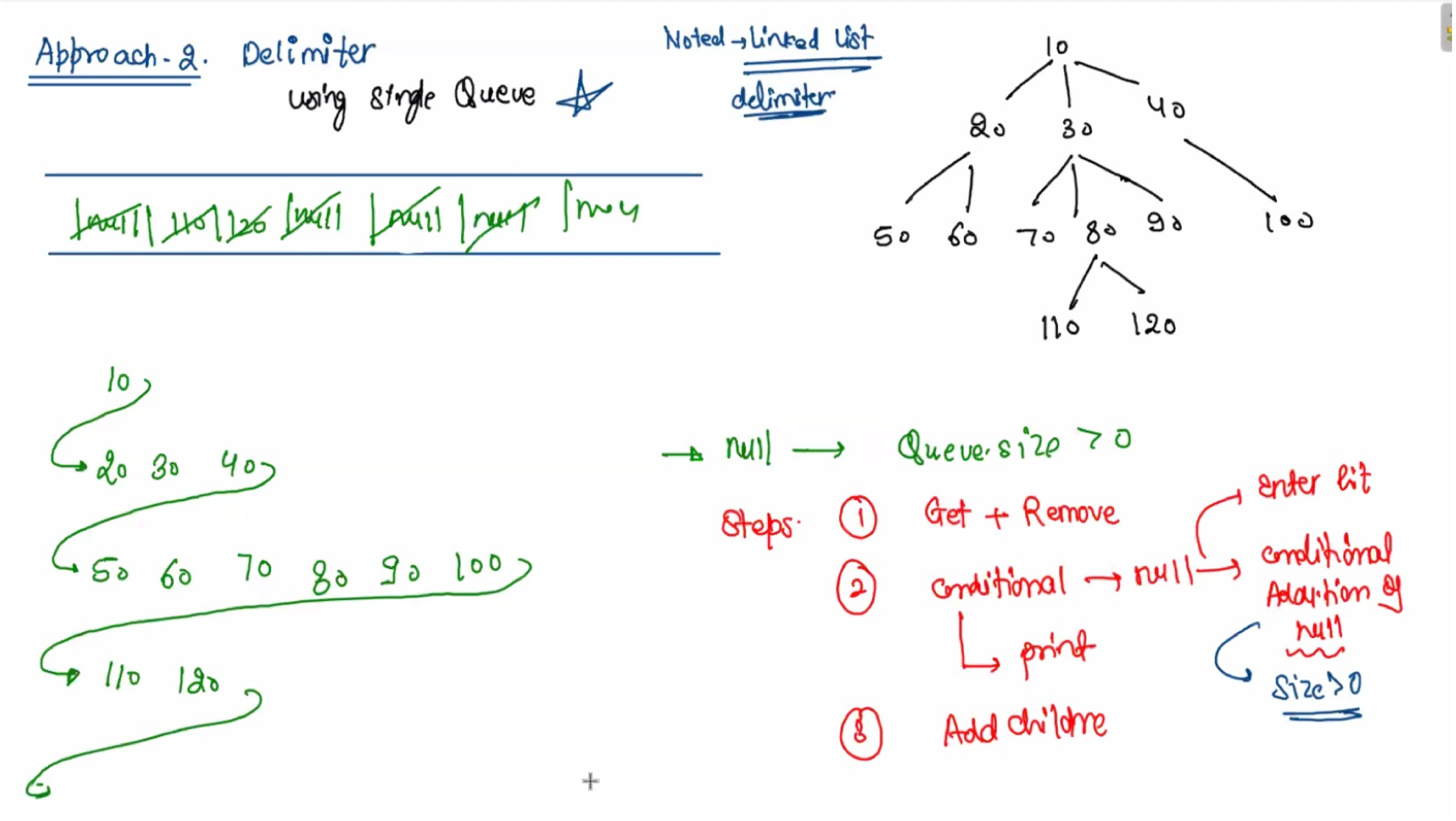


```java
// approach 1 - using 2 queue
public static void levelOrderLinewise(Node node) {
    // write your code here
    // you need to put extra line in levelorder traversal

    // 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 I,
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~~ | ~~110~~ | ~~120~~ ~~null~~ | ~~null~~ | ~~null~~ | null

→ null → Queue.size > 0

Steps. ① Get + Remove

② conditional → null → enter lit / conditional Addition of null

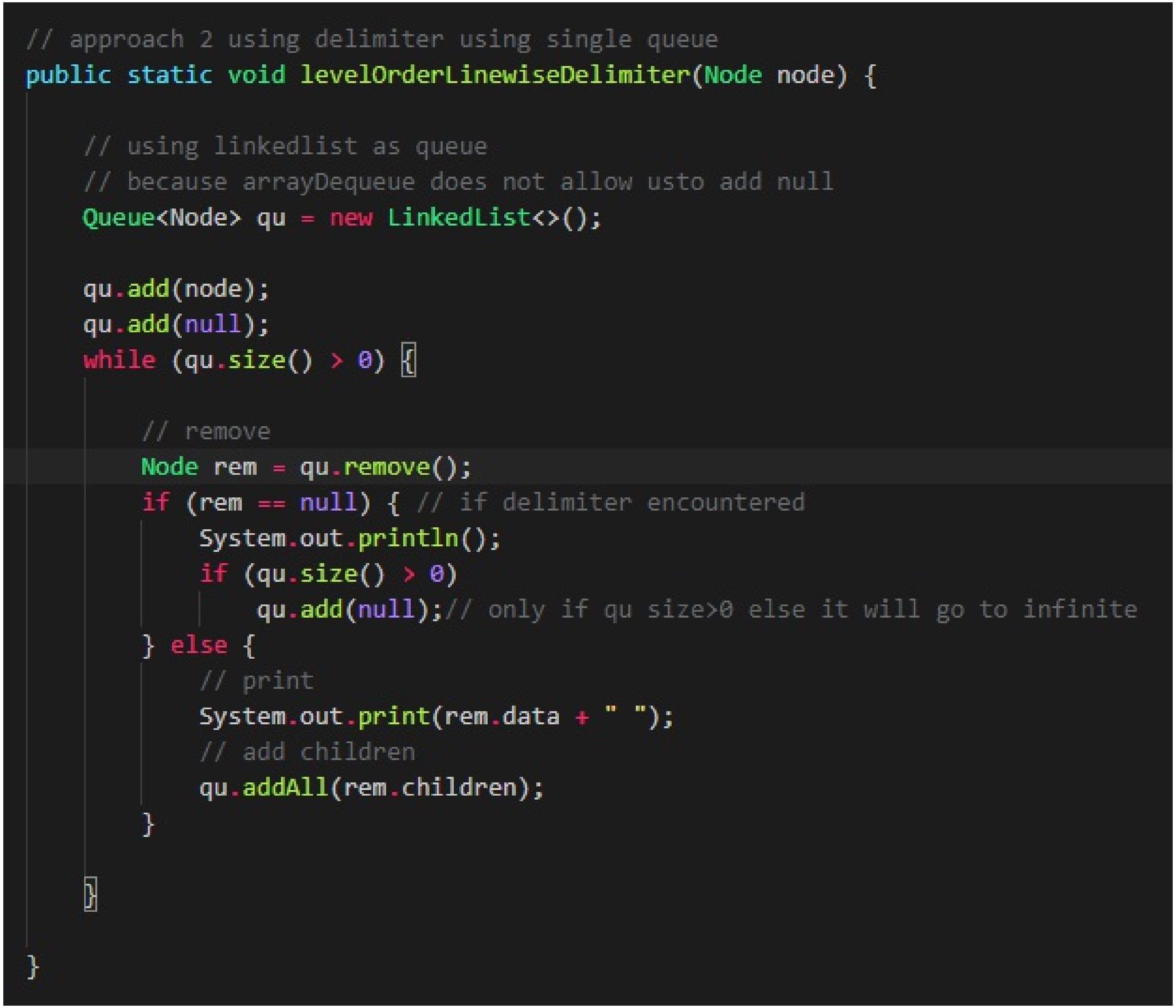
└→ print

size > 0

③ Add children

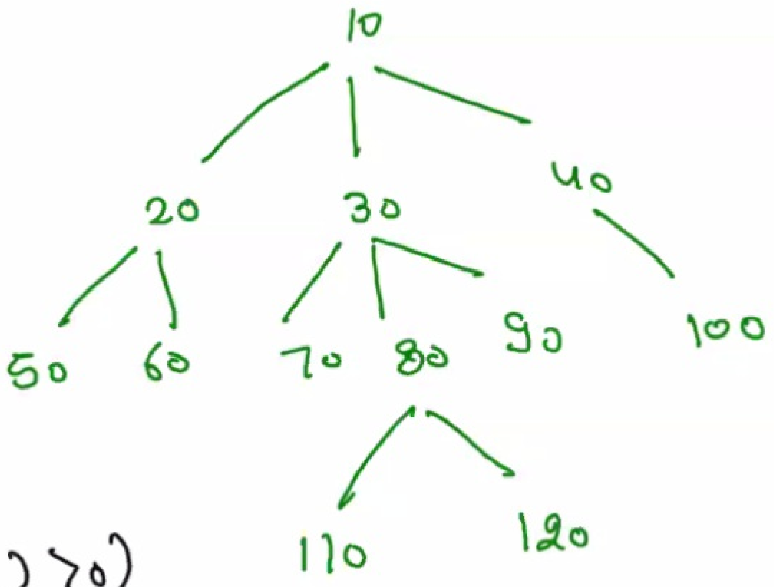
when you encounter nulll, we make sure 2 things
1. previous level is completed and
2.all the children of next level is added

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: ~~1~~ ~~3~~ ~~6~~ 2

10

20 30 40

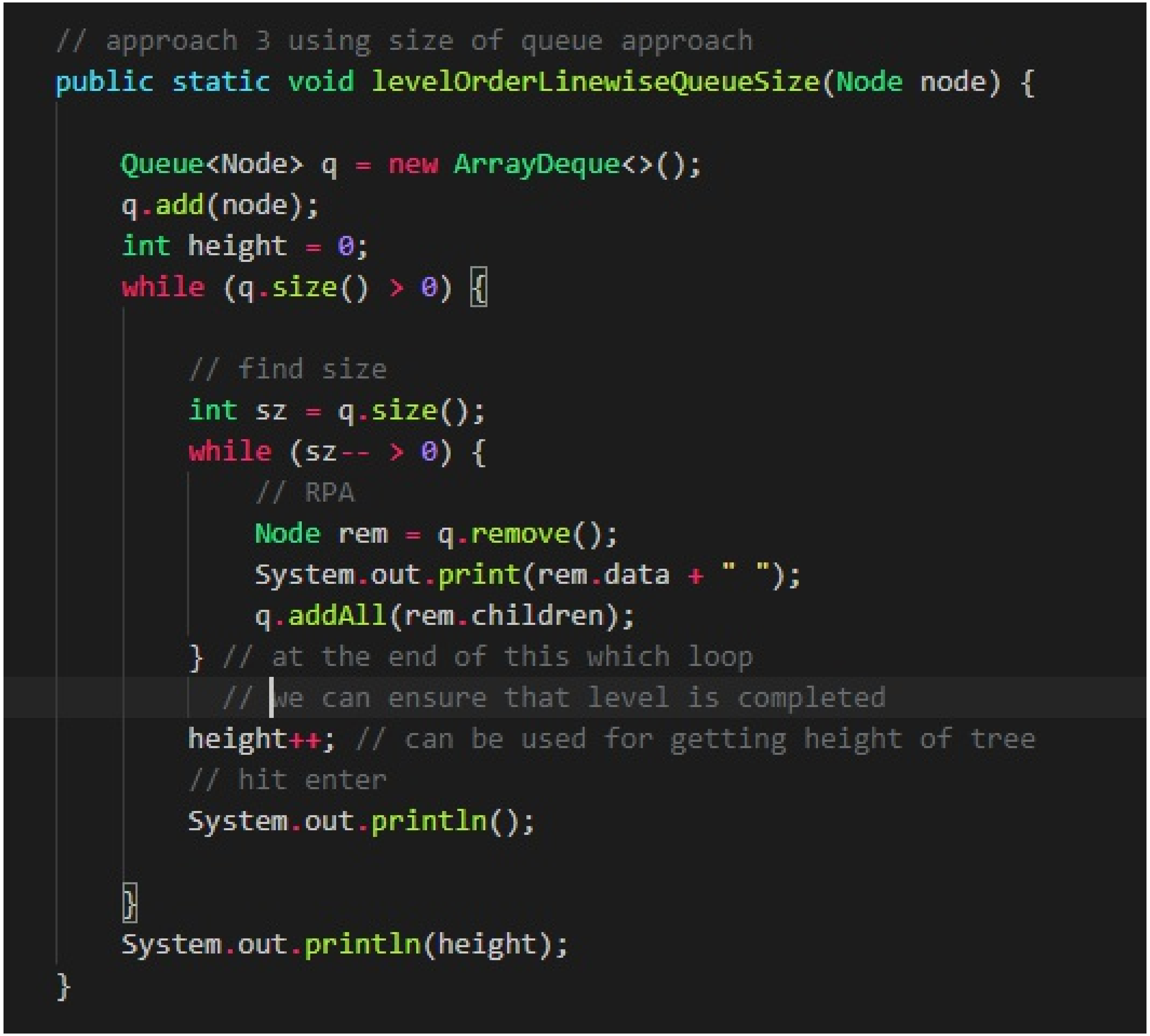
50 60 70 80 90 100

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

aur ek particular level pe jyada freedom and ocntrol he



```
// 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);
}
```

can be used for getting height of tree