DAY 127 DSA

HASHMAP AND HEAP CONTINUED

Date → 1 feb, 22
Day → Tuesday

Priority Queue एक class है जो Heap Data Structure का use करके बनाई गई है।

Date / /
Page No.

Q.1) PRIORITY QUEUE USING HEAP

* हमें अपनी खुद की PQ class बनानी है. → PriorityQueue.
which will have the following functions

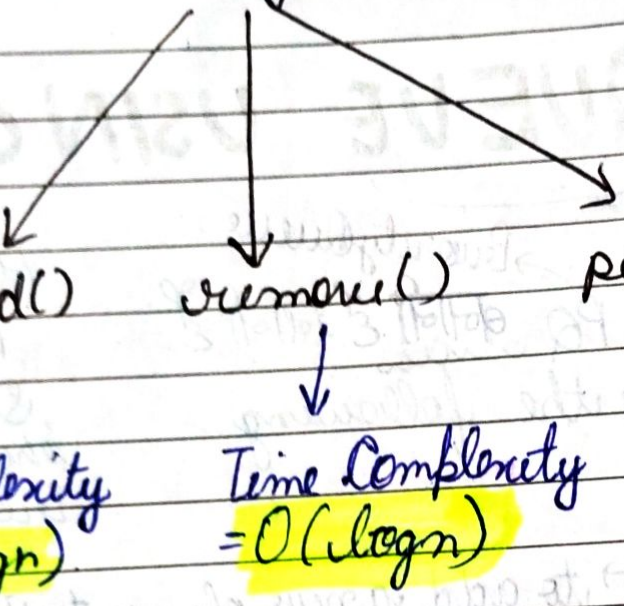
Heap is a Data Structure which is Binary Tree based.

↳ add() → to add a new element in the PQ.
remove()
peek() → should return the smallest value if available or print "Underflow" and return -1.
size() → Should return the no. of elements available in the PQ.

remove() → Should remove and return the smallest value if available (if the priority is given to smaller) Or print underflow (if there is no value) & return -1.
By default (डिफॉल्ट value तो higher priority की जाएगी).

Priority Queue को बनाने के हमारे पास 3 ways होते हैं।

Priority Queue के main 3 function होते हैं.
(हमें ये Time Complexity use करने वाली PQ बनानी है).



✗ Way 1 Priority Queue बनाने का पहला तरीका हो सकता है

USING A SORTED LINKED LIST

WORKS BUT THE TIME COMPLEXITY IS NOT GOOD.

add 10 → [10] (head & tail is 10)

add 20 → [10] → [20] (अब 20 है tail)

add 5 → [5] → [10] → [20] (head अब 5 है) (10 तक traverse करना पड़ा फिर 20 को add किया)

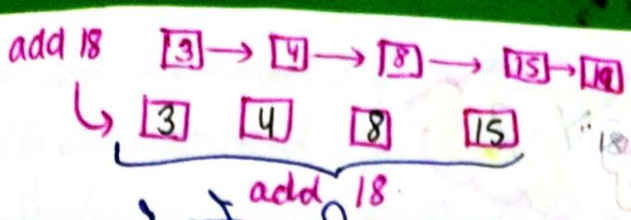
add 44 → [5] → [10] → [20] → [44]

20 तक traverse करना पड़ा फिर 44 को add किया

add 22 → [5] → [10] → [20] → [22] → [44]

20 तक traverse करना पड़ा फिर 22 को add किया

add()



तो add() function में हमें पूरी linked list पर travel करना पड़ रहा है and then we get our correct position to add the relevant item तो इस case में because we are travelling the linkedlist to find the correct position, add() function की time complexity हो जाएगी O(n).

peek()

peek() function gives us the smallest element by default and because our linked list is a sorted linked list तो head of the linkedlist will be the smallest element in the linkedlist.
So, basically the time complexity taken to get the head of the linkedlist is O(1).

remove()

remove() function removes the smallest element & return the smallest element. So, similar to peek() function, the head of linked list is the smallest element in the linkedlist. So, the Time Complexity taken to remove the head of the linkedlist is O(1).

(1-n) n } n time loop चलाना, array के सभी element को add कराने के लिए → n(n)=n² हमें add, remove → O(log n) में चाहिए और peek O(1) में.

USING SORTED LINKED LIST

add()	peek()	remove()
for 1 element O(n)	O(1)	O(1)
for array → n(n)=n²	n(1)=n	n(n) → n².

WAY 2

USING UNSORTED LINKED LIST

elements को randomly tail पर add करते चलो तो हमारे पास unsorted Linked List बनती जाएगी

add 10 — 10
add 20 — 10 → 20
add 5 — 10 → 20 → 5
add 22 — 10 → 20 → 5 → 22
add 44 — 10 → 20 → 5 → 22 → 44

add

हमें बस tail पर next element को add करना होगा & that element will be the new tail now.
So, we are just adding an element at the end of LinkedList. So the Time Complexity becomes O(1)

remove

So, basically in a Priority Queue, we remove an element with the maximum priority and by default, the Priority is given to the smallest element.

लेकिन हमारी linkedlist तो unsorted है तो यहाँ अगर हमें सबसे छोटे element को remove करना है तो पूरी linkedlist पर traverse करना होगा, smallest element को remove करने के लिए so, the Time complexity becomes O(n).

peek Similar to the remove() function, हमें smallest element को अगर unsorted linkedlist में find करना होगा और return करना होगा तो unsorted linkedlist में अगर हमें smallest element find करना है तो पूरे linked list पर travel करना होगा ; So the time complexity automatically becomes O(n).

Date / / Page No.

USING UNSORTED LINKED LIST

add	remove	peek
O(1)	O(n)	O(n).

n time add करना होगा → n(1) → n. { array के सभी elements के लिए
n time remove करना होगा → n(n) → n^2

Here, also, the Time Complexities are not the desired Time Complexities

The original PriorityQueue class have the Time Complexities
↳ add() → O(log n)
remove → O(log n)
peek → O(1).

We will try to achieve these Time Complexities using Heap Data Structure

Way 3

USING ARRAYLIST LIKE HEAP DATA STRUCTURE

इसे use करने से , PQ में add() function की T.C → O(log(n))
PQ में remove() function की T.C → O(log(n))
PQ में peek() function की T.C → O(1).

* पुरे array के सभी elements को PQ में add करने की TC → O(n element * logn)
  ↓
  log(n) time complexity for 1 element
  = n (logn).
  = O(n logn)

* पुरे array के सभी element को PQ में से remove करने की T.C.
  → O(n logn)

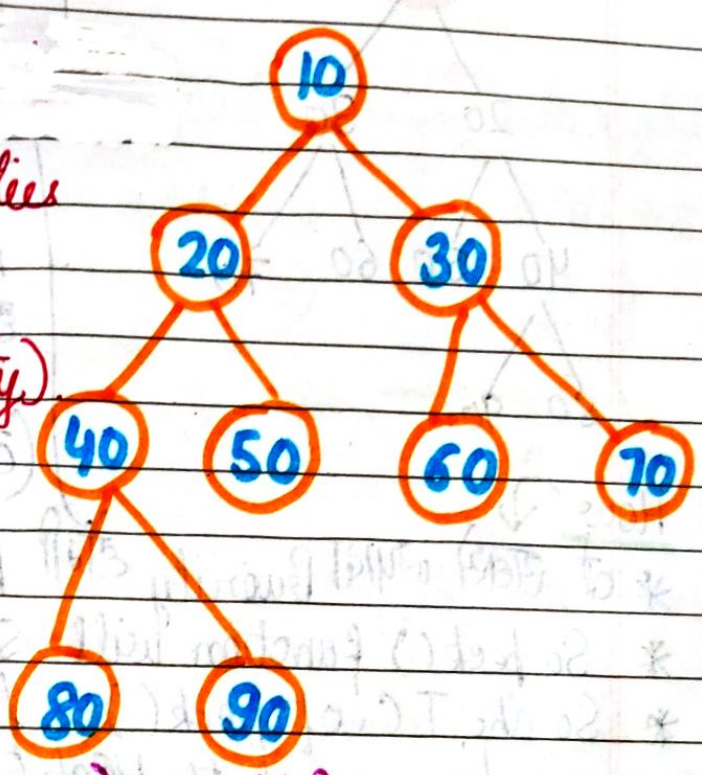
* पुरे array के सभी elements के लिए peek() function की T.C = n * 1
  = O(n).

What is Heap ? हम Heap ही क्यों use कर रहे हैं PQ बनाने के लिए

Heap is a Binary Tree.

having 2 special properties

→ CBT (complete Binary Tree)
→ HOP (Heap Order Property)



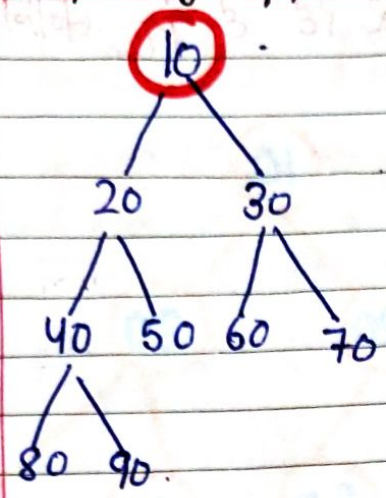
We are using Arraylist as a Heap so that we get
T.C of add() in PQ → log(n)
T.C of remove() in PQ → log(n)
T.C of peek() in PQ → O(1)

* तो हम (मतलब हम Arraylist से Heap जैसी working कराते हैं) Arraylist को as a Heap use करते हैं.
Because Heap के पास 2 special property होती हैं.

① Heap Order Property → * What ? Parent की priority दोनो children से बड़ी होती है हमेशा (मतलब Parent की value हमेशा children से छोटी होगी) (हर Parent की Priority उसके दोनो children से ज्यादा होगी).
(left और right child की Priority में कोई relation नही है. कोई भी बड़ा हो सकता है)

* Why ? हमने Heap Order Property का use क्यो किया है ?
→ अब Heap Order Property का use करने से peek() function की T.C O(1) हो जाएगी क्योकि हमे बस root return करना होगा क्योकि Parent, उसके दोनो children से ज्यादा Priority का होगा तो root की Priority सबसे ज्यादा होगी और हमे बस root return करना होगा.

The Heap Order Property means the priority of a parent is greater than its child. And to be specific the priority of any child is not pre-decided. This property basically helps to achieve the most efficient time complexity of peek() function which is constant O(1).



This tree is a Binary Tree (क्योंकि हर node के लिए 2 children हैं).

This Binary Tree follows the Heap Order Property क्योंकि हर Parent Node की Priority उसके दोनों children से ज्यादा है। (हर Parent node की value उसके दोनों children node से कम होगी).

* How ↓
* तो सबसे ज्यादा Priority होगी Root node की.
* So peek() function will simply return the root node.
* So the T.C of peek() function is O(1). Because we have used the Heap Order Property (HOP).

② COMPLETE BINARY TREE

* What?

↳ Complete Binary Tree Property says that

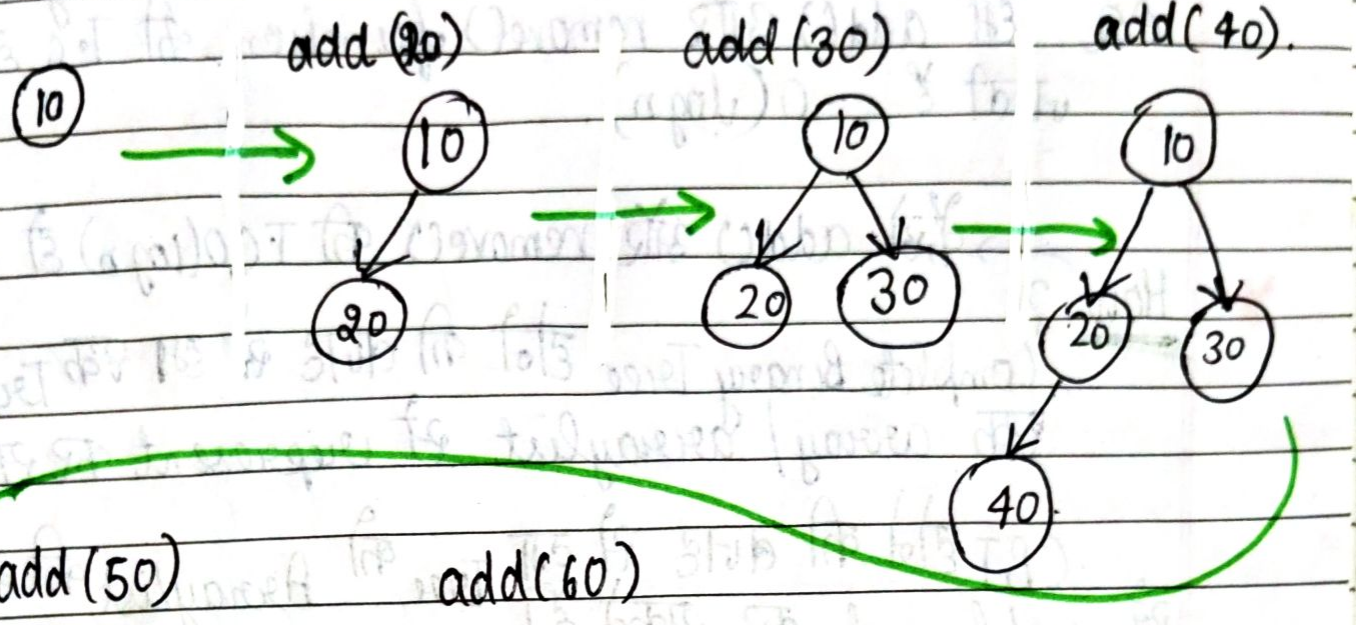
Let's suppose the height of the tree is h, then according to this property atleast h-1 levels should be completely filled, And the last hth level should be filled from left to right.

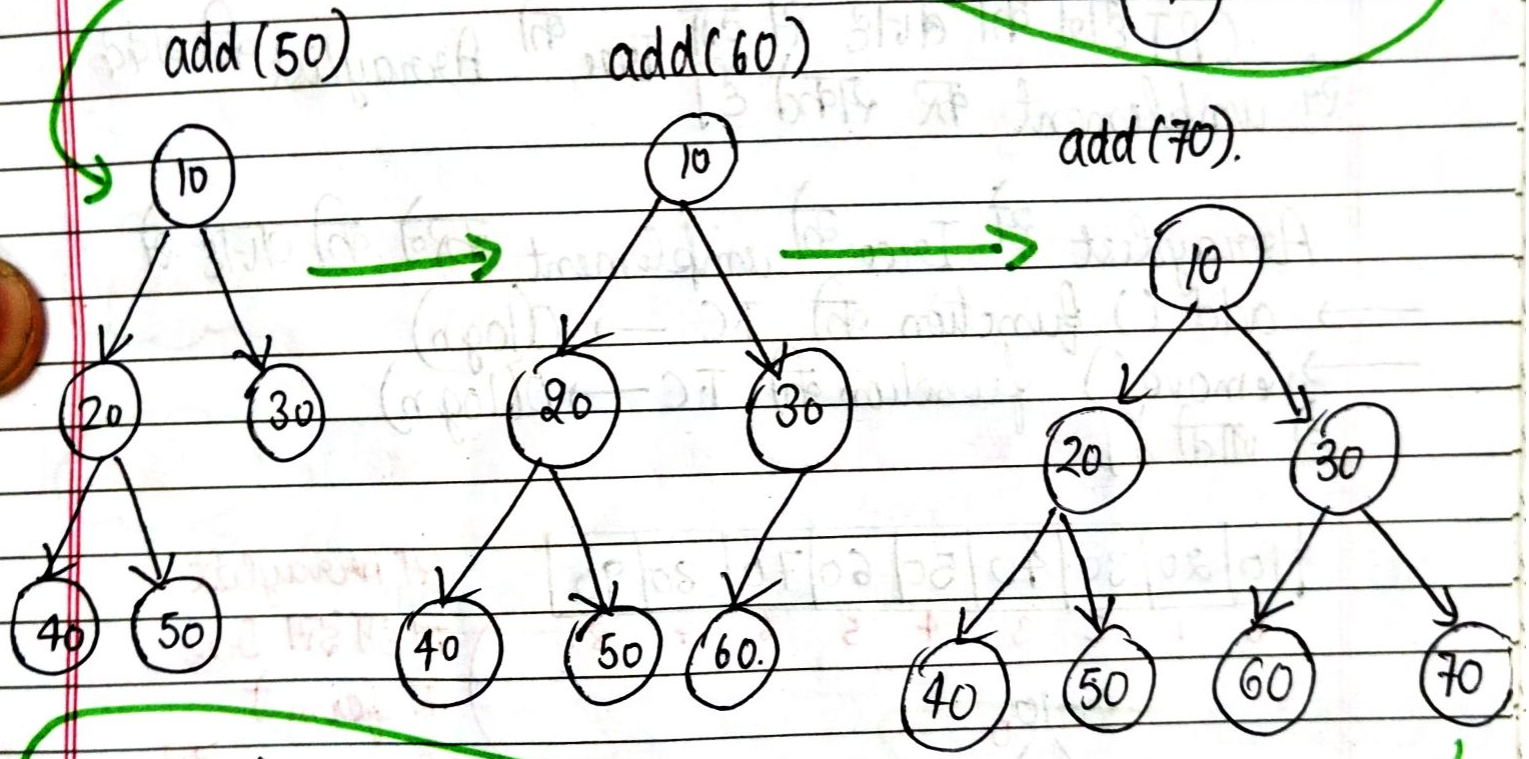
→ हर Parent को Right child attach करने से पहले left child attach होना चाहिए।

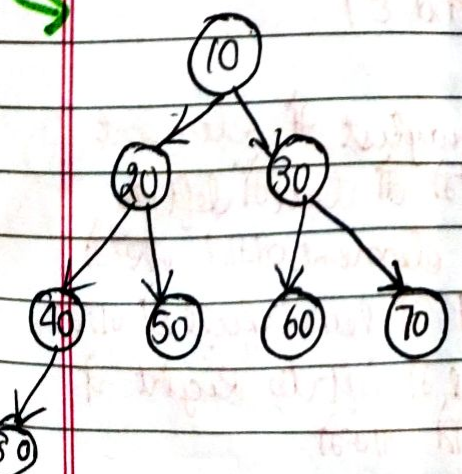
→ पहले वाले level को पूरा fill करने के बाद ही next level पर कोई element जाएगा

CBT → h-1 levels are full

→ h^{th} level is filled from left to right







There are 4 level in this Binary Tree.

3 levels are completely filled.

4th level is filled from left to right.

Why CBT (Complete Binary Tree)?

→ Complete Binary Tree में add() → log n
remove() → log n.

PQ में add() और remove() function की T.C हो जाती है O(log n).

→ कैसे add() और remove() की T.C, O(log n) हो गई?

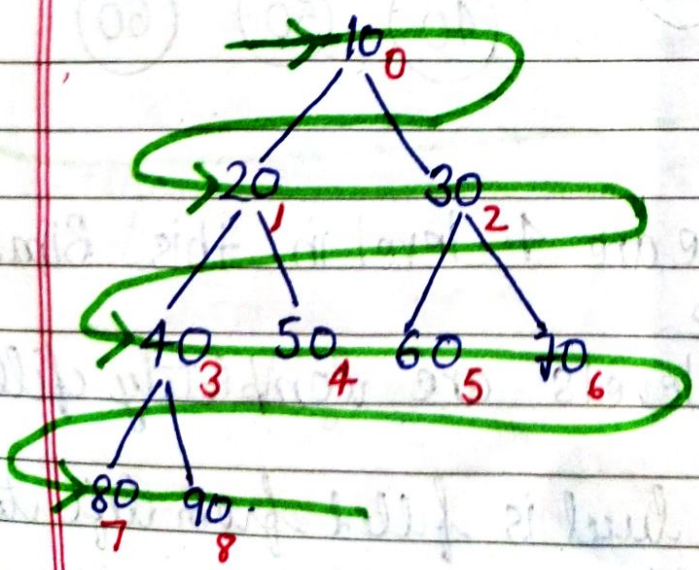
How?

Complete Binary Tree होने की वजह से हम एक Tree को एक array / Arraylist से represent कर सकते हैं।

CBT होने की वजह से हम Tree को ArrayList की मदद से implement कर सकते हैं।

ArrayList से Tree को implement करने की वजह से
→ add() function की T.C → O(log n)
→ remove() function की T.C → O(log n)
हो जाती है।

10	20	30	40	50	60	70	80	90
0	1	2	3	4	5	6	7	8

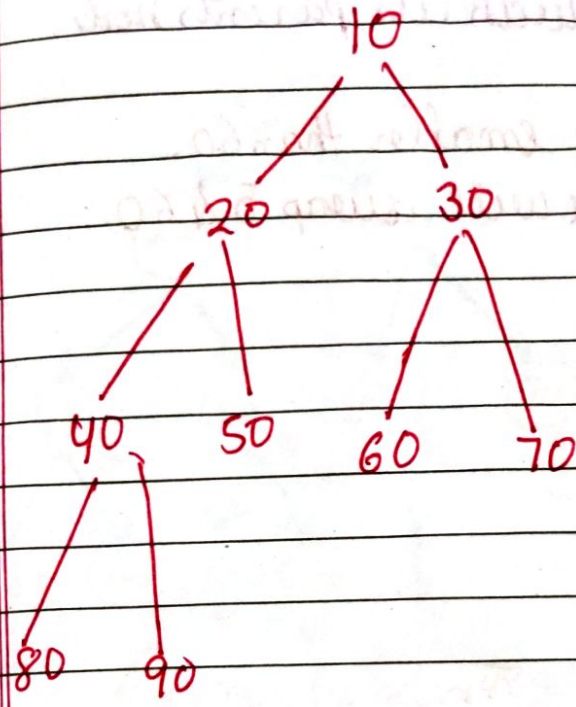


इस Arraylist को हम इस Tree की form में भी represent कर सकते हैं।

अगर arraylist में element बढ़ते जाएँगे तो tree में left to right वो element लगते जाएँगे और जब ये level full हो जाए तो next level में left to Right ये element लगते जाएँगे

An array / arraylist can also be visualized as a tree.

* An array/arraylist can also implement a Complete Binary Tree



So, this CBT is our Priority Queue.

This CBT is actually a Arraylist behind the scene.

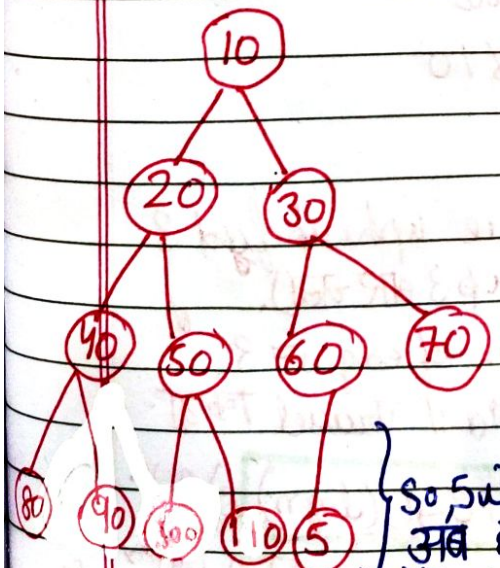
This arraylist also follows the Heap Order Property जो Parent की priority child से ज्यादा है So root is the peek.

peek() → peek() gives us the root ∴ O(1).
Root → list.get(0); ⟹ 10.

add() → add() function will add the data in the end of arraylist. So, In the CBT, left to right add करना होता, so 5 will be added after 120.

10	20	30	40	50	60	70	80	90	100	110	5

5 is added here

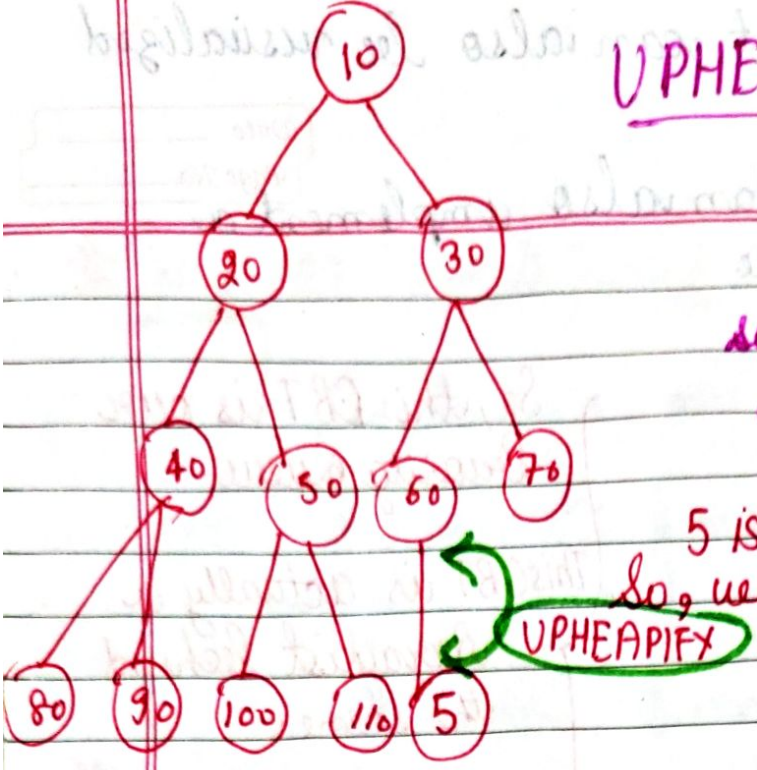


So 5 is added here → swap the node with its parent

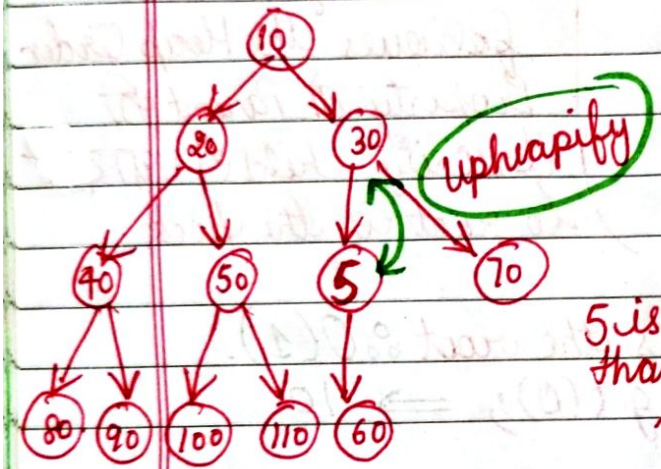
अब हम upHapify() function का use करेंगे और 5 को उसके parent से swap करते जाएँगे जब तक 5 is smaller than its parent.

UPHEAPIFY

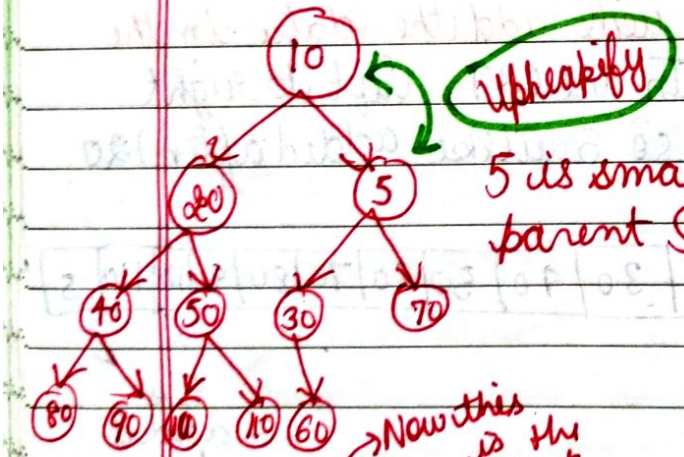
↳ node आगर उसके Parent से छोटा है (MIN heap property) so we will swap this node with its parents node.



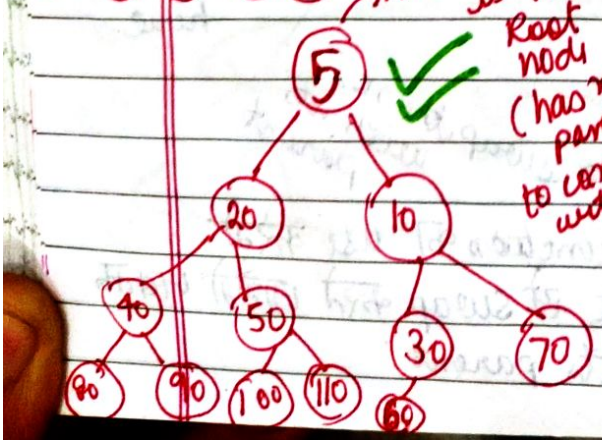
5 is smaller than 60.
So, we will swap 5 & 60.



5 is smaller than its parent node So swap 5 & 30.



5 is smaller than its parent So swap 5 & 10.



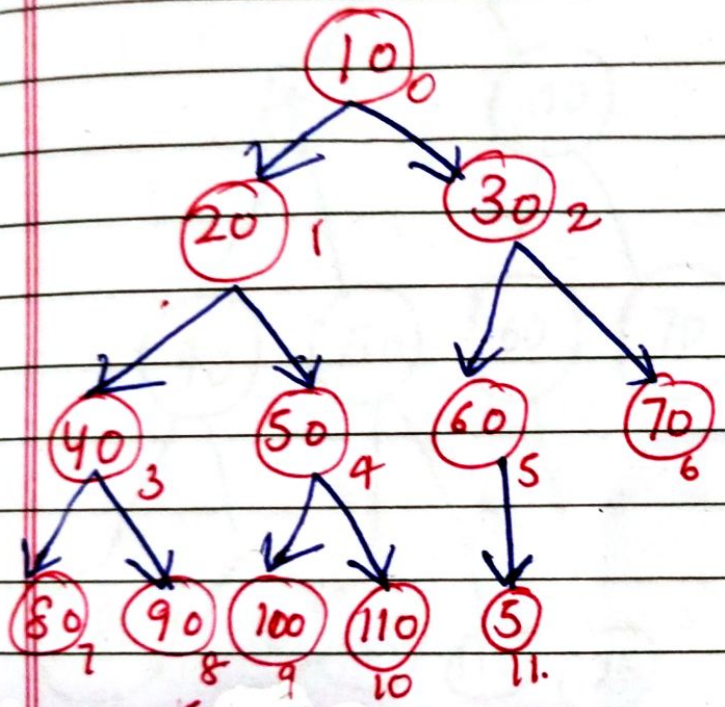
Now this is the Root node (has no parent to compare with)

So, we have upheapified 3 times (loop 3 बार चला).
height = no. of edges = 3
height के जितना travel किया.
height की T.C → O(log n) होती है
↳ इतना time लगेगा upheapify के loop में

Arraylist से Tree बनाने का क्या Benefit मिला?

Date / /
Page No.

↳ अब हम हर node से उसका Parent Index find कर सकते हैं।



2^0

2^1

2^2

2^3

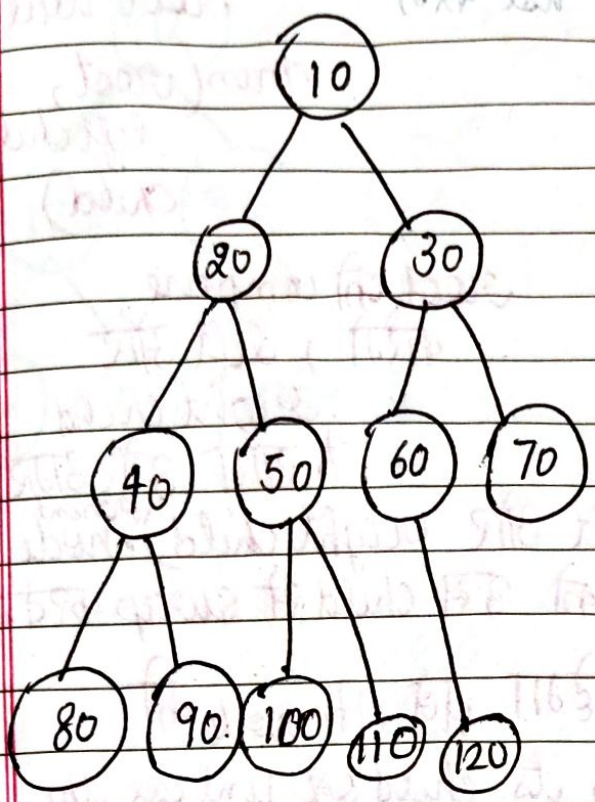
हमे कौनसे spot पर 5 को add करना है Arraylist से पता चलजायगा

(11) next index पर add करना है

लेकिन अगर हम Tree use करते CBT बनाने के लिए तो हमे पूरे Euler Tree को travel करके पता चलता कि 60 का child है 5 & 5 का Parent है 60.

remove() → हमें Highest priority element को remove करना है।

Date / /
Page No.



10	20	30	40	50	60	70	80	90	100	110	120
0	1	2	3	4	5	6	7	8	9	10	11

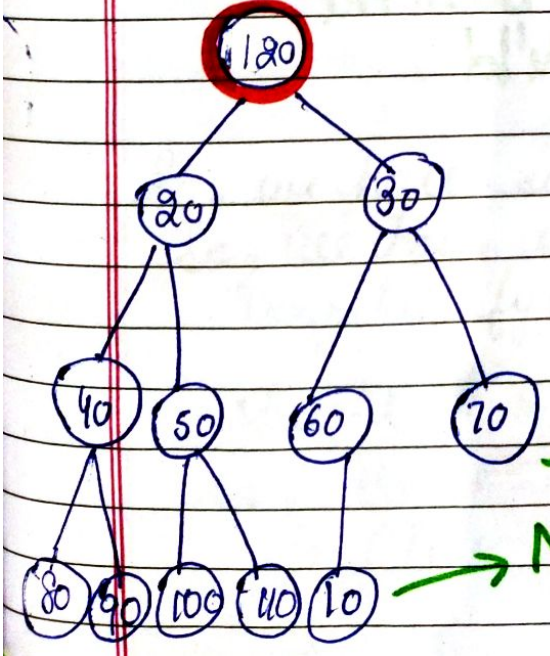
ArrayList में smallest element remove करना है. Ath index वाला element remove करने के लिए ArrayList में T.C O(n) होती है क्योंकि सभी elements को 1 index आगे shift होना पड़ेगा.

तो सबसे पहले हम Arraylist के first और last element को swap करेंगे.

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

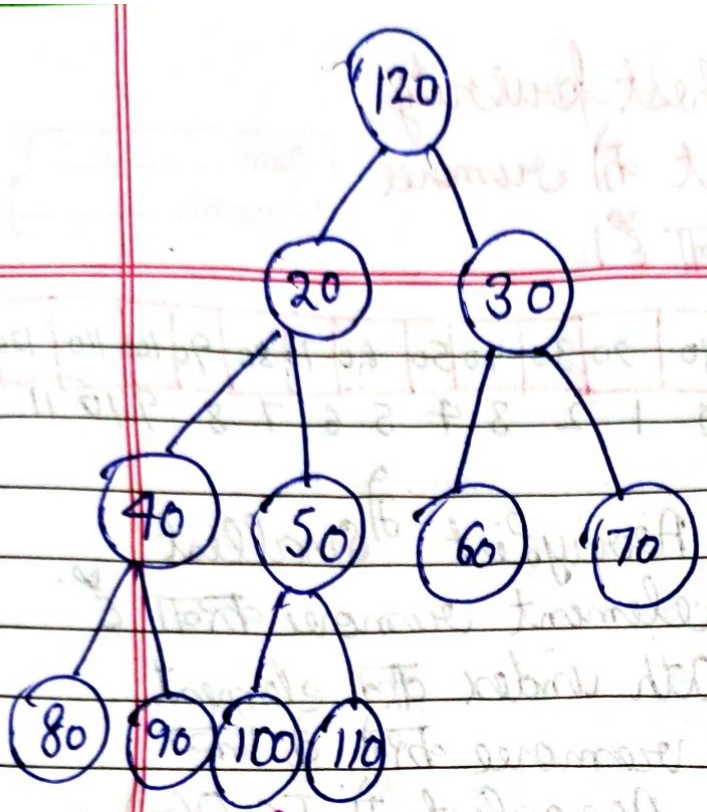
120	20	30	40	50	60	70	80	90	100	110	10

↳ Now, remove the last element of ArrayList



This CBT does not follow the Heap Order Property

→ Now, we will remove the last element of Arraylist.



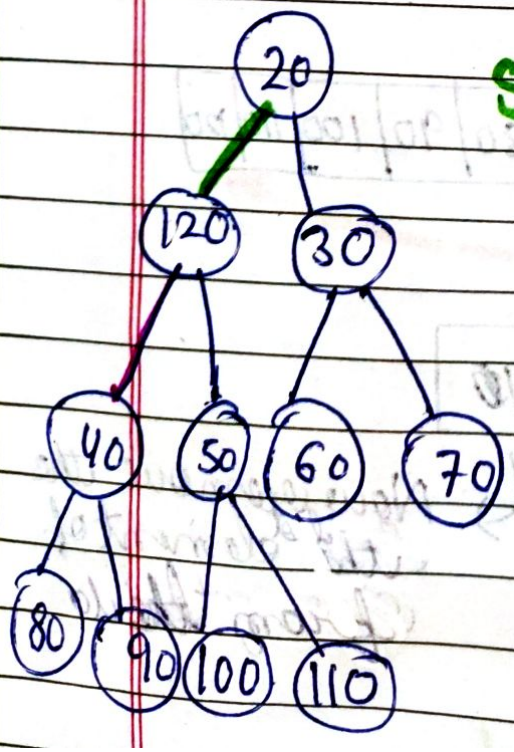
अब हम Downheapify() function का use करेंगे.

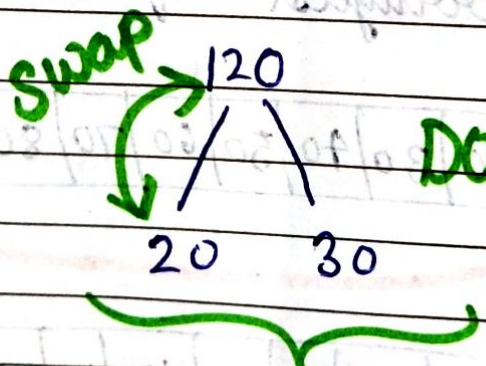
Date / /
Page No.

root will be min(root, left child, child)

root को compare करेंगे, left और right child के साथ और अगर parent node से किसी child की value (left और right child) ज्यादा है तो parent node को उस child से swap कर देंगे

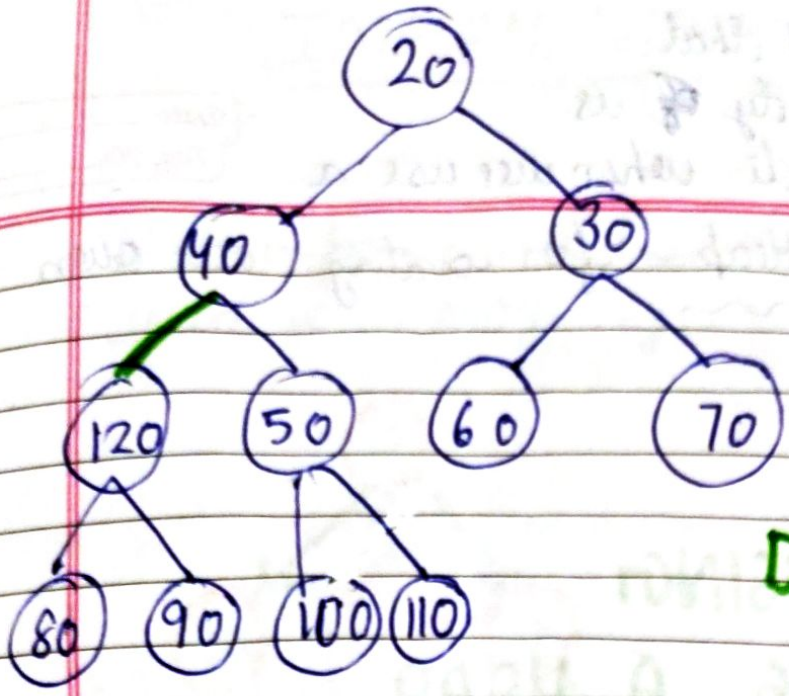
(ये loop तब तक चलता रहेगा जब तक node की value is greater than its child or unless the node is the leaf node (having no child).

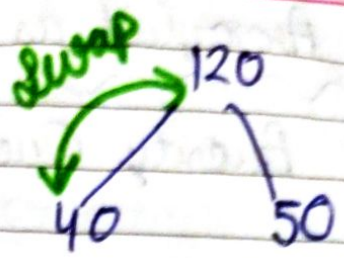




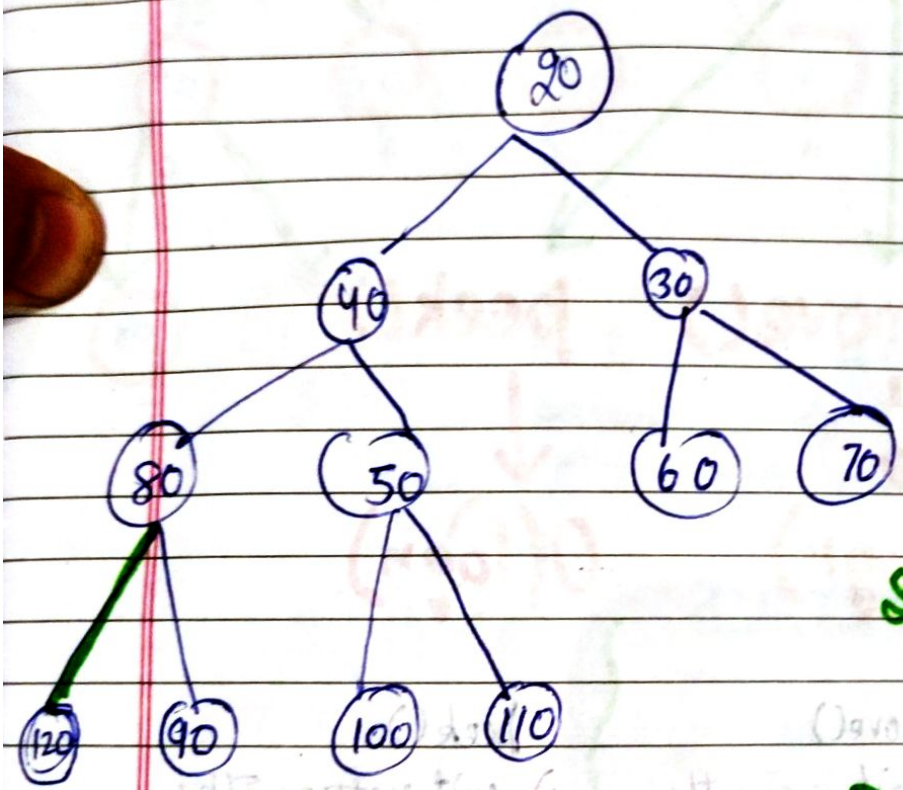
DOWNHEAPIFY

COMPARE the node with both the child

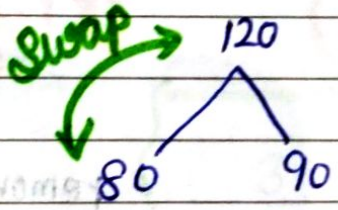




DOWNHEAPIFY



This tree follows the Heap order property



DOWNHEAPIFY

So, we have swapped the nodes 3 times
So, we have used 3 times the Downheapify() function ↳ 3 = height.

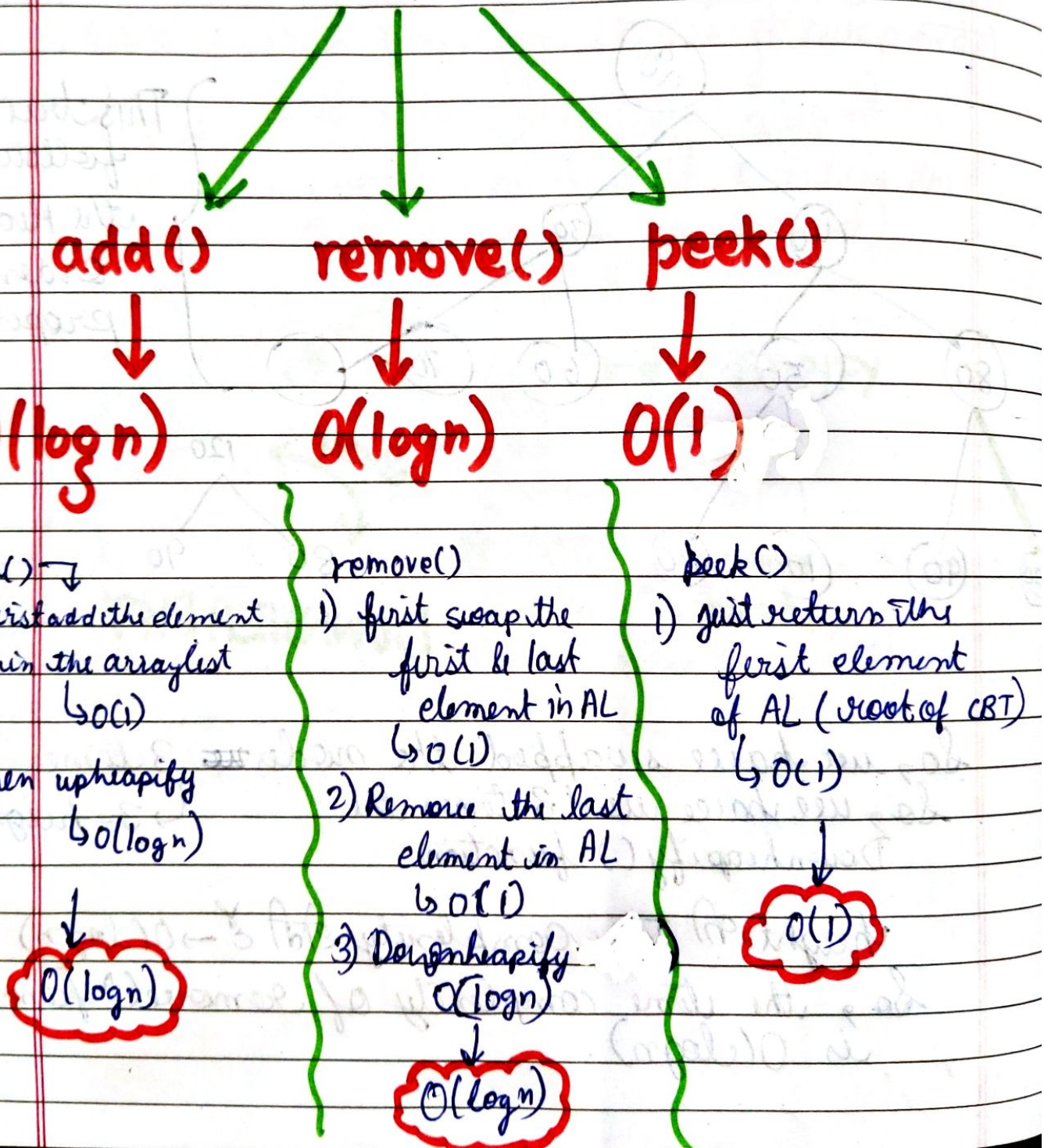
height of Time Complexity ~~of~~ & → O(log n)

So, the time complexity of remove() function is O(log n).

So, we have proved that
The time complexity is good & accurate when we use a Arraylist as a Heap for creating our own Priority Queue.

Date / / Page No.

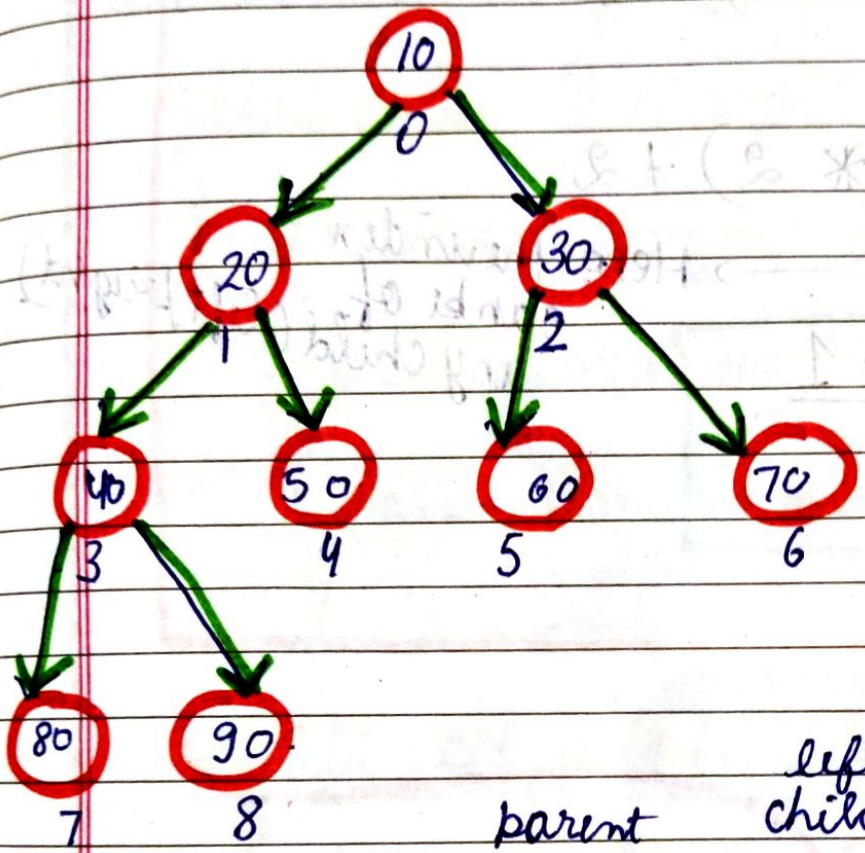
PRIORITY QUEUE USING ARRAYLIST AS A HEAP



IMPLEMENTATION

Date / /
Page No.

It is important to know that the data structure used for storage by the Heap is Arraylist



Formulas.

Left child index
$= 2(\text{parent index}) + 1.$

Right child index
$= 2(\text{parent index}) + 2.$

parent	left child	right child
0	1	2.
	$2 \cdot 0 + 1 = 1$	$2 \cdot 0 + 2 = 2.$
1	2	3.
	$2 \cdot 1 + 1 = 1$	$2 \cdot 1 + 2 = 3$

$$\text{parent index} = \frac{\text{child index} - 1}{2}$$

for left child
$pi = \frac{2-1}{2} = \frac{1}{2} \approx 1.$

for right child
$pi = \frac{3-1}{2} = \frac{2}{2} = 1.$

IMPORTANT FORMULAES

Date / /
Page No.

left child index → LCI = (pi * 2) + 1

Parent Index → RCI = (pi * 2) + 2

Here the index can be of any child (left / right)

$$pi = \frac{CI - 1}{2}$$

If we implement Arraylist as a CBT (complete binary tree), then these formulaes are very important

PRIORTY QUEUE USING HEAP

<code>

```
import java.util.*;

public class Main {

  public static class PriorityQueue
  { ArrayList<Integer> data;

    public PriorityQueue()
    {
      { data = new ArrayList<>();
      }
    }

    public void add(int val)
    {
      data.add(val);
      upheapify(data.size()-1);
    }

    void upheapify(int i)
    {
      if(i==0)
      { return;
      }
      int pi = (i-1)/2;
      if(data.get(pi) > data.get(i))
      { swap(pi,i);
      }
      upheapify(pi);
    }
```

हमने एक Arraylist को declare किया

This is the constructor of Priority Queue class

This Arraylist works as a heap

हमने data Arraylist को initialize किया

ये add function है

we are adding the data in the end of AL

Then we will upheapify the data at the last index जहाँ index pass किया गया है

ये upheapify function है

अगर index 0 है तो उसका कोई parent ही नहीं होगा because वो root है तो simply return

parent index find किया

अगर parent index पर data की value ज्यादा है तो हमें swap करना है और फिर recursive call लगाओ

```
public int remove()
{
  if (data.size() == 0)
  {
    System.out.println("Underflow");
    return -1;
  }
  swap(0, data.size()-1);
  int val = data.remove(data.size()-1);
  downheapify(0);

  return val;
}

void downheapify(int i)
{
  int mini = i;
  int lci = 2*i + 1;
  int rci = 2*i + 2;
  if (lci < data.size() &&
      data.get(lci) < data.get(mini))
  {
    mini = lci;
  }
  if (rci < data.size() &&
      data.get(rci) < data.get(mini))
  {
    mini = rci;
  }
```

यह remove() function है

पहले AL के first और last element को swap करदो

फिर first element (0 index) पर downheapify function call करदो

फिर last element को remove करदो AL से

downheapify function है। इसमें parent index (i) डालदिया।

mini variable में parent node का index store कर लिया।

अगर left child index का data parent index से कम है तो parent index में child index को बनादो।

```
if (mini != i)
{
    swap(mini, i);
    downheapify(mini);
}
}
```

और अगर mini variable और parent index की value अलग नहीं है तो actual parent index (i) की value change हुई है, और हमें recursive call करनी होगी।

```
public void swap(int i, int j)
{
    int ith = data.get(i);
    int jth = data.get(j);
    data.set(i, jth);
    data.set(j, ith);
}

public int peek()
{
    if (data.size()==0)
    {
        System.out.println("Underflow");
        return -1;
    }
    return data.get(0);
}

public int size()
{
    return data.size();
}
```

```
public static void main (String args[])
{ BufferReader br = new BufferReader (new InputStreamReader (System.in));
PriorityQueue qu = new PriorityQueue ();
String str = br.readLine();
while (str.equals("quit") == false)
{
if (str.startsWith("add"))
{
int val = Integer.parseInt (str.split(" ")[1]);
qu.add(val);
}
else if (str.startsWith("remove"))
{
int val = queue.remove();
if (val != -1)
{
System.out.println(val);
}
}
else if (str.startsWith("peek"))
{
int val = queue.peek();
if (val != -1)
{
System.out.println(val);
}
}
}
```

→ PriorityQueue class का एक object बनाया।

→ इस line by line string को read करेंगे और element को add करेंगे।

अगर string add से start हो रही है तो उस पर split करेंगे और string को qu पर add करेंगे।

→ splitted array से 1st index वाला data हमारा element है जो हम PQ में add करना है।

PQ में element को add करेंगे

अगर string का first word remove है तो value को PQ से remove करेंगे।

removed element को print करेंगे

अगर string का first word peek है तो peek function call होगा और received value को print करना होगा।

```
else if (str.startsWith ("size"))
{
    System.out.println (qu.size());
}
str = br.readLine();
}
}
```

अगर string की first word size है तो size function call होगा।

फिर पहले size print करना होगा।

read the string in next line

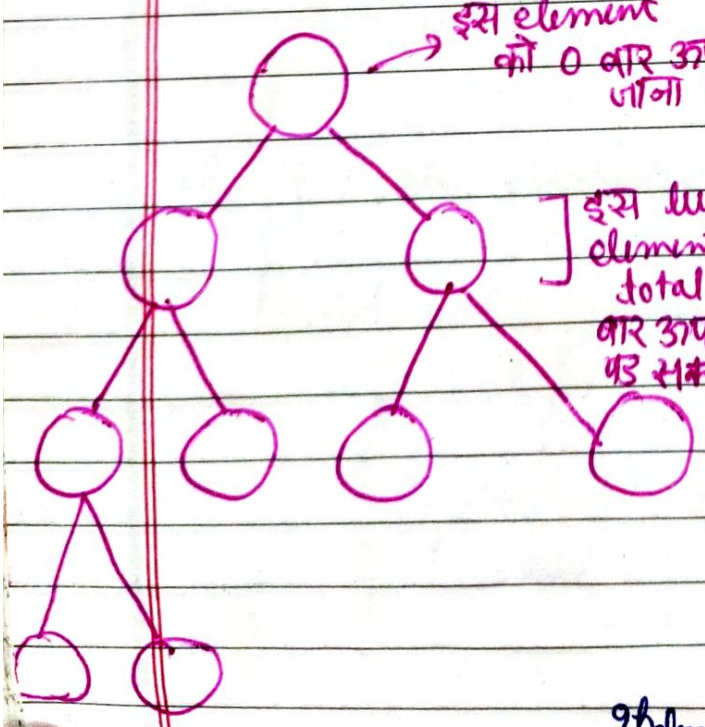
EFFICIENT HEAP CONSTRUCTOR

```
public MyPriorityQueue(int[] ars)
{ data = new ArrayList<>();
  for (int val: ars)
      add(val);
}
```

जो Heap Constructor हमारे पास था
↳ उसमे अगर हमारी array में n element given है
PQ में add() function
की time complexity O(log n) है.

तो array के सभी elements पर for loop लगाकर अगर
हम PQ में add करते है तो
for loop की time complexity → O(n)
1 element की PQ में add करने की T.C → O(log n)

So, the Time Complexity to add n elements in PQ → O(n log n).



$2^0 \cdot 0$ — element level — upheapify

$2^1 \cdot 1$ — element level — upheapify

$2^2 \cdot 2$ — element level

$2^h \cdot h = 2^{\log n} \cdot \log n$

2^h elements को h level ऊपर जाना पड़ सकता है

$= n \log n$

Find out h ? → h = height

$n = 2^0 + 2^1 + 2^2 + 2^3 \text{ ---} + 2^h$

Date / /
Page No.

यह GP है h+1 terms है total

n = no. of elements
h = height of Tree

GM का sum $= \frac{a(r^n - 1)}{r - 1}$

r = ratio = 2^1
a = 1 = 2^0
n = no. of terms = h+1

$n = \frac{2^0(2^{h+1} - 1)}{2 - 1}$

$n = \frac{1(2^{h+1} - 1)}{1}$

$n = \frac{2^{h+1} - 1}{1}$

$n = 2^{h+1} - 1$

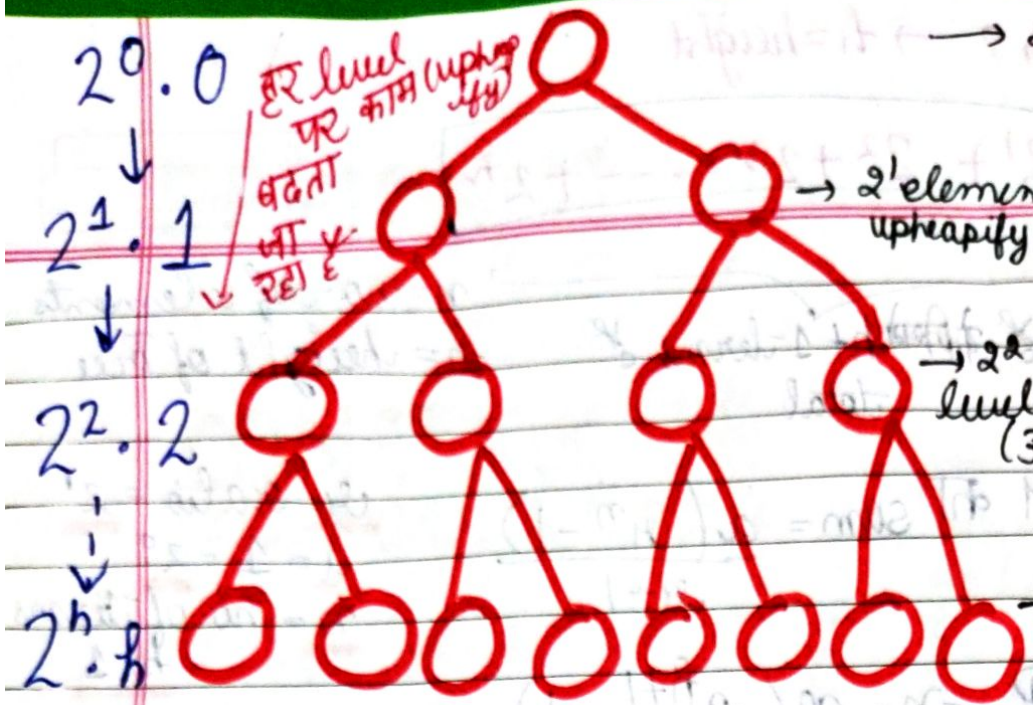
$n + 1 = 2^{h+1}$

Taking log both sides

$h + 1 = \log_2(n+1)$

$h = \log_2(n+1) - 1.$

$h \simeq \log n.$ (तो COMPLETE Binary Tree की height $\log n$ होती है ये तो हमने Prove करलिया.



$$T(n) = 2^0 \cdot 0 + 2^1 \cdot 1 + 2^2 \cdot 2 + 2^3 \cdot 3 + \cdots + 2^h \cdot h.$$

$T(n)$ ↓ Time Complexity to add n elements in PQ & then upheapify (मतलब in the construction of PQ)

Total time taken in adding n elements in the PQ and then upheapifying them

$$= \underbrace{2^0 \cdot 0 + 2^1 \cdot 1 + 2^2 \cdot 2 \cdots}_{\text{इस सब को अगर ignore भी करे}} \underbrace{2^h \cdot h.}_{\text{सिर्फ अगर ये देखी जाए}}$$

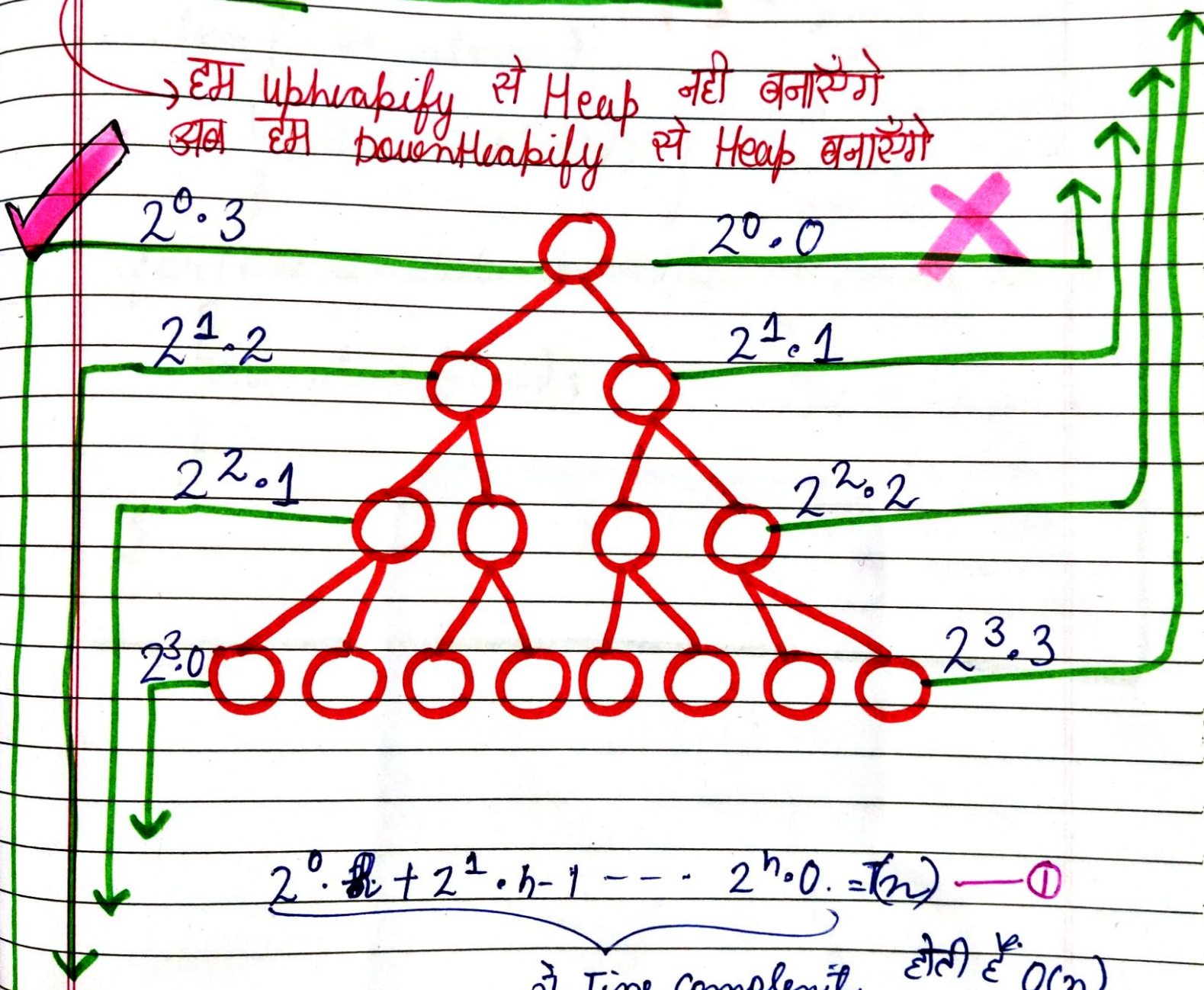
$$2^h \cdot h = 2^{\log n} \cdot \log n$$

$$T(n) = \underbrace{2^{\log n}}_{n} \cdot \underbrace{\log n}_{\log n} = n \log n.$$

सबसे ज्यादा elements last level पर होंगे (2^h) तो सबसे ज्यादा काम भी (upheapify) हमें last level पर ही करना पड़ेगा

BETTER APPROACH

→ हम upheapify से Heap नहीं बनाएँगे अब हम downHeapify से Heap बनाएँगे



$2^0 \cdot h + 2^1 \cdot h-1 \cdots 2^h \cdot 0 = T(n)$ —— ①

∴ Time complexity होती है $O(n)$.

Multiply Both sides by 2.

$2^1(h) + 2^2(h-1) \cdots 2^{h+1}(0) = 2T(n)$ —— ②

Subtract ② from ①

$-2^h - 2^1(h-1) - \cdots -2^1 = -T(n)$

$T(n) = 2^h + 2^{h-1} + 2^{h-2} \cdots 2^1 - 2^0(h) = \boxed{2(2^h-1) = h}$

$T(n) = 2n - 2 - \log n$

$T(n) \propto n$

* पहले सभी elements को PQ में add करदो और फिर

* $\left(\frac{n}{2} - 1\right)$ index से 0 index तक downheapify function call करना होता है।

It works

→ it Priority Queue class का Constructor है

Date / /
Page No.

```
public MyPriorityQueue (int[] arr)
{
    data = new ArrayList<>();

    for (int val : arr)
    {
        add(val);
    }
}
```

→ अब हम PQ class के add function का use करेंगे जो add function uses the upheapify() in it.

```
for (int val : arr)
{
    data.add(val);
}
```

} पहले सभी element को Arraylist(PQ) में add करलो

```
for (int i = data.size()/2 - 1 ; i >= 0 ; i--)
{
    downheapify(i);
}
}
```

→ फिर $\frac{n}{2} - 1$ index से 0th index तक सभी elements पर downheapify function call करदो।

Now, the time Complexity for adding all the elements of array in PQ will be $O(n)$.