

HashMap {Order is not maintained}

- O(1) ① put → hm.put(key, value); \Rightarrow if key is already present then value is overrided.
- O(1) ② containsKey → hm.containsKey(key); $\begin{cases} \text{true (present)} \\ \text{false (not present)} \end{cases}$
- O(1) ③ get → hm.get(key); \rightarrow if key is present, returns the value, else returns null.
- O(1) ④ getOrDefault \hookrightarrow hm.getOrDefault(key, specifiedVal);
if key is present, returns the val,
else returns the specified val.
- O(n) ⑤ keySet() \hookrightarrow hm.keySet() \rightarrow returns a set of keys.

HashMap {Order is not maintained}

- O(1) ① put → hm.put(key, value); \Rightarrow if key is already present then value is overrided.
- O(1) ② containsKey → hm.containsKey(key); $\begin{cases} \text{true (present)} \\ \text{false (not present)} \end{cases}$
- O(1) ③ get → hm.get(key); \rightarrow if key is present, returns the value, else returns null.
- O(1) ④ getOrDefault → hm.getOrDefault(key, specifiedVal);
if key is present, returns the val,
else returns the specified val.
- O(n) ⑤ keySet() \rightarrow hm.keySet() \rightarrow returns a set of keys.

Highest frequency character

Highest Frequency Character

• Easy

◀ Prev ▶ Next

1. You are given a string str.
2. You are required to find the character with maximum frequency.

"aabbaacabcaabbbaaddda"

a - X 2 B 1 S K 7

b - > 1 Z P 4 \$ 6

c - X 2 3

d - Y 2 Z 1

Code for Max freq character

```
public static void main(String[] args) throws Exception {
    // write your code here
    Scanner scn = new Scanner(System.in);
    String str = scn.nextLine();
    HashMap<Character, Integer> fmap = new HashMap<>();

    for(int i=0;i<str.length();i++) {
        fmap.put(str.charAt(i),fmap.getOrDefault(str.charAt(i),0) + 1);
    }

    char ans = '0';
    int maxFreq = 0;
    for(Character ch : fmap.keySet()) {
        if(fmap.get(ch) > maxFreq) {
            ans = ch;
            maxFreq = fmap.get(ch);
        }
    }

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

Get Common Elements - I

$a_1 \rightarrow 1 \ 1 \ 2 \ 2 \ 2 \ 3 \ 5$

$a_2 \rightarrow 1 \ 1 \ 1 \ 2 \ 2 \ 4 \ 5$

Ans : 1 2 5

Here we do not need a
hashmap as only hashset
will be enough to check the
occ of ele in the array once.

hm1 (for a1)

1 - 2

2 - 3

3 - 1

5 - 1

hm2 (for a2)

1 - 3

2 - 2

4 - 1

5 - 1

Code for get Common Elements -1

```
public static void main(String[] args) throws Exception {
    // write your code here
    Scanner scn = new Scanner(System.in);
    int n1 = scn.nextInt();
    int[] a1 = new int[n1];
    HashSet<Integer> hs1 = new HashSet<>();
    for(int i=0;i<n1;i++) {
        a1[i] = scn.nextInt();
        hs1.add(a1[i]);
    }

    int n2 = scn.nextInt();
    int[] a2 = new int[n2];
    HashSet<Integer> hs2 = new HashSet<>();
    for(int i=0;i<n2;i++) {
        a2[i] = scn.nextInt();
        hs2.add(a2[i]);
    }

    for(int i=0;i<a2.length;i++) {
        if(hs2.contains(a2[i]) && hs1.contains(a2[i])) {
            System.out.println(a2[i]);
            hs2.remove(a2[i]);
        }
    }
}
```

Get Common Elements - II (find Intersection)

```
Scanner scn = new Scanner(System.in);
int n1 = scn.nextInt();
int[] a1 = new int[n1];

HashMap<Integer, Integer> hm1 = new HashMap<>();
for(int i=0;i<n1;i++) {
    a1[i] = scn.nextInt();
    int oFreq = hm1.getOrDefault(a1[i],0);
    hm1.put(a1[i],oFreq + 1);
}

int n2 = scn.nextInt();
int[] a2 = new int[n2];
HashMap<Integer, Integer> hm2 = new HashMap<>();
for(int i=0;i<n2;i++) {
    a2[i] = scn.nextInt();
    int oFreq = hm2.getOrDefault(a2[i],0);
    hm2.put(a2[i],oFreq + 1);
}

for(int ele : a2) {
    if(hm2.get(ele) > 0) {
        if(hm1.containsKey(ele) && hm1.get(ele) > 0) {
            System.out.println(ele);
            hm1.put(ele, hm1.get(ele) - 1);
            hm2.put(ele, hm2.get(ele) - 1);
        }
    }
}
```

Here we need hashmap
because in intersection,
frequency is also necessary.

Longest Consecutive Sequence of Elements {Largest Consecutive Seq : Leetcode - 1253}

{100, 4, 200, 1, 3, 2}

- ① Put true against all ele's in Hash Map.
- ② Now, if ele - 1 is found in hashmap, make ele false as it is not the start of seq.
- ③ If the value is true, try to find out the max len.

100 - t
4 - ~~t~~ f
200 - t
1 - t
3 - ~~t~~ f
2 - ~~t~~ f

Code

```
public int longestConsecutive(int[] nums) {
    HashMap<Integer,Boolean> hm = new HashMap<>();

    for(int i=0;i<nums.length;i++) {
        hm.put(nums[i],true);
    }

    for(int key: hm.keySet()) {
        if(hm.containsKey(key-1) == true) {
            hm.put(key,false);
        }
    }

    int maxLen = 0;
    for(int key: hm.keySet()) {
        if(hm.get(key) == true) {
            int x = key;
            int len = 1;
            while(hm.containsKey(x + 1) == true) {
                len++;
                x = x + 1;
            }
            maxLen = Math.max(maxLen,len);
        }
    }
}
```

Heaps {Priority Queue} {Removes higher priority element first}

Based on Priority { add } { remove } { peek } → $O(1)$

```
public static void main(String[] args) throws Exception {
    PriorityQueue<Integer> pq = new PriorityQueue<>(Collections.reverseOrder());
    int[] ranks = {22, 99, 3, 11, 88, 4, 1};
```

Now, it will give high priority to higher value!

In PQ, we can specify whether lower value has a higher Priority or vice versa. { By default, lower value has higher priority }

K Largest Elements $\{K \text{ sized Static Window}\}$

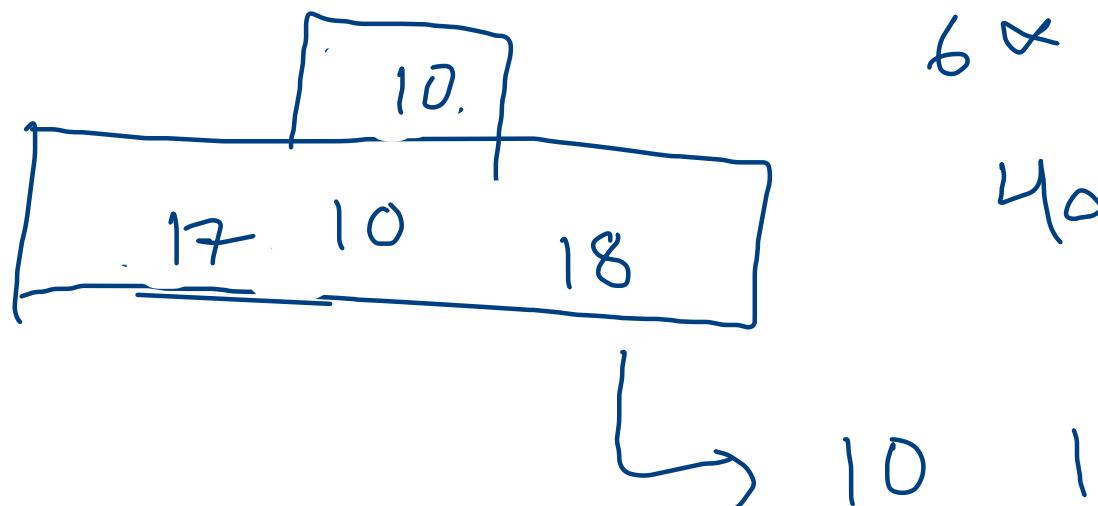
3 largest elements.

2 10 5 17 7 18 6 4

⇒ Make a PA of size = 3.

2 10 5 17 7
0 1 2 3 4

18 6 4
5 6 7



10 17 18 (3 largest elems in sorted order)

Code for K Largest Elements

```
int k = Integer.parseInt(br.readLine());
// write your code here
PriorityQueue<Integer> pq = new PriorityQueue<>();

for(int i=0;i<k;i++) {
    pq.add(arr[i]);
}

for(int i=1;i<=n-k;i++) {
    int incomingEle = arr[i + k -1];
    if(incomingEle > pq.peek()) {
        pq.remove();
        pq.add(incomingEle);
    }
}

while(pq.size() > 0) {
    System.out.println(pq.remove());
}
```

k^{th} Largest Element {Leetcode : 215}

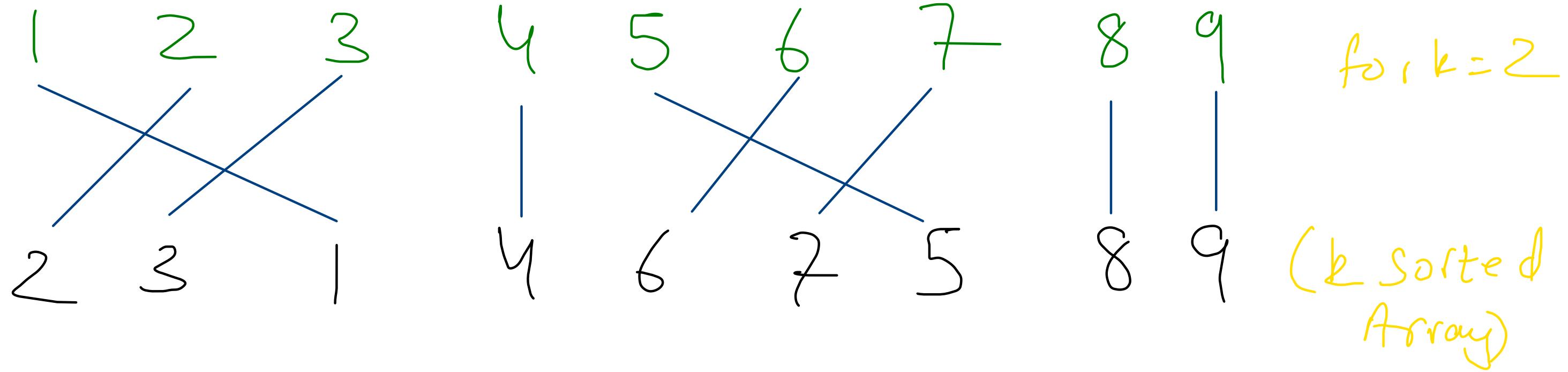
Same as prev ques. Just, return the peek at the end .

```
class Solution {
    public int findKthLargest(int[] arr, int k) {
        PriorityQueue<Integer> pq = new PriorityQueue<>();
        int n = arr.length;
        for(int i=0;i<k;i++) {
            pq.add(arr[i]);
        }

        for(int i=1;i<=n-k;i++) {
            int incomingEle = arr[i + k -1];
            if(incomingEle > pq.peek()) {
                pq.remove();
                pq.add(incomingEle);
            }
        }

        return pq.peek();
    }
}
```

Sort k sorted Array



k Sorted Array: Every element in the array moves to at most k right/left positions from its sorted position.

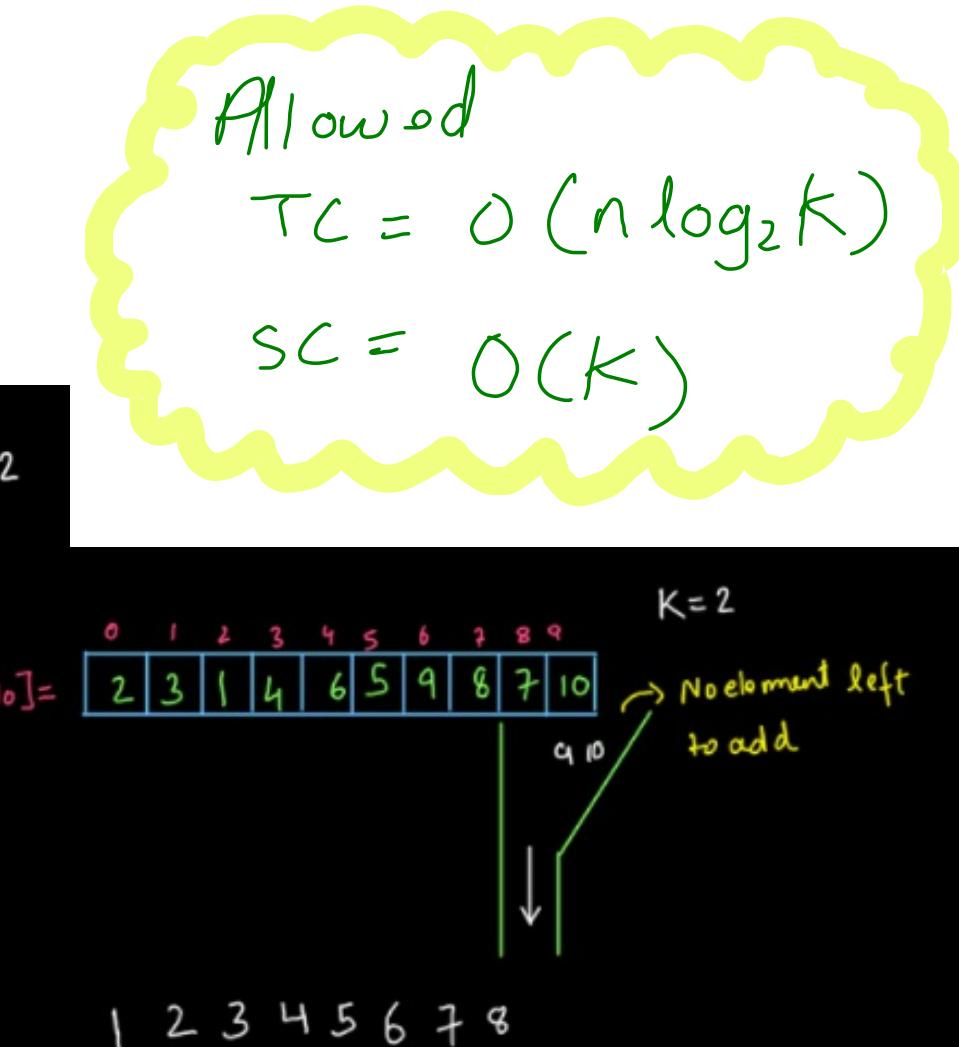
Q

2 3 1 4 6 7 5 8 9

| $a[10] =$ | $\begin{array}{cccccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ \hline 2 & 3 & 1 & 4 & 6 & 5 & 9 & 8 & 7 & 10 \end{array}$ | $K=2$ |
|-----------|--------------------------------------------------------------------------------------------------------------------------------|-------|
| | <p>2 3</p> | |

| $a[10] =$ | $\begin{array}{cccccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ \hline 2 & 3 & 1 & 4 & 6 & 5 & 9 & 8 & 7 & 10 \end{array}$ | $K=2$ |
|-----------|--------------------------------------------------------------------------------------------------------------------------------|-------|
| | <p>2 3 4</p> | |

| $a[10] =$ | $\begin{array}{cccccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ \hline 2 & 3 & 1 & 4 & 6 & 5 & 9 & 8 & 7 & 10 \end{array}$ | $K=2$ |
|-----------|--------------------------------------------------------------------------------------------------------------------------------|-------|
| | <p>3 4</p> | |



Code

```
public static void main(String[] args) throws Exception {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    int n = Integer.parseInt(br.readLine());
    int[] arr = new int[n];

    for (int i = 0; i < n; i++) {
        arr[i] = Integer.parseInt(br.readLine());
    }

    int k = Integer.parseInt(br.readLine());
    // write your code here

    PriorityQueue<Integer> pq = new PriorityQueue<>();
    int[] ans = new int[n];

    for(int i=0;i<=k;i++) {
        pq.add(arr[i]);
    }

    for(int i=k+1 ; i< n; i++) {
        System.out.println(pq.remove());
        pq.add(arr[i]);
    }

    while(pq.size() > 0) {
        System.out.println(pq.remove());
    }
}
```

Median Priority Queue

(UV VImp) { find Median from
Data Stream }

Fintcocke
Hard

add → adds an element to the Median PQ

remove → removes the median ele from the PQ

peek → returns the median element.

size → returns the size of the PQ.

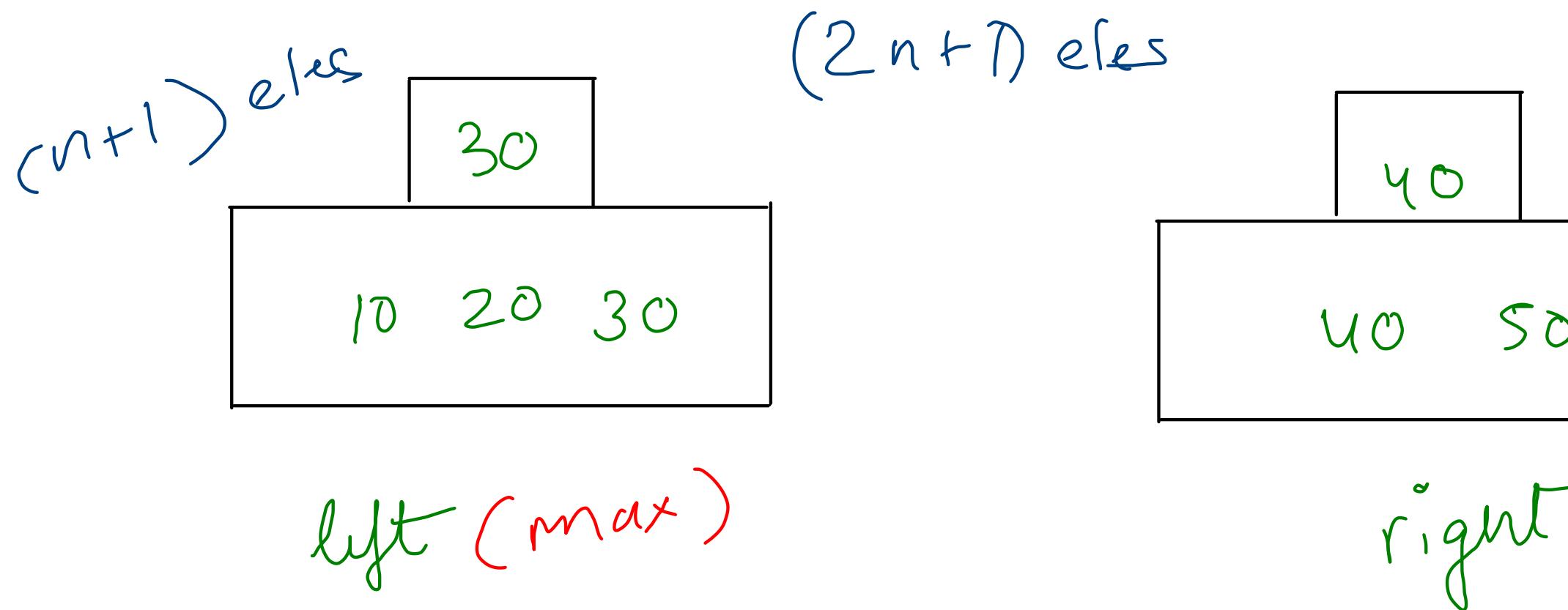
10, 20, 30, 40
↓
median

In case
of even
elec

10, 20, 30, 40, 50
↓
median

⇒ Odd
elec

We will use 2 priority queues to obtain the MPQ.



By default:
insert in left.

left = smaller half of the data (max Heap)

right = larger half of the data (min Heap)

* The gap b/w size of PQs should always be either 0 or 1 and more -

Ex: $10, 20, 30, 40, 50$

By default, try to add in left. If incoming ele is greater than peek of the right, then insert the ele in the right.

Median will be the peek of that PA whose size is bigger.

If size is equal, median will be picked from the left PA.

Remove the ele from larger sized PA. In case of same size, remove from left.

Concept

add(10)

add(20) * size diff

add(30) ↳ right peek

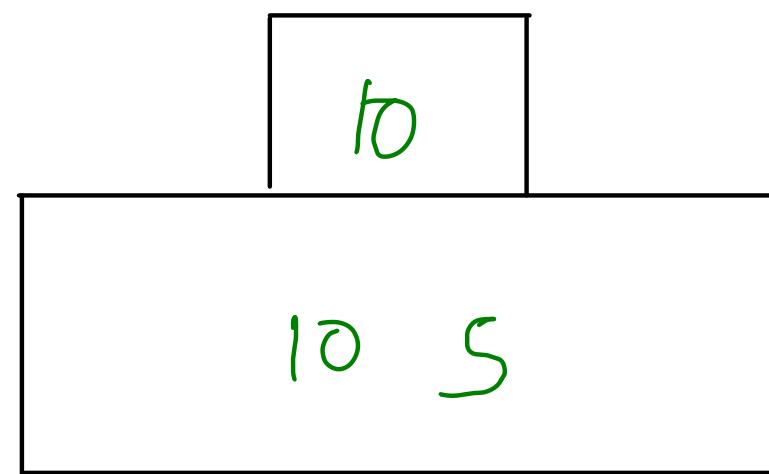
add(40) * size diff

add(50)

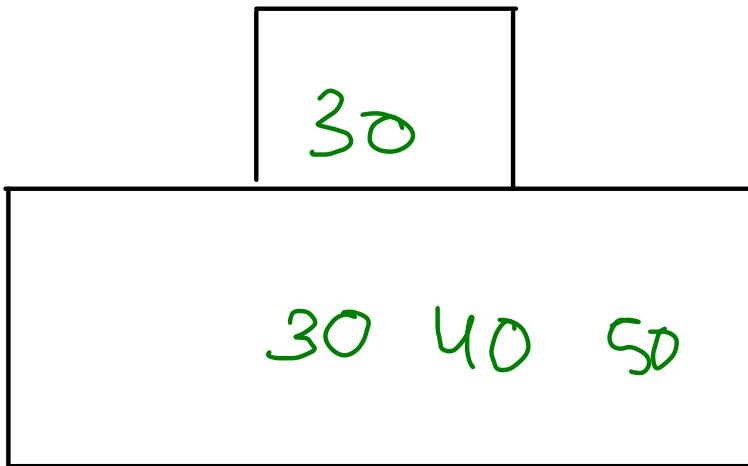
add(5)

remove

Implementation



left (max)



right (min)

Code { COPY PASTE THE SAME CODE IN LINTCODE }

```
public static class MedianPriorityQueue {
    PriorityQueue<Integer> left;
    PriorityQueue<Integer> right;

    public MedianPriorityQueue() {
        left = new PriorityQueue<>(Collections.reverseOrder());
        right = new PriorityQueue<>();
    }
}
```

```
public void add(int val) {
    // write your code here
    if(right.size() > 0 && val > right.peek()) right.add(val);
    else left.add(val);

    if(left.size() - right.size() >= 2) right.add(left.remove());
    else if(right.size() - left.size() >= 2) left.add(right.remove());
}
```

```
public int size() {
    // write your code here
    return left.size() + right.size();
}
```

```
public int peek() {
    // write your code here
    if(size() == 0) {
        System.out.println("Underflow");
        return -1;
    }

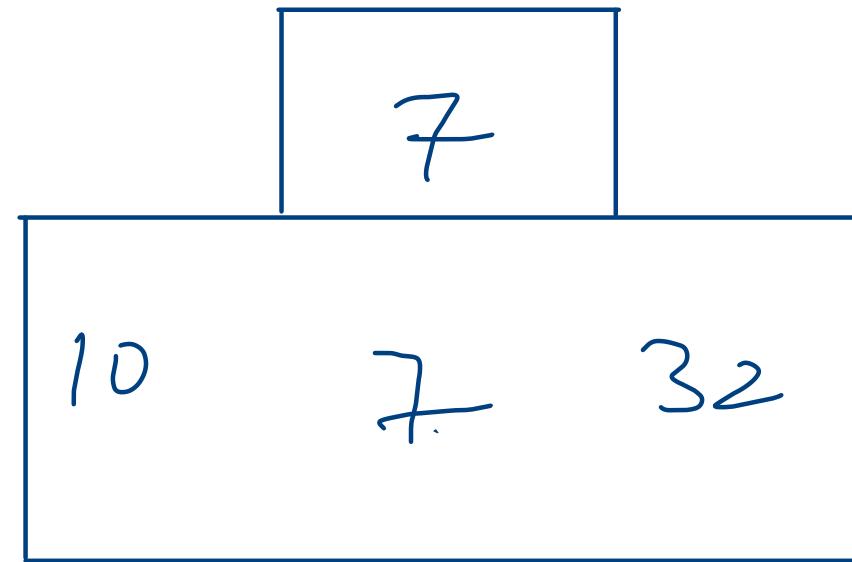
    if(left.size() < right.size()) return right.peek();
    return left.peek();
}
```

```
public int remove() {
    // write your code here
    if(size() == 0) {
        System.out.println("Underflow");
        return -1;
    }

    if(left.size() < right.size()) return right.remove();
    return left.remove();
}
```

Merge K Sorted Lists

10 20 30 40 50
↓
5 7 9 11 19 55 57
↓
1 2 3 4
↓
32 39



1 2 3 5

Code

```
public static class Pair implements Comparable<Pair>{
    int li;
    int di;
    int val;

    Pair() {
    }

    Pair(int li, int di, int val) {
        this.li = li;
        this.di = di;
        this.val = val;
    }

    public int compareTo(Pair other) {
        return this.val - other.val;
    }
}
```

```
public static ArrayList<Integer> mergeKSortedLists(ArrayList<ArrayList<Integer>> lists) {
    ArrayList<Integer> rv = new ArrayList<>();

    PriorityQueue<Pair> pq = new PriorityQueue<>();
    for(int i=0;i<lists.size();i++) {
        Pair p = new Pair(i,0,lists.get(i).get(0));
        pq.add(p);
    }

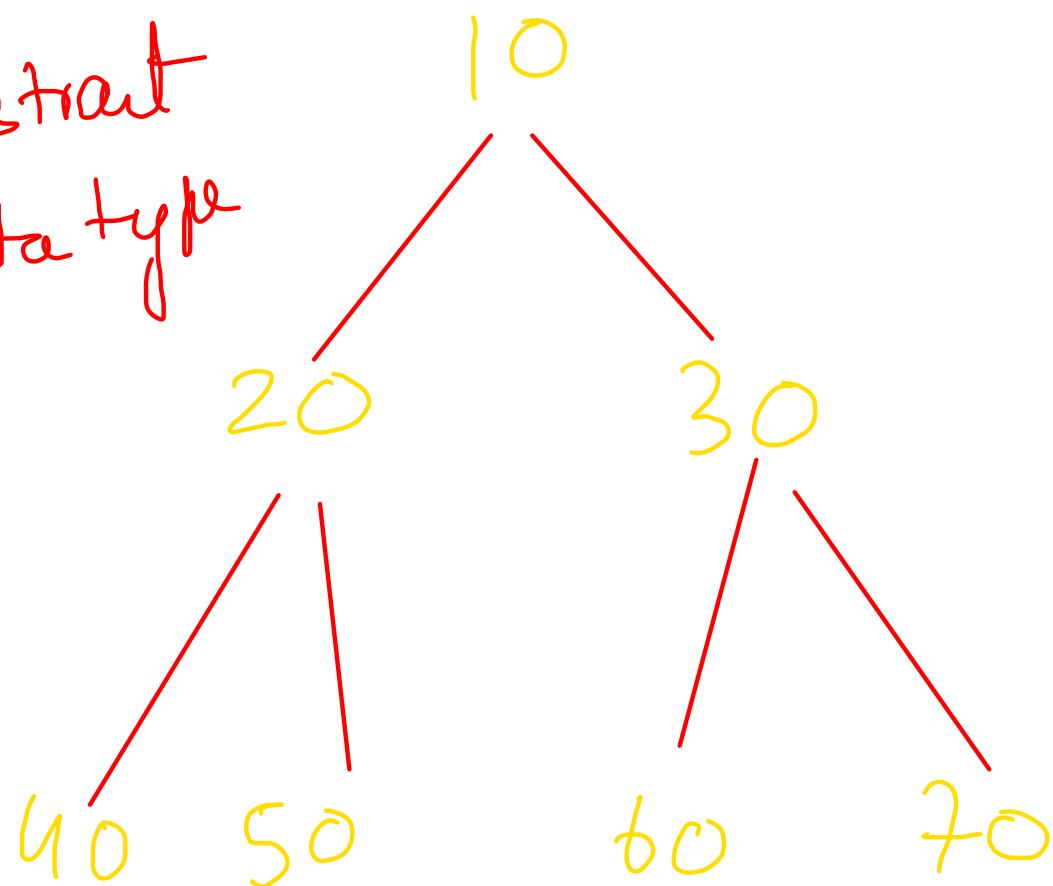
    while(pq.size() > 0) {
        Pair p = pq.remove();
        rv.add(p.val);
        p.di++;
        if(p.di < lists.get(p.li).size()) {
            p.val = lists.get(p.li).get(p.di);
            pq.add(p);
        }
    }

    return rv;
}
```

Write priority Queue using Heap \rightarrow Binary Heap (DS)

Heap: A BT data structure that follows 2 properties.

Abstract
Data type



- ① Heap Order Property
- ② Complete Binary Tree Property

~~What~~

Heap Order Property: Every parent has higher priority

than their children. There is no specification on whether left child's priority will be higher than right or vice versa.

Why Heap Order Property?

⇒ Because of the Heap Order Property, every parent will have greater priority than its children. Since root is the ancestor of all the nodes in a tree, root will have the highest priority. So, we will be able to achieve peek() in $O(1)$ time.

Aim

add

remove

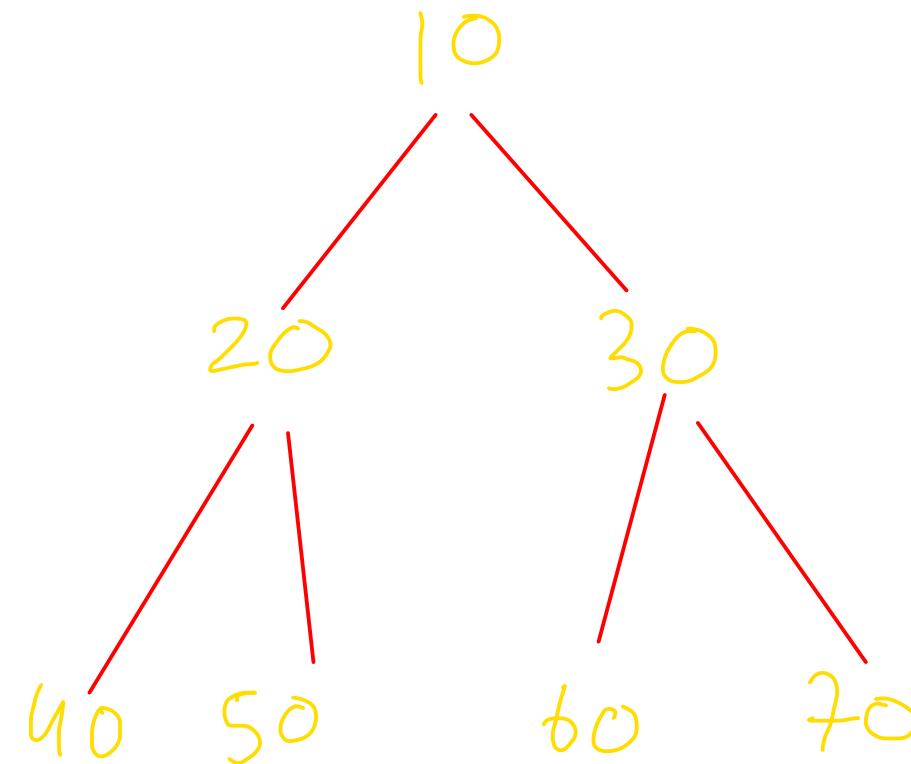


$O(\log_2 n)$

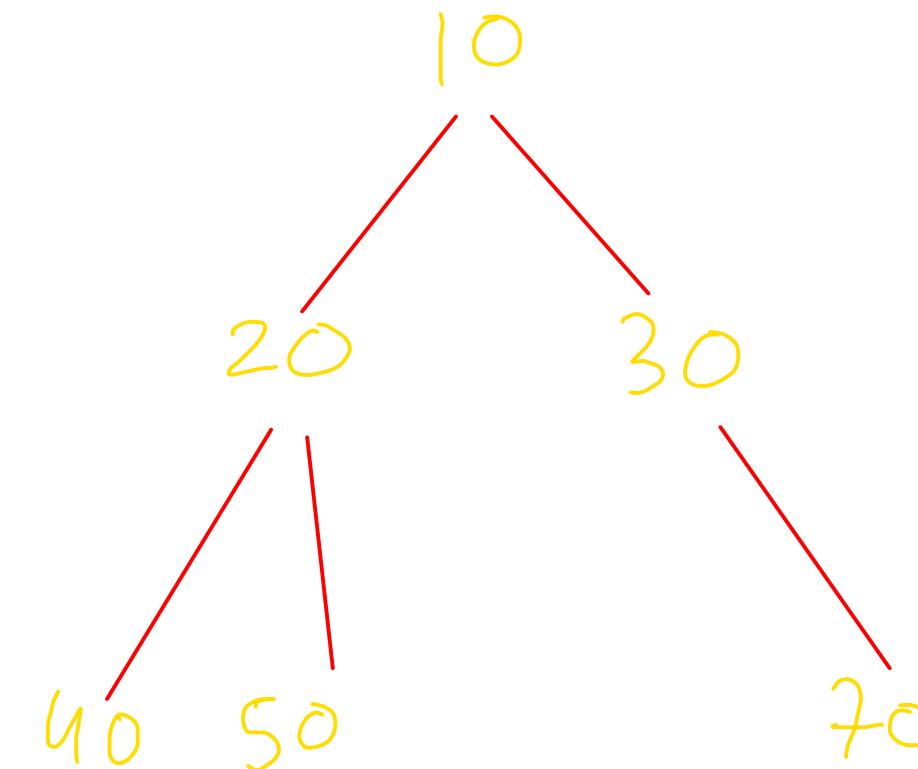
✓ peek $\Rightarrow O(1)$ (Achieved due to HOP)

What is complete binary tree property?

⇒ Either all the levels of a tree should be completely filled or $H-1$ levels should be completely filled & the last level (H^{th} level) should be filled from left to right.



Complete BT ✓



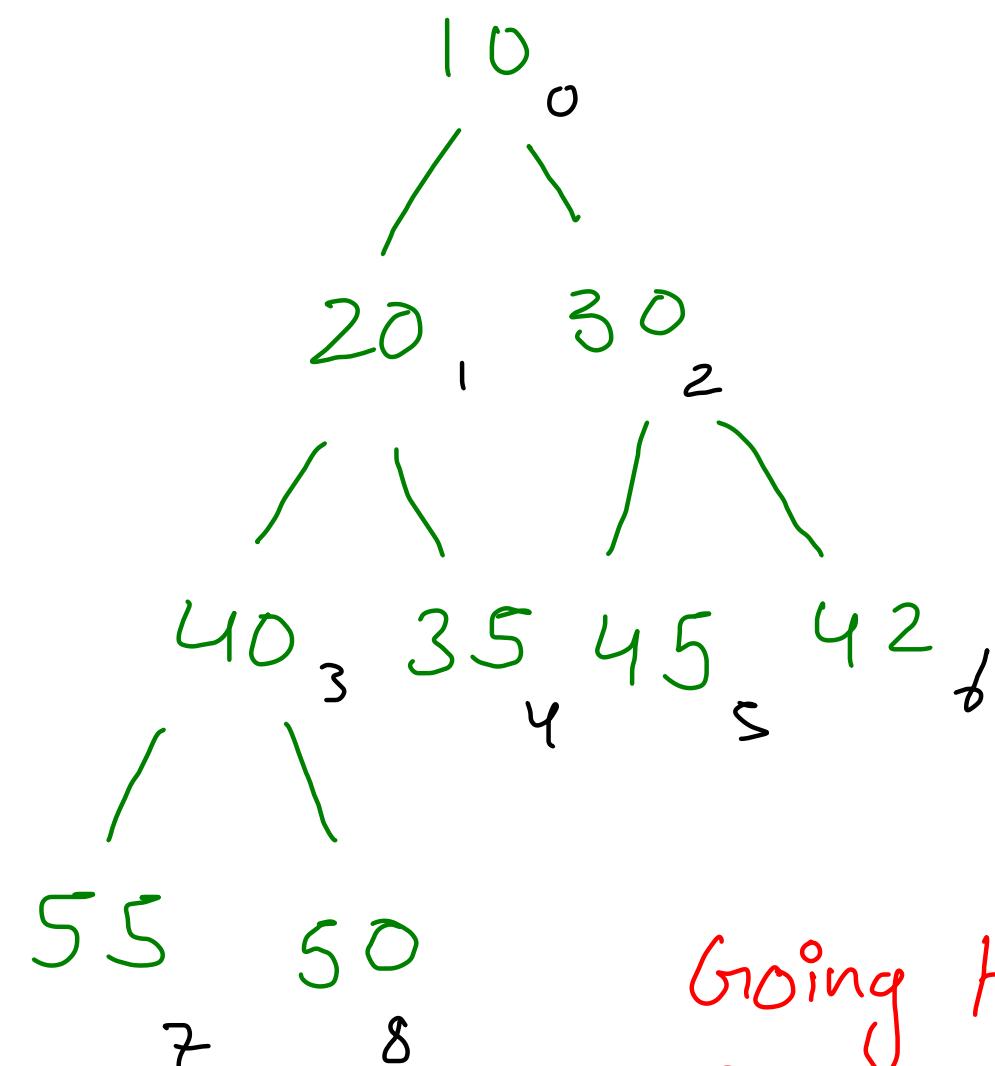
Not a Complete BT ✗

Why ??

To achieve add
& remove in
 $O(\log_2 N)$

for storage, heap uses an arraylist only internally.

10 20 30 40 35 45 42 55 50



$$\text{left child index} = 2 * \text{parent index} + 1$$

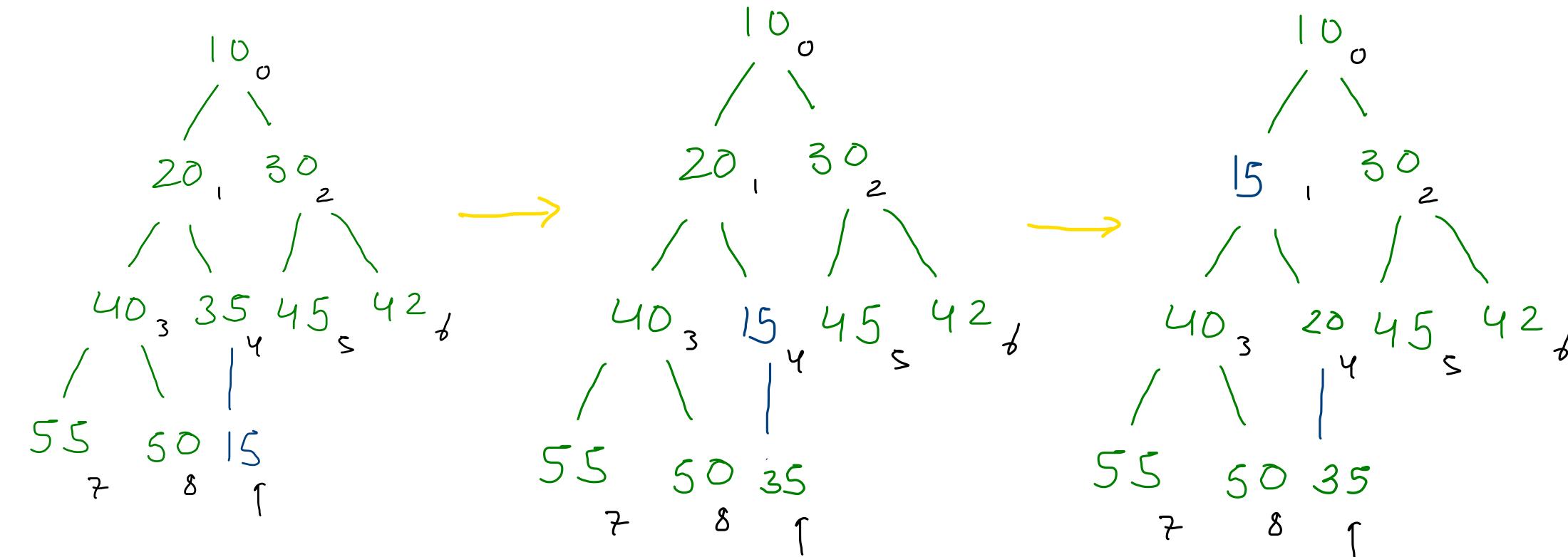
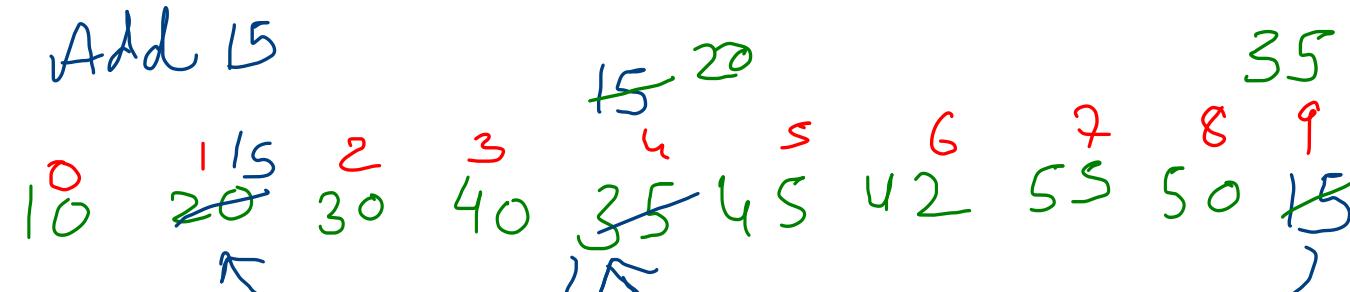
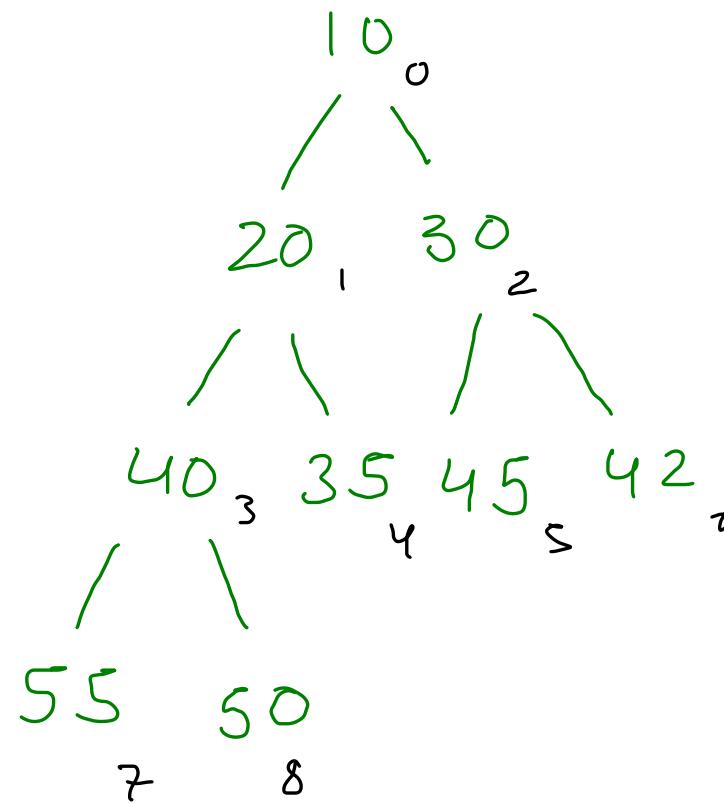
$$\text{right child index} = 2 * \text{parent index} + 2$$

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

Going towards the parent from the child
is the extra capability that we get on
seeing arraylist as a tree.

Add

10 20 30 40 35 45 42 55 50



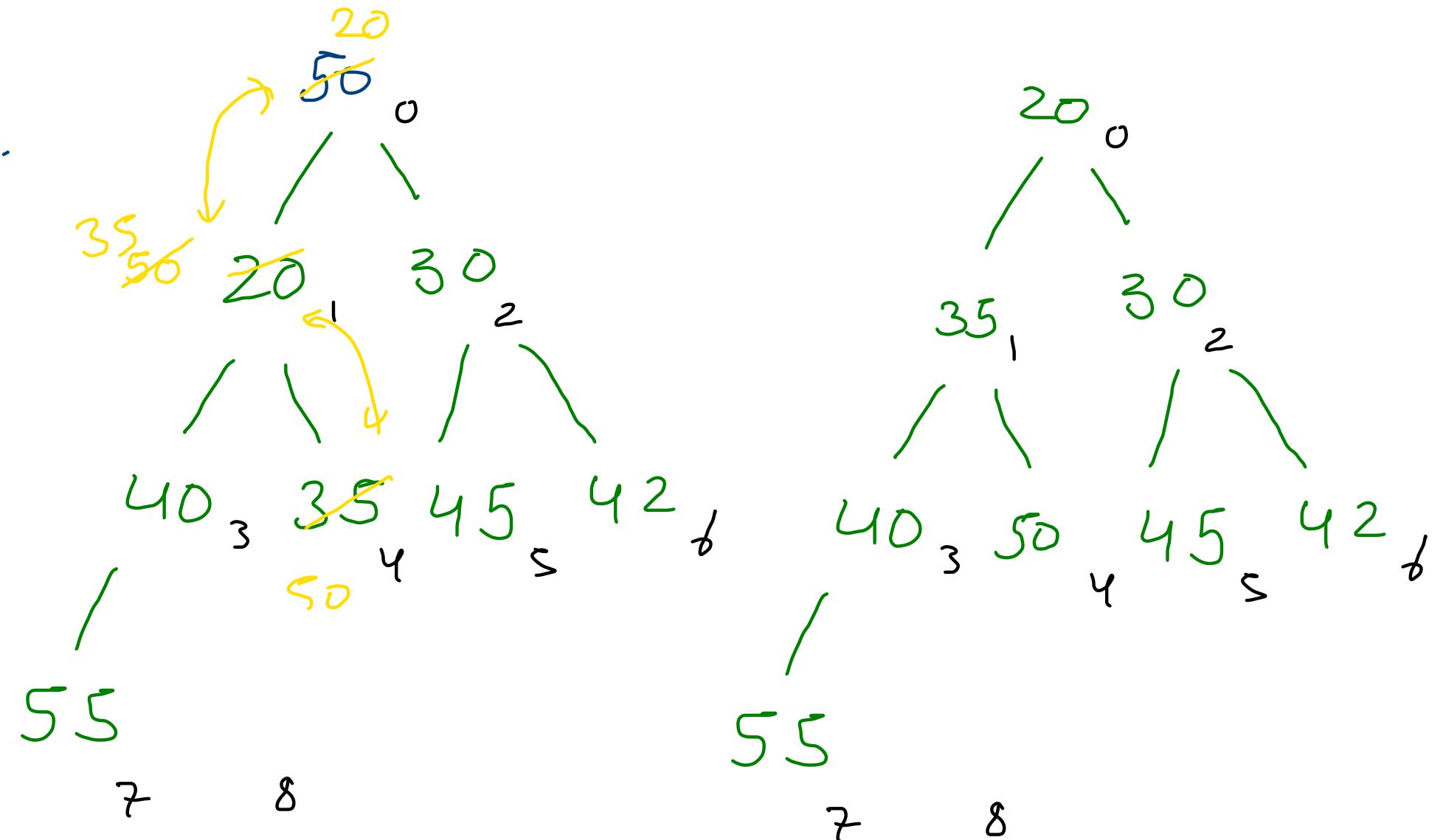
Complete Binary Tree \rightarrow AL \rightarrow idx \rightarrow parent \rightarrow upheapify \rightarrow Add $O(\log n)$

Remove

① Swap idx = 0 & last idx.

② Delete last element.

③ Down heapify.



```
public static class PriorityQueue {  
    ArrayList<Integer> data;  
  
    public PriorityQueue() {  
        data = new ArrayList<>();  
    }  
}
```

```
public void add(int val) {  
    // write your code here  
    data.add(val);  
    upheapify(data.size() - 1);  
}  
  
private void upheapify(int i) {  
  
    if(i == 0) {  
        return;  
    }  
  
    int pi = (i-1)/2;  
    if(data.get(i) < data.get(pi)) {  
        swap(i,pi);  
        upheapify(pi);  
    }  
}
```

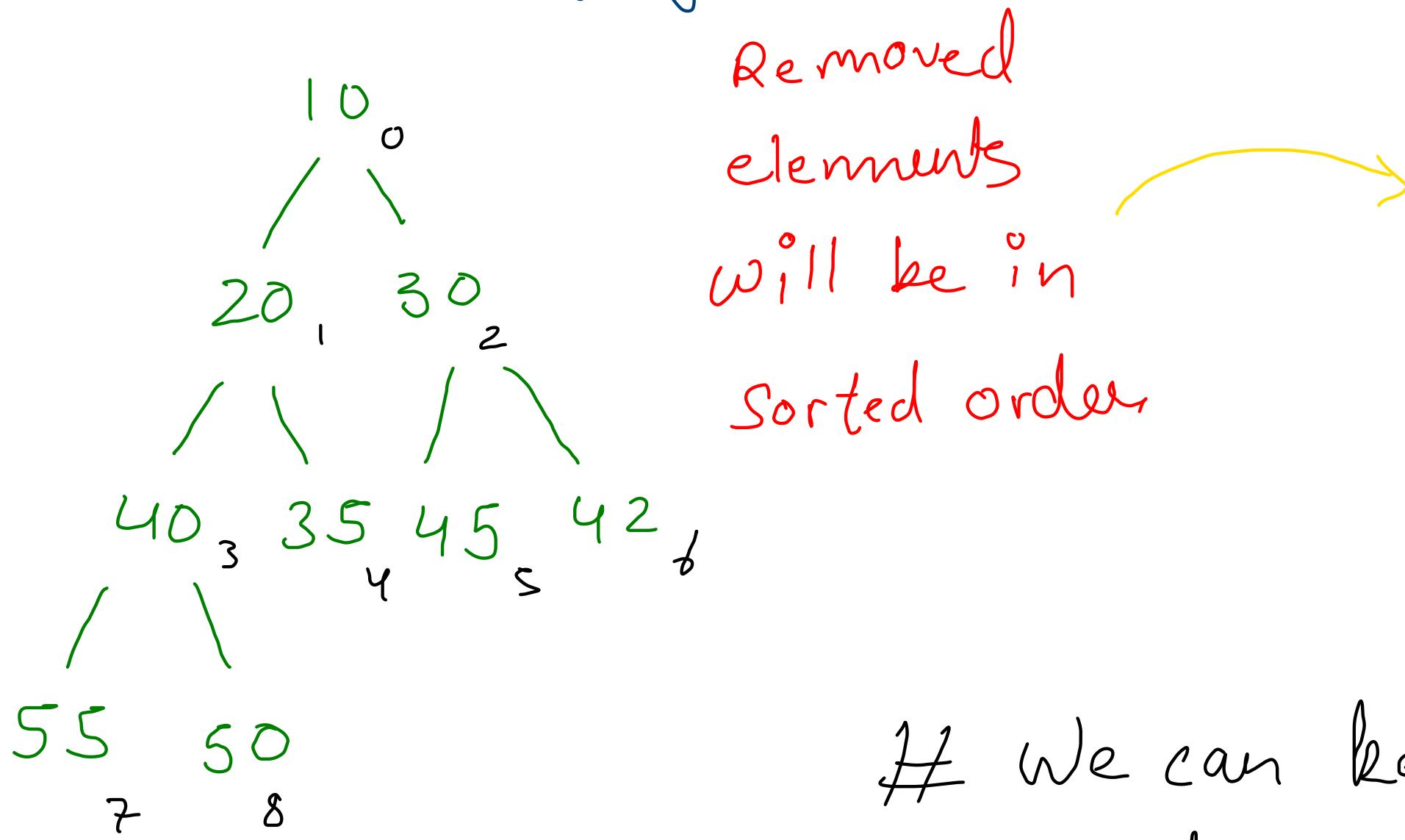
```
public int remove() {  
    // write your code here  
    if(this.size() == 0) {  
        System.out.println("Underflow");  
        return -1;  
    }  
  
    swap(0,data.size() - 1);  
    int val = data.remove(data.size() - 1);  
    downheapify(0);  
    return val;  
}  
  
private void downheapify(int pi) {  
  
    int mini = pi;  
  
    int li = 2*pi + 1;  
    if(li < data.size() && data.get(li) < data.get(mini)) {  
        mini = li;  
    }  
  
    int ri = 2*pi + 2;  
    if(ri < data.size() && data.get(ri) < data.get(mini)) {  
        mini = ri;  
    }  
  
    if(mini != pi) {  
        swap(pi,mini);  
        downheapify(mini);  
    }  
}
```

```
private 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() {  
    // write your code here  
    if(this.size() == 0) {  
        System.out.println("Underflow");  
        return -1;  
    }  
  
    return data.get(0);  
}  
  
public int size() {  
    // write your code here  
    return data.size();  
}
```

Heap Sort

{ By default not stable, stable banana padega }



| | |
|----|-----------------------|
| 10 | $T C = O(N * \log N)$ |
| 20 | $S C = O(N)$ |
| 30 | |
| 35 | |
| . | |
| . | |
| . | |

QuickSort → ~~stable~~

Heap Sort → stable

Merge Sort → stable

We can keep the element attached to the heap instead of removing it & the space complexity will be $O(1)$ in that case.

Heap Sort Code

```
public ArrayList<Integer> heapSort(){  
    ArrayList<Integer> sorted = new ArrayList<>();  
  
    while(this.size() > 0) {  
        int val = this.remove();  
        sorted.add(val);  
    }  
  
    return sorted;  
}
```



$O(n \log N)$ Time
 $O(n)$ Space

```

public static class PriorityQueue {
    ArrayList<Integer> data;
    private int size;

    public PriorityQueue() {
        data = new ArrayList<>();
        size = 0;
    }
}

```

```

public void add(int val) {
    // write your code here
    data.add(val);
    size++;
    upheapify(data.size() - 1);
}

```

```

private void upheapify(int i) {
    if(i == 0) {
        return;
    }

    int pi = (i-1)/2;
    if(data.get(i) < data.get(pi)) {
        swap(i,pi);
        upheapify(pi);
    }
}

```

```

private 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 remove() {
    // write your code here
    if(this.size() == 0) {
        System.out.println("Underflow");
        return -1;
    }

    swap(0,data.size() - 1);
    int val = this.peek();
    size--;
    downheapify(0);
    return val;
}

```

```

public ArrayList<Integer> inPlaceHeapSort() {
    while(size() > 0) {
        remove();
    }
    return data;
}

```

 $O(1)$ space

```

private void downheapify(int pi) {
    int mini = pi;

    int li = 2*pi + 1;
    if(li < size && data.get(li) < data.get(mini)) {
        mini = li;
    }

    int ri = 2*pi + 2;
    if(ri < size && data.get(ri) < data.get(mini)) {
        mini = ri;
    }

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

```

```

public int peek() {
    // write your code here
    if(this.size() == 0) {
        System.out.println("Underflow");
        return -1;
    }

    return data.get(0);
}

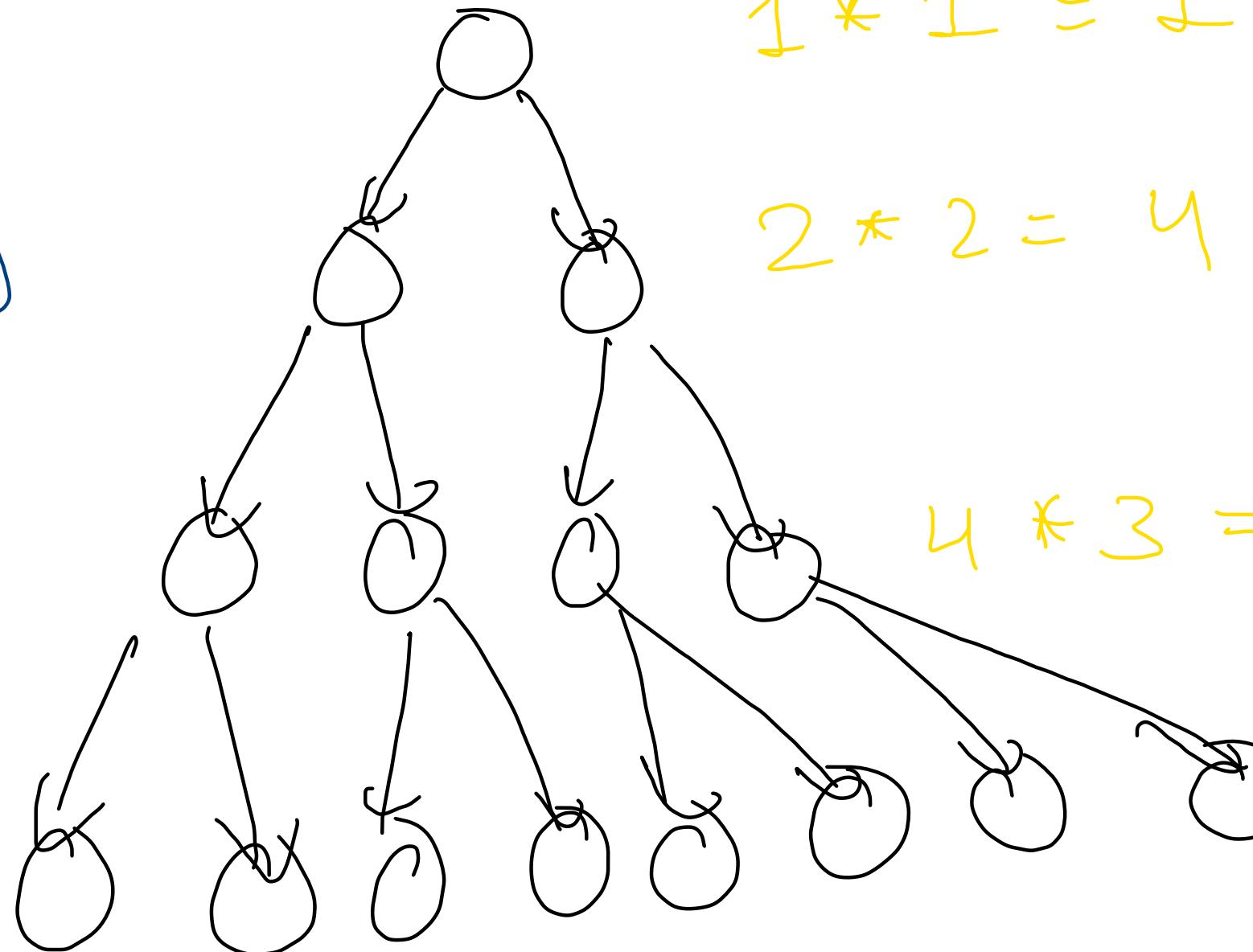
public int size() {
    // write your code here
    return size;
}

```

Inversion using Down Heapify

Up Hea^cify

TC



$$1 * 1 = 1$$

$$2 * 2 = 4$$

$$4 * 3 = 12$$

$$8 * 4 = 32$$

$$2^0 \times 1 + 2^1 \times 2 + 2^2 \times 3 + 2^3 \times 4$$

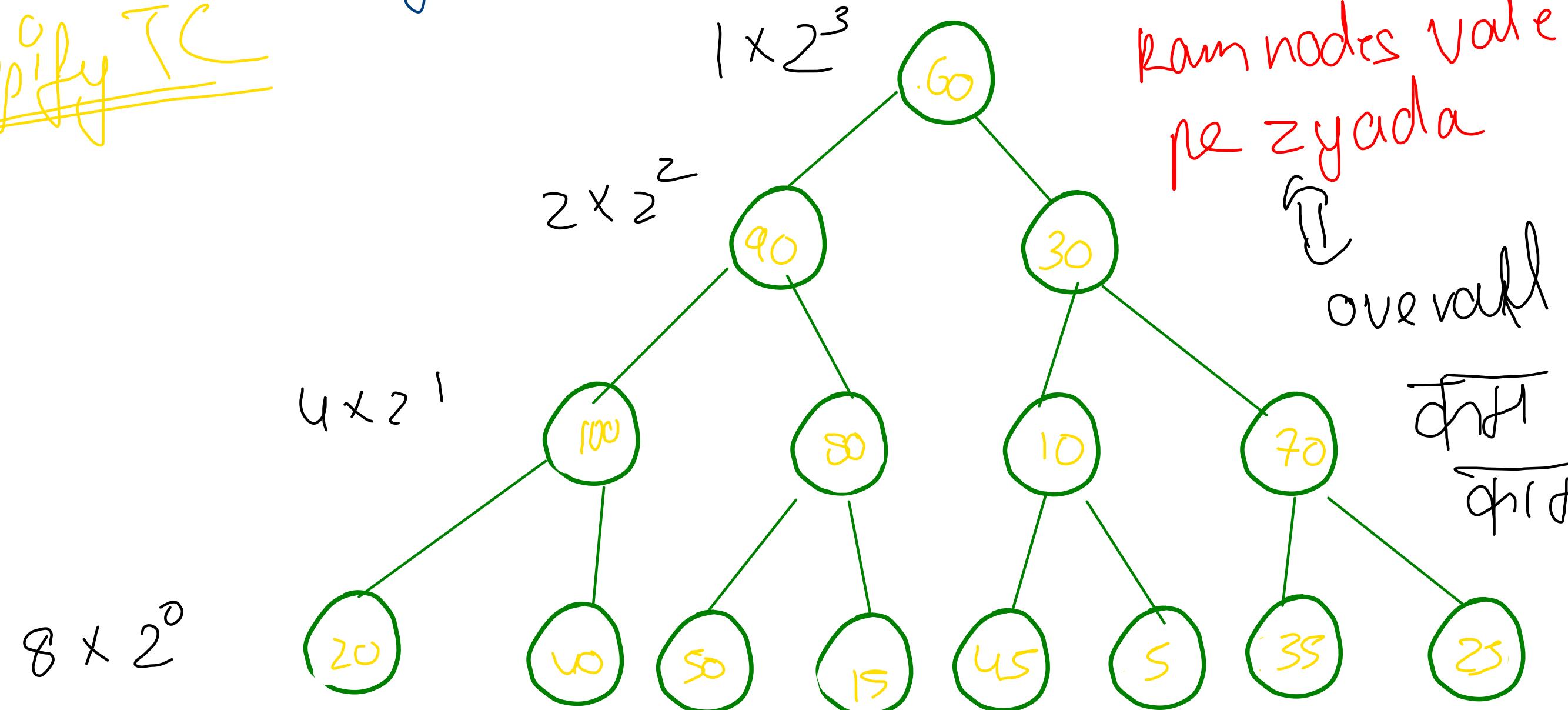
$\frac{N}{2}$ terms

$N \log N$

Insertion using Down Heapify

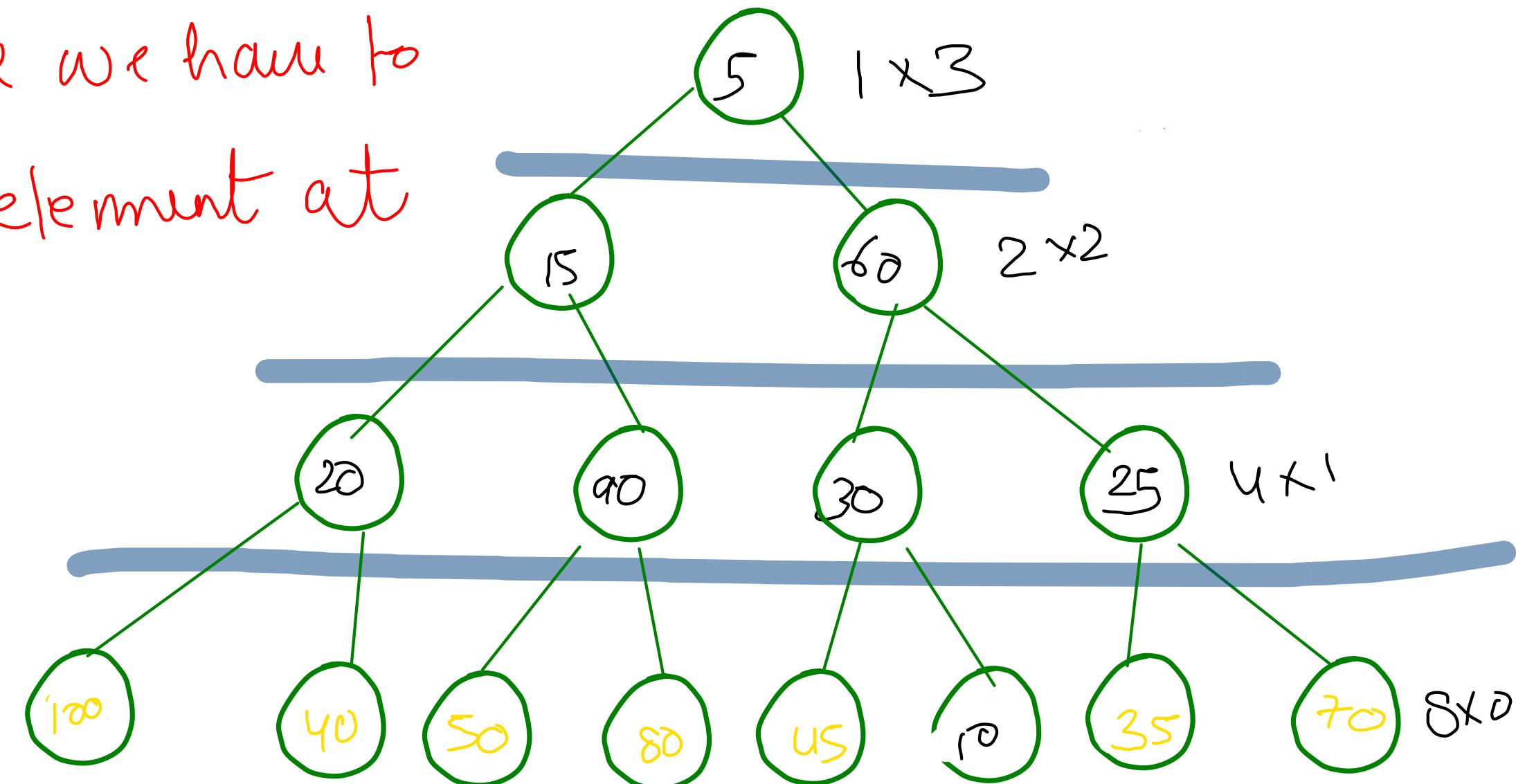
⇒ Pass the entire array as input in one go-

~~Down Heapify TC~~



down heapify will start from the bottom level because we have to bring the smallest element at the top-

No work at the last level.



Code for Insertion using downHeapify

```
// O(n) time for inserting N Elements
public PriorityQueue(int[] arr) {
    data = new ArrayList<>();
    for(int val : arr) {
        data.add(val);
        size++;
    }
    for(int i=(size()-1)/2;i>=0;i--) {
        downheapify(i);
    }
}
```

Efficient Heap

Constructor

Time Complexity of Insertion using down-heapify -

$$\text{① } DT(n) = 2^0 \cdot h + 2^1 \cdot (h-1) + 2^2 \cdot (h-2) + 2^3 \cdot (h-3) + \dots + 2^{h-1} \cdot \frac{1}{h}$$

AGP

$$\text{② } RT(n) = 2^1 \cdot h + 2^2 \cdot (h-1) + 2^3 \cdot (h-2) + \dots + 2^h \cdot 1$$

$$\textcircled{2} - \textcircled{1}$$

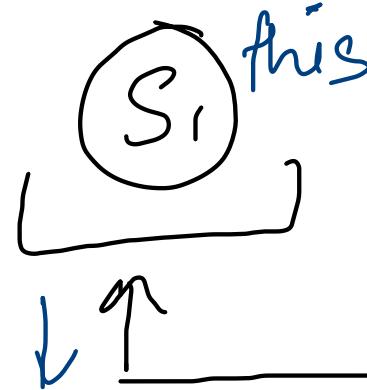
$$(RT - DT) = -2^0 \cdot h + \left\{ 2^1 \cdot 1 + 2^2 \cdot 1 + 2^3 \cdot 1 + \dots + 2^h \cdot 1 \right\}$$

$$T(n) = (2^h - 1) - 2^0 \cdot h = 2^{\log_2 n} - 1 - \log_2 n$$

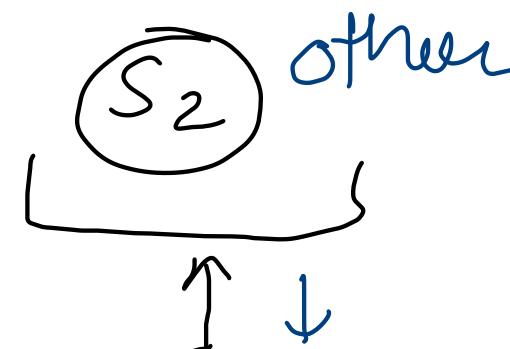
$$\Rightarrow T(n) = n - 1 - \log_2 n = O(n)$$

Comparable & Comparator

Comparable



Comparable



to give higher priority to
higher rollNo

$$\text{other.rollNo} - \text{this.rollNo}$$

$S_1 \downarrow \quad S_2 \uparrow$

$$\{\text{this.rollNo} - \text{other.rollNo}\}$$



-ve \Rightarrow we are giving priority to smaller element

Comparable Interface Example

```
public static class Student implements Comparable<Student>{
    int rollNo;
    int marks;
    int height;
    int weight;

    Student() {
    }

    Student(int rollNo, int marks, int height, int weight) {
        this.rollNo = rollNo;
        this.height = height;
        this.marks = marks;
        this.weight = weight;
    }

    public int compareTo(Student other) {
        //Smaller roll No -> Greater Priority
        return this.rollNo - other.rollNo;
    }

    @Override
    public String toString() {
        return ("rollNo = " + this.rollNo + " marks = " + this.marks + " height = " + this.height);
    }
}
```

```
public static void main(String[] args) throws Exception {
    PriorityQueue<Student> q = new PriorityQueue<>();
    q.add(new Student(30,91,100,70));
    q.add(new Student(40,90,101,71));
    q.add(new Student(50,100,102,72));
    q.add(new Student(60,89,103,73));

    while(q.size() > 0) {
        Student top = q.remove();
        System.out.println(top);
    }
}
```

Comparator Example

```
public static class StudentComparator implements Comparator<Student> {
    public int compare(Student s, Student other) {
        //Greater the marks-> Greater the priority
        return other.marks - s.marks;
    }
}
```

```
public static void main(String[] args) throws Exception {
    PriorityQueue<Student> q = new PriorityQueue<>(new StudentComparator());
    q.add(new Student(30,91,100,70));
    q.add(new Student(40,90,101,71));
    q.add(new Student(50,100,102,72));
    q.add(new Student(60,89,103,73));

    while(q.size() > 0) {
        Student top = q.remove();
        System.out.println(top);
    }
}
```

Genric PQ

```
public static class Student implements Comparable<Student> {
    int marks;
    int rollNo;
    int weight;
    int height;

    Student() {
    }

    @Override
    public String toString() {
        return ("rollNo = " + this.rollNo + " marks = " + this.marks + " height = " + this.height);
    }

    Student(int marks,int weight, int rollNo, int height) {
        this.marks = marks;
        this.weight = weight;
        this.height = height;
        this.rollNo = rollNo;
    }

    public int compareTo(Student other) {
        return other.marks - this.marks;
    }
}

public static class StudentRollNoComparator implements Comparator<Student> {
    public int compare(Student me, Student other) {
        return me.rollNo - other.rollNo;
    }
}
```

```
public static class PriorityQueue<K> {
    ArrayList<K> data;
    private int size;
    private Comparator<K> comp;

    public PriorityQueue() {
        data = new ArrayList<>();
        size = 0;
        comp = null;
    }

    public PriorityQueue(Comparator<K> comp) {
        this.comp = comp;
        data = new ArrayList<>();
        size = 0;
    }

    // O(n) time for inserting N Elements
    public PriorityQueue(K[] arr) {
        data = new ArrayList<>();

        for(K val : arr) {
            data.add(val);
            size++;
        }

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

```
public void add(K val) {  
    // write your code here  
    data.add(val);  
    size++;  
    upheapify(size() - 1);  
}  
}
```

```
private void downheapify(int pi) {  
  
    int mini = pi;  
  
    int li = 2*pi + 1;  
    if(li < size && isSmaller(li,mini) == true) {  
        mini = li;  
    }  
  
    int ri = 2*pi + 2;  
    if(ri < size && isSmaller(ri,mini) == true) {  
        mini = ri;  
    }  
  
    if(mini != pi) {  
        swap(pi,mini);  
        downheapify(mini);  
    }  
}
```

```
public K peek() {  
    // write your code here  
    if(this.size() == 0) {  
        System.out.println("Underflow");  
        return null;  
    }  
}
```

```
return data.get(0);
```

```
public int size() {  
    // write your code here  
    return size;  
}
```

```
private boolean isSmaller(int i1,int i2) {  
    K obj1 = data.get(i1);  
    K obj2 = data.get(i2);  
  
    if(comp == null) {  
        Comparable obj1C = (Comparable)obj1;  
        Comparable obj2C = (Comparable)obj2;  
  
        if(obj1C.compareTo(obj2C) < 0) {  
            return true;  
        }  
  
        return false;  
    } else {  
        if(comp.compare(obj1,obj2) < 0) return true;  
        else return false;  
    }  
}
```

```
public ArrayList<K> heapSort() {  
    ArrayList<K> sorted = new ArrayList<>();  
  
    while(this.size() > 0) {  
        sorted.add(this.remove());  
    }  
  
    return sorted;  
}  
  
public ArrayList<K> inPlaceHeapSort() {  
    while(size() > 0) {  
        remove();  
    }  
  
    return data;  
}
```

```
private void upheapify(int i) {  
  
    if(i == 0) {  
        return;  
    }  
  
    int pi = (i-1)/2;  
    if(isSmaller(i,pi)) {  
        swap(i,pi);  
        upheapify(pi);  
    }  
}
```

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

```
public K remove() {  
    // write your code here  
    if(this.size() == 0) {  
        System.out.println("Underflow");  
        return null;  
    }  
  
    swap(0,size() - 1);  
    K val = data.get(size - 1);  
    size--;  
    downheapify(0);  
    return val;  
}
```

```
public static void main(String[] args) throws Exception {
    PriorityQueue<Student> pq = new PriorityQueue<>(new StudentRollNoComparator());
    pq.add(new Student(80,75,2,103));
    pq.add(new Student(70,70,1,150));
    pq.add(new Student(20,60,8,120));
    pq.add(new Student(90,90,4,115));

    while(pq.size() > 0) {
        Student top = pq.remove();
        System.out.println(top);
    }
}
```

HashMap Construction

Array < LinkedList >



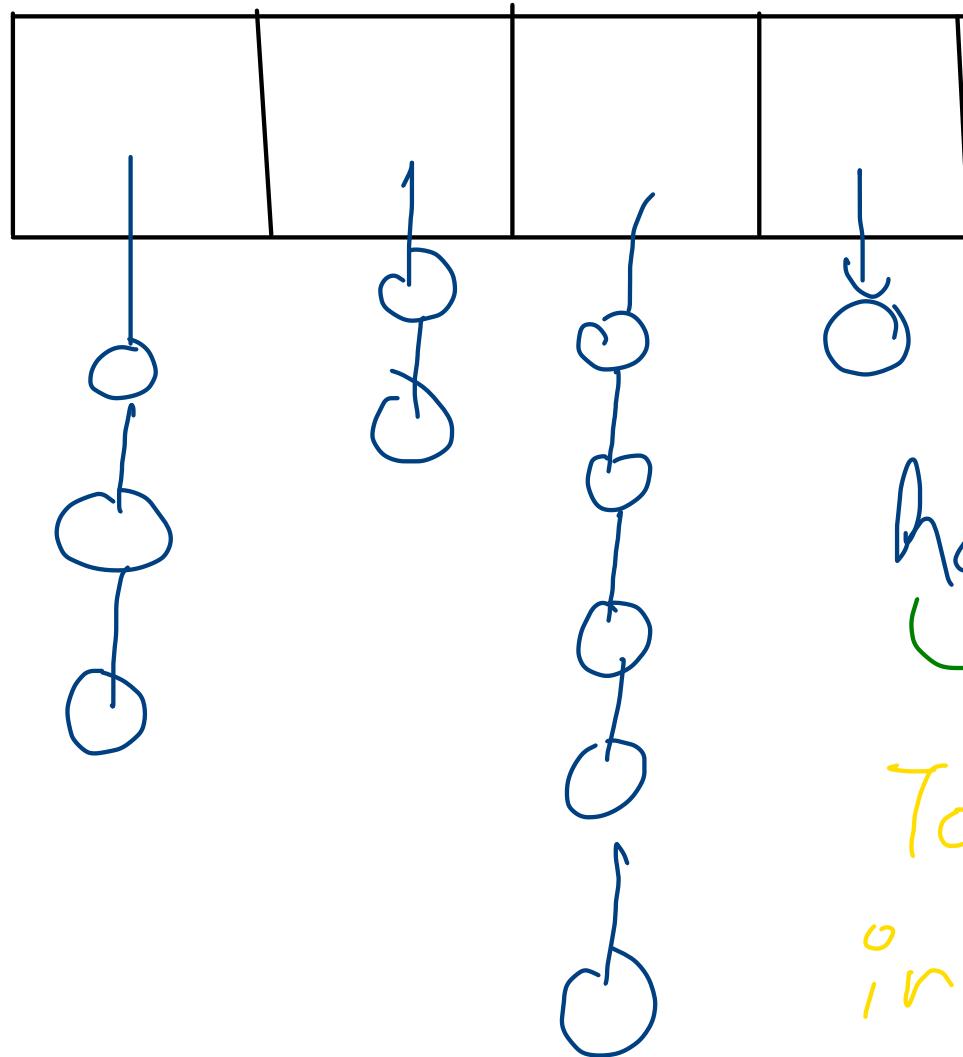
HashMap Node

hash function

2 different keys can have
the same hash function

but one key cannot have

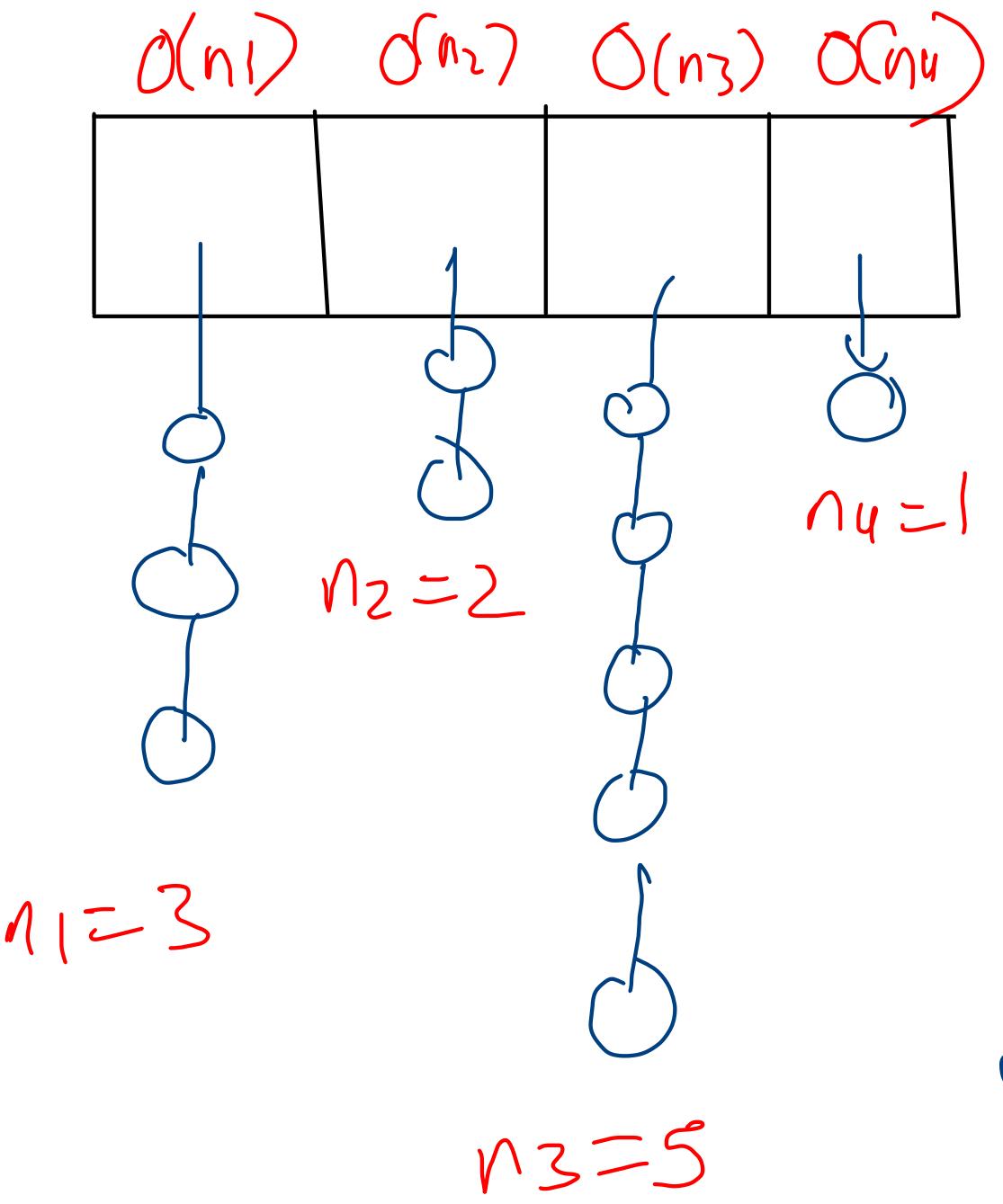
2 diff hash func & a key will have a unique hash func



HashCode is there for every predefined class in Java -
User defined → makes yourself

hash for % 4

To get the data
in one of the 4
buckets.



Put $TC = O(\lambda)$ $\approx O(2) \approx \text{constant}$
 \downarrow
 loading factor

$$\lambda = \text{Total nodes} / \text{No of buckets} = 11/4 = 2$$

\hookrightarrow Avg nodes per bucket.

loading factor = $\frac{\text{No of nodes}}{\text{No of Buckets}}$ ↑
 inc with inc
 TC of put

Worst case TC is $O(N)$
 if all nodes belong to L1. Hence, with inc no of nodes, no of buckets

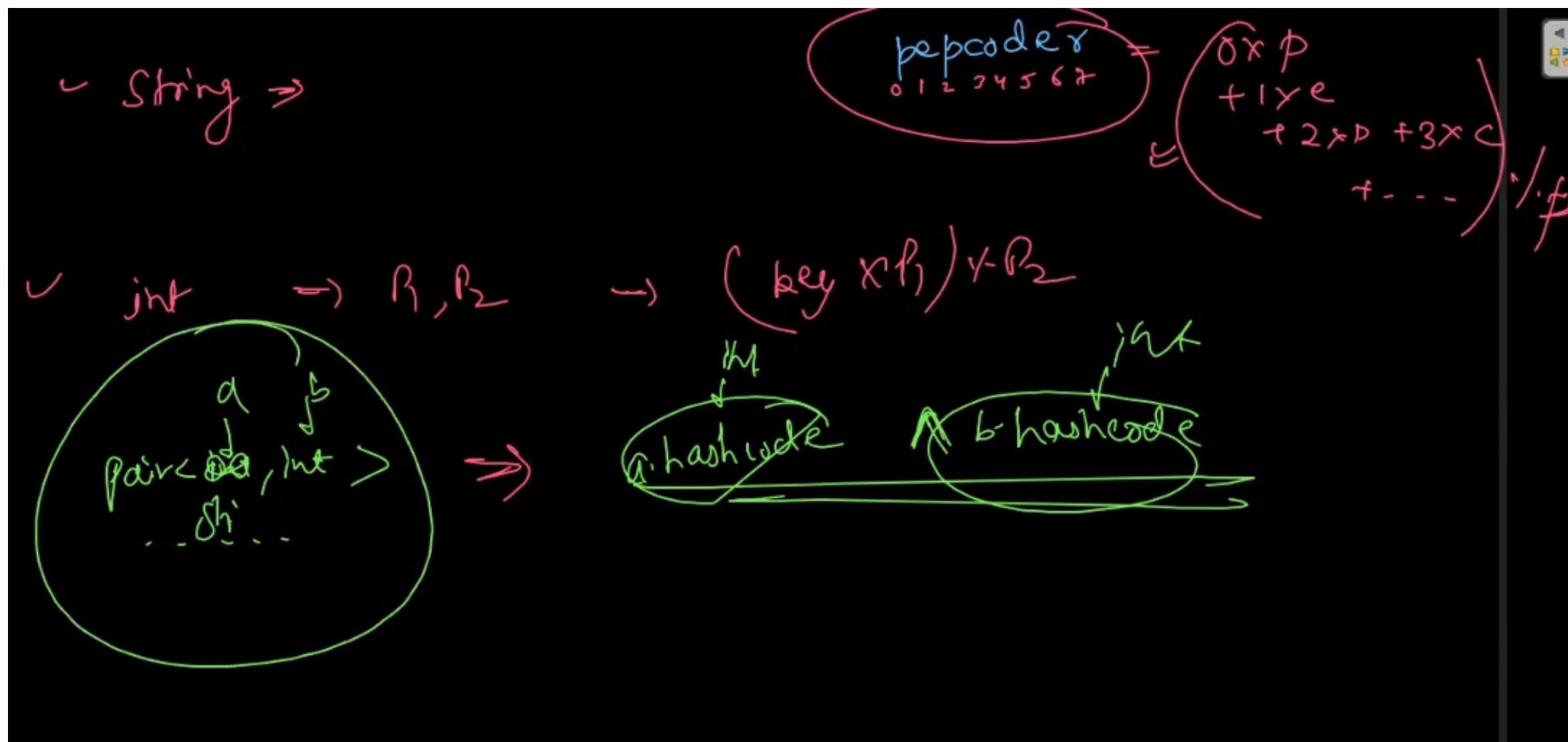
should also be increased.

Collisions should be minimized \rightarrow create such a hash fun

(good hash fun)

else

(bad hash function)



```
public static class HashMap<K, V> {

    private class HMNode {}

    private LinkedList<HMNode>[] buckets;
    private int noOfNodes;
    private int noOfBuckets;
    private double loadingFactor;

    public HashMap() {
        noOfBuckets = 4;
        noOfNodes = 0;
        loadingFactor = 0.0;
        init();
    }

    public void init() {
        buckets = new LinkedList[noOfBuckets];

        for(int i=0;i<noOfBuckets;i++) {
            buckets[i] = new LinkedList<>();
        }
    }
}
```

```
public int getBucketID(K key) {
    //O(1)
    int hashCode = key.hashCode();
    int bucketID = (Math.abs(hashCode)) % noOfBuckets;
    return bucketID;
}

public HMNode getData(int bucketID, K key) {
    for(HMNode node : buckets[bucketID]) {
        if(node.key.equals(key) == true) {
            return node; //data already exist
            //return its index in the LL
        }
    }
    return null; //data does not exist
}
```

```
public void put(K key, V value) throws Exception {  
    // write your code here  
    //O(1)  
    int bucketIdx = getBucketID(key);  
    HMNode data = getData(bucketIdx, key);  
  
    if(data == null) {  
        //Insertion (does not exist)  
        double newLoadingFactor = (noOfNodes + 1.0) / noOfBuckets;  
  
        if(newLoadingFactor > 2.0) {  
            //rehashing  
            LinkedList<HMNode>[] oldBuckets = buckets;  
            noOfBuckets = 2 * noOfBuckets;  
            init();  
  
            for(int i=0;i<oldBuckets.length;i++) {  
                for(HMNode node: oldBuckets[i]) {  
                    int bucketID = getBucketID(node.key);  
                    buckets[bucketID].addLast(node);  
                }  
            }  
            int newBucketID = getBucketID(key);  
            HMNode node = new HMNode(key,value);  
            buckets[newBucketID].addLast(node);  
            noOfNodes++;  
            loadingFactor = (noOfNodes * 1.0) / noOfBuckets;  
        } else {  
            //Updation (key already exist)  
            data.value = value;  
        }  
    }  
}
```

```
public V get(K key) throws Exception {  
    // write your code here  
    int bucketID = getBucketID(key);  
    HMNode data = getData(bucketID, key);  
  
    if(data != null) {  
        return data.value;  
    } else {  
        return null;  
    }  
}  
  
public boolean containsKey(K key) {  
    // write your code here  
    int bucketID = getBucketID(key);  
    HMNode data = getData(bucketID, key);  
  
    if(data != null) return true;  
    else return false;  
}  
  
public V remove(K key) throws Exception {  
    // write your code here  
    int bucketID = getBucketID(key);  
    HMNode data = getData(bucketID, key);  
  
    if(data == null) return null;  
    V value = data.value;  
    buckets[bucketID].remove(data);  
    noOfNodes--;  
    loadingFactor = (noOfNodes * 1.0) / noOfBuckets;  
    return value;  
}
```

```
public ArrayList<K> keyset() throws Exception {
    // write your code here
    ArrayList<K> keys = new ArrayList<>();

    for(int i=0;i<noOfBuckets;i++) {
        for(HMNode node : buckets[i]) {
            keys.add(node.key);
        }
    }

    return keys;
}

public int size() {
    // write your code here
    return noOfNodes;
}

public void display() {
    System.out.println("Display Begins");
    for (int bi = 0; bi < buckets.length; bi++) {
        System.out.print("Bucket" + bi + " ");
        for (HMNode node : buckets[bi]) {
            System.out.print( node.key + "@" + node.value + " ");
        }
        System.out.println(".");
    }
    System.out.println("Display Ends");
}
```