

Hashmap

- » Hashmap - Introduction
- «/» Highest Frequency Character
- «/» Get Common Elements - 1
- «/» Get Common Elements - 2
- «/» Longest Consecutive Sequence Of Elements
- «/» Write Hashmap

Heap/Priority Queue

- » Heaps - Introduction And Usage
- «/» K Largest Elements
- » Efficient Heap Constructor
- «/» Write Priority Queue Using Heap
- «/» Sort K-sorted Array
- » Heap - Comparable V/s Comparator
- «/» Merge K Sorted Lists
- «/» Median Priority Queue

#HashMap

Insert → put $\Rightarrow O(1)$

Update

Delete \rightarrow remove $\Rightarrow O(1)$

Read \rightarrow get $\Rightarrow O(1)$

If key exist then
return value
else return null

Display

size $\Rightarrow O(1)$

contains key \rightarrow find $\Rightarrow O(1)$

keyset \rightarrow keys

Key \rightarrow value

e.g. IPL teams \rightarrow trophies count
String \rightarrow int

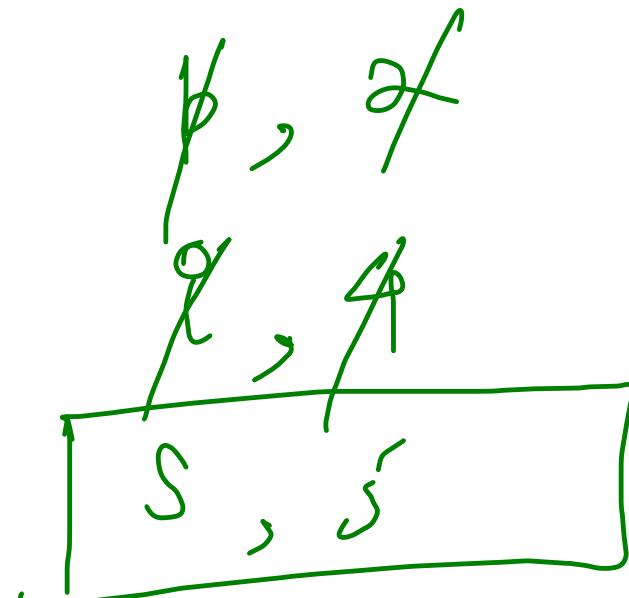
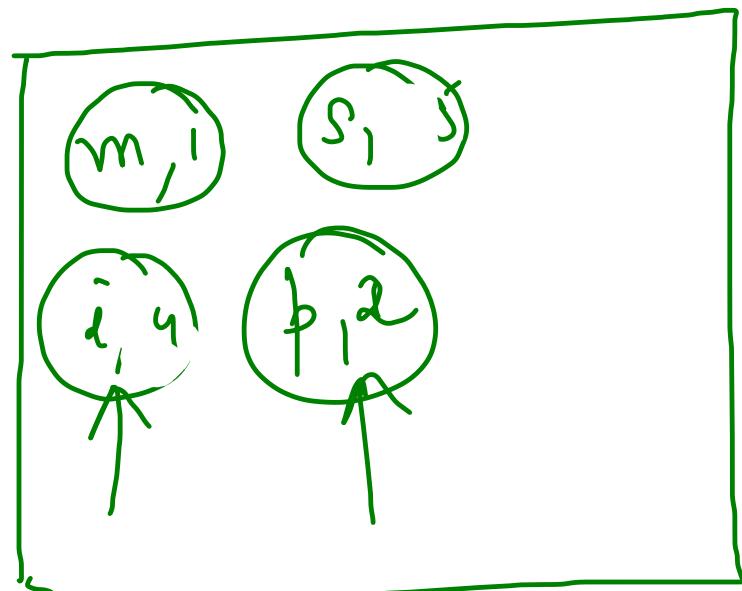
{ CSK \rightarrow 4
MI \rightarrow 5
SRH \rightarrow 2
DC \rightarrow 0

e.g. countries \rightarrow population
String \rightarrow integer

{ India \rightarrow 130
USA \rightarrow 90
China \rightarrow 200

Highest Frequency Character

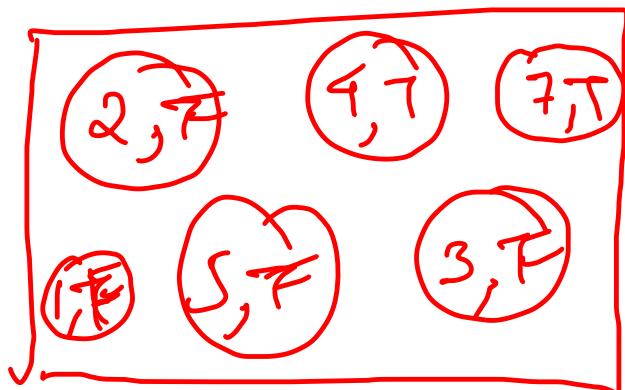
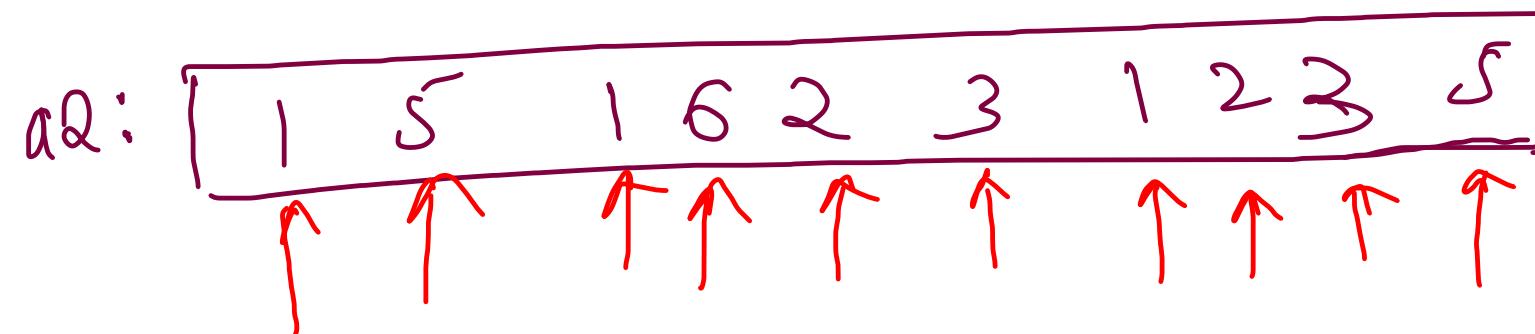
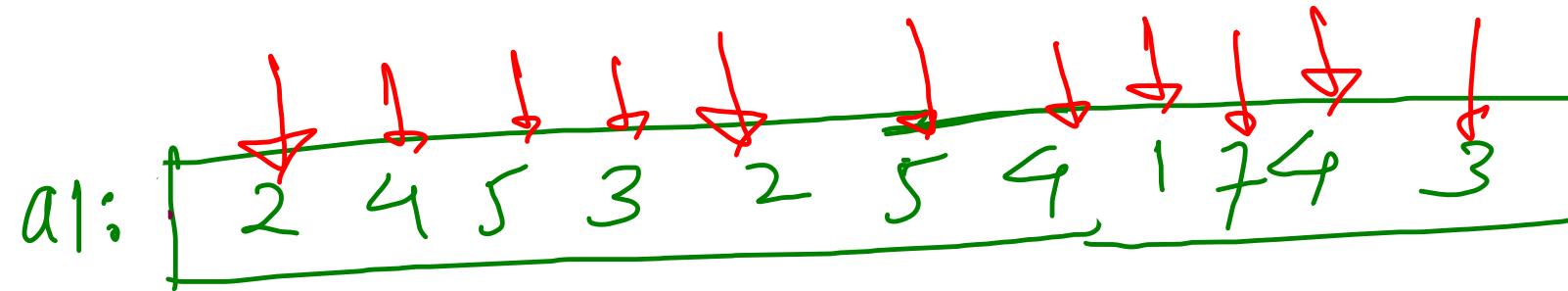
m i s s o s i p b o x f



```
HashMap<Character, Integer> freq = new HashMap<>();  
  
for(int i=0; i<str.length(); i++){  
    char ch = str.charAt(i);  
    if(freq.containsKey(ch)){  
        int oldFreq = freq.get(ch);  
        freq.put(ch, oldFreq + 1);  
    }  
    else {  
        freq.put(ch, 1);  
    }  
}  
  
char ch = str.charAt(0);  
int maxFreq = freq.get(ch);  
  
for(Character key: freq.keySet()){  
    int currFreq = freq.get(key);  
  
    if(currFreq > maxFreq){  
        ch = key;  
        maxFreq = currFreq;  
    }  
}  
  
System.out.println(ch);
```

The code is annotated with time complexities:
- The first loop is labeled $O(N)$.
- The second loop is labeled $O(\text{Keys})$.
- The entire algorithm is labeled $= O(256)$.

Get Common Elements ↗



$O(n)$

```
HashMap<Integer, Boolean> hm = new HashMap<>();
for(int i=0; i<n1; i++)
    hm.put(arr1[i], true);

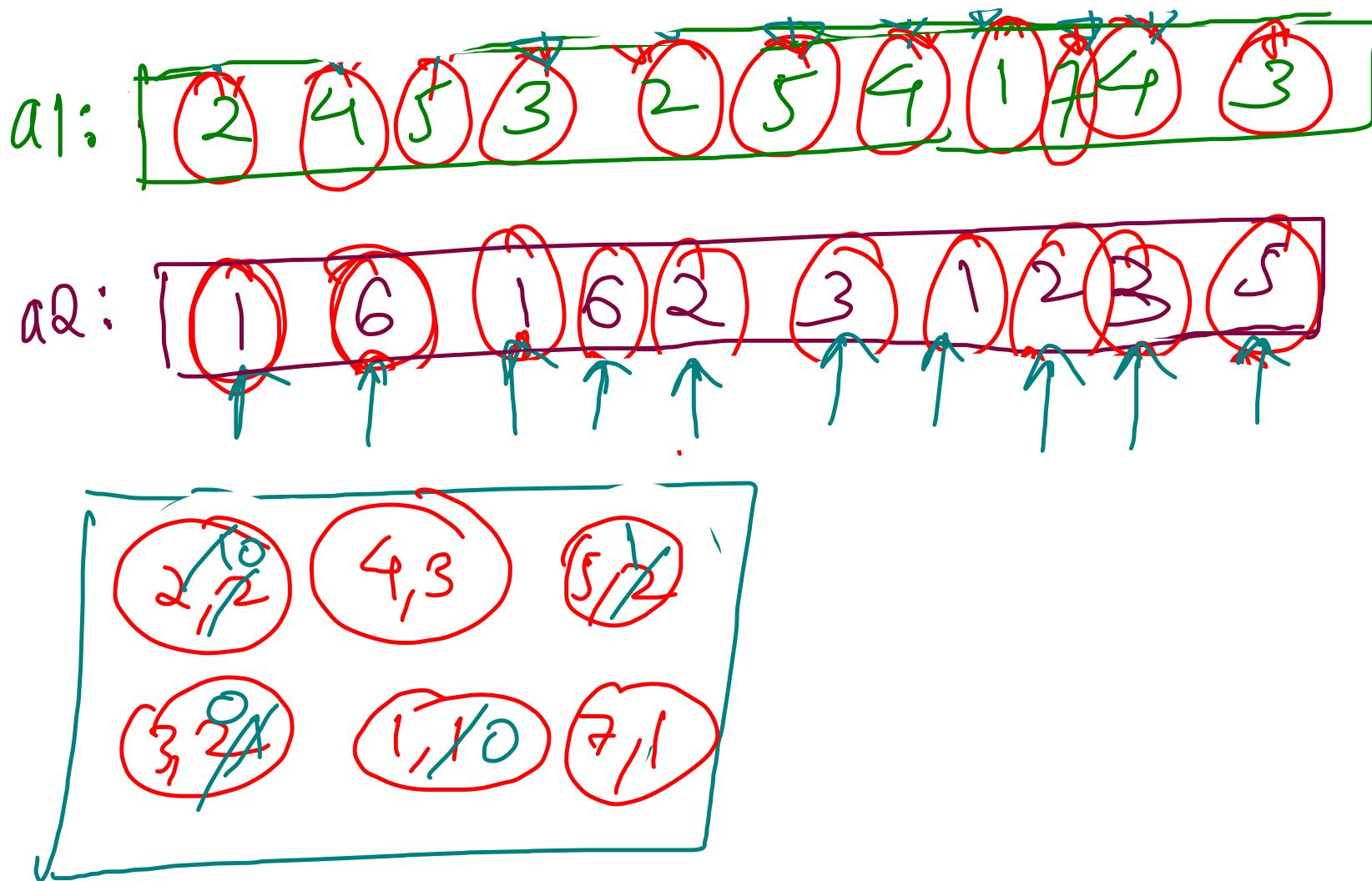
for(int i=0; i<n2; i++){
    if(hm.containsKey(arr2[i]) && hm.get(arr2[i])){
        System.out.println(arr2[i]);
        hm.put(arr2[i], false);
    }
}
```

$\mathcal{O}(n)$

$\langle \text{Integer}, \text{Boolean} \rangle$

$[O(n_1 + n_2)]$

Get Common Elements - 2

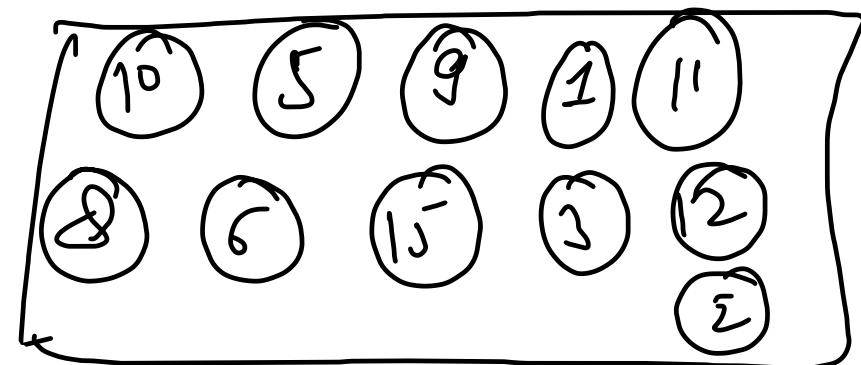
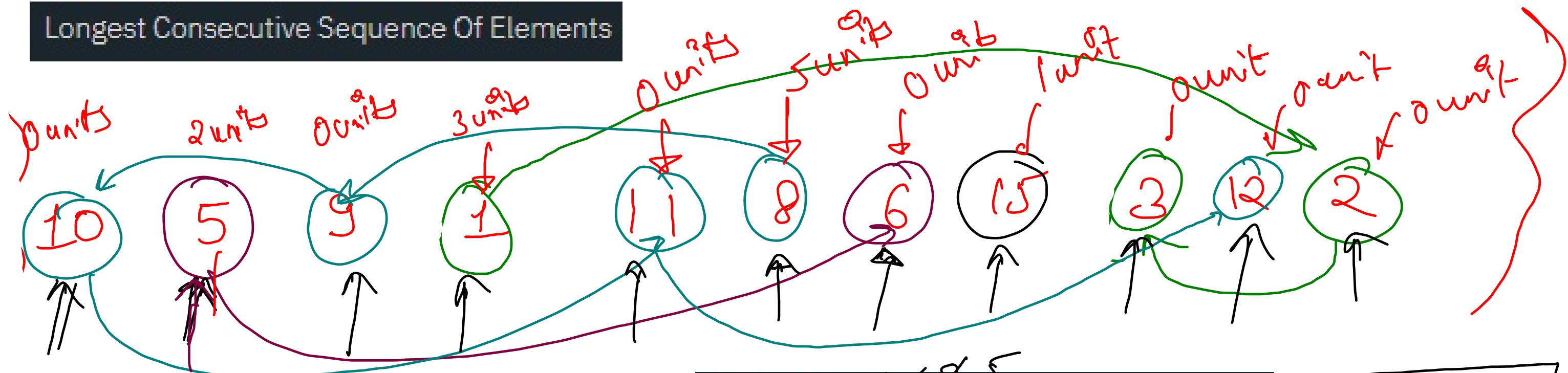


1, 2, 3, 2, 3, 5

```
HashMap<Integer, Integer> hm = new HashMap<>();
for(int i=0; i<n1; i++){
    if(hm.containsKey(arr1[i])){
        hm.put(arr1[i], hm.get(arr1[i]) + 1);
    } else {
        hm.put(arr1[i], 1);
    }
}

for(int i=0; i<n2; i++){
    if(hm.containsKey(arr2[i]) && hm.get(arr2[i]) > 0){
        System.out.println(arr2[i]);
        hm.put(arr2[i], hm.get(arr2[i]) - 1);
    }
}
```

Longest Consecutive Sequence Of Elements



```

int maxChain = 0; // 8
int startingPt = 0; // 8 X 8

for(Integer key: hm.keySet()){

    if(hm.containsKey(key - 1) == false){
        // chain starting pt

        int length = 1;
        while(hm.containsKey(key + length) == true){
            length++;
        }

        if(length > maxChain){
            maxChain = length;
            startingPt = key;
        }
    }
}

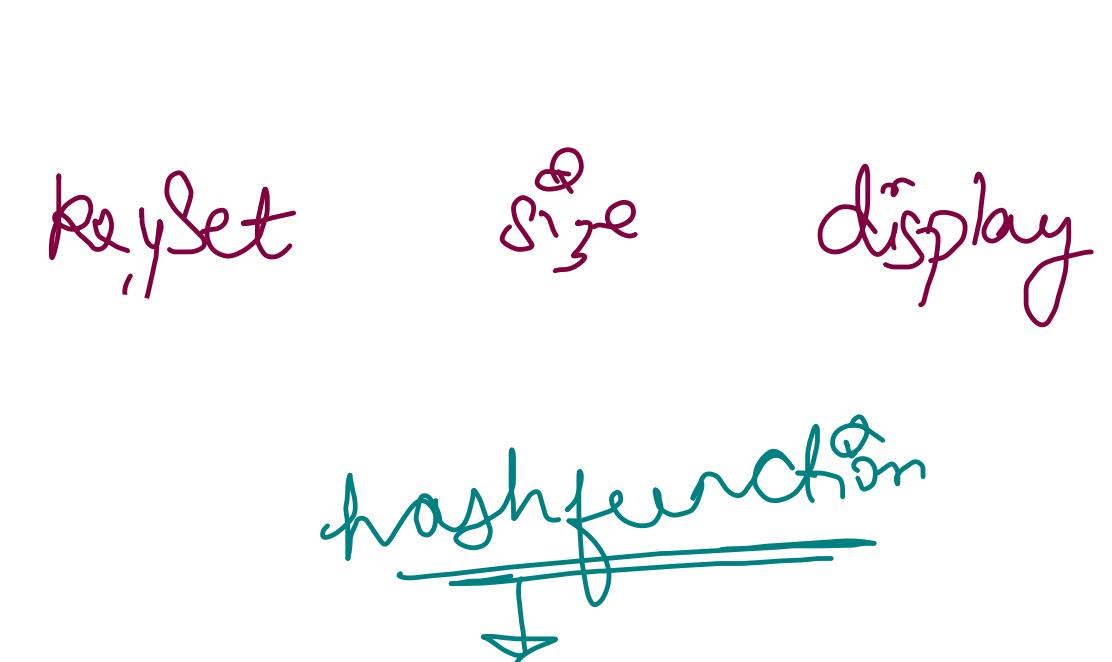
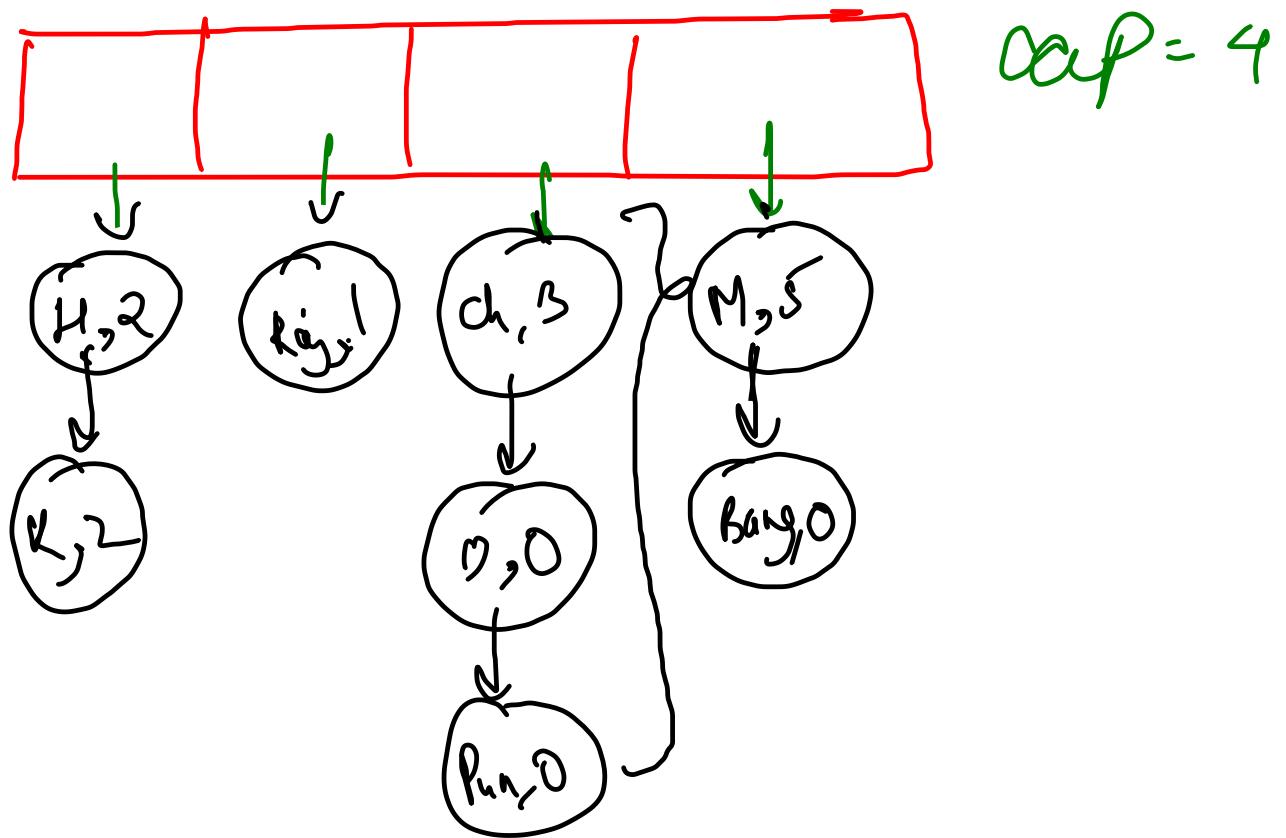
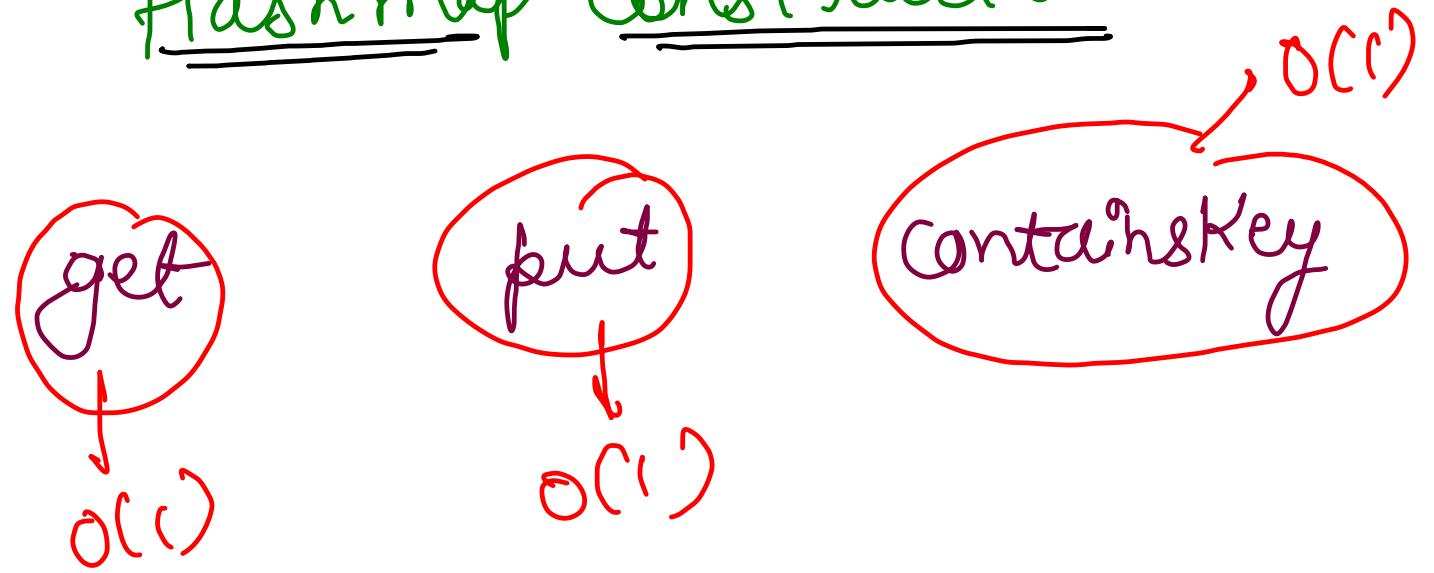
```

$1 \rightarrow 2 \rightarrow 3$
 $8 \rightarrow 9 \rightarrow 10 \rightarrow 11 \rightarrow 12$

$5 \rightarrow 6$
 15

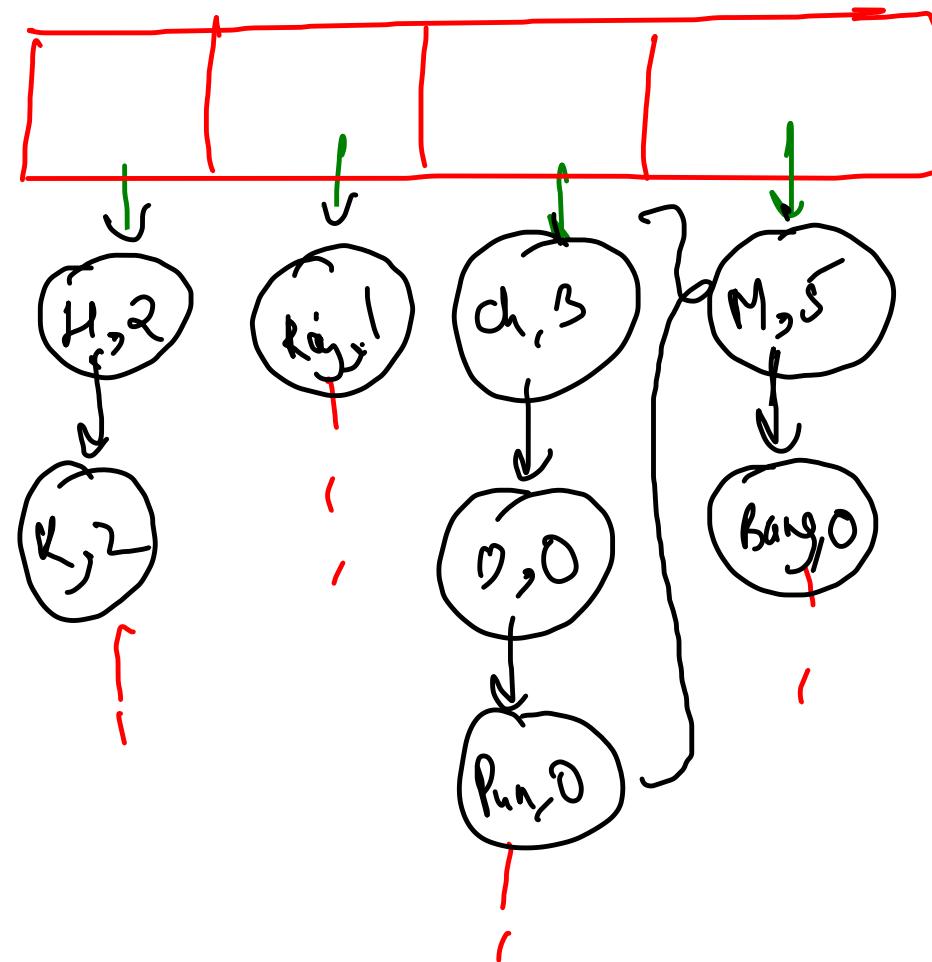
$O(N)$ Time

HashMap Construction

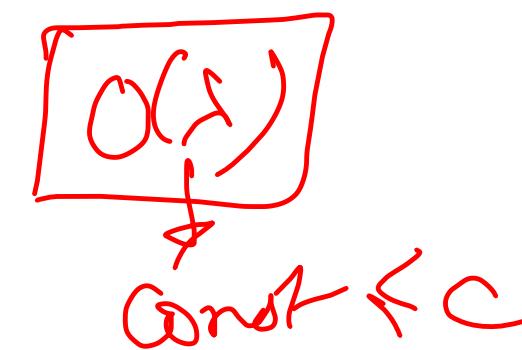


→ Two keys can have same hashfn

→ A given key will never have two hashfn values.



$\text{cap} = 4$



loading factor $\frac{1}{4}$

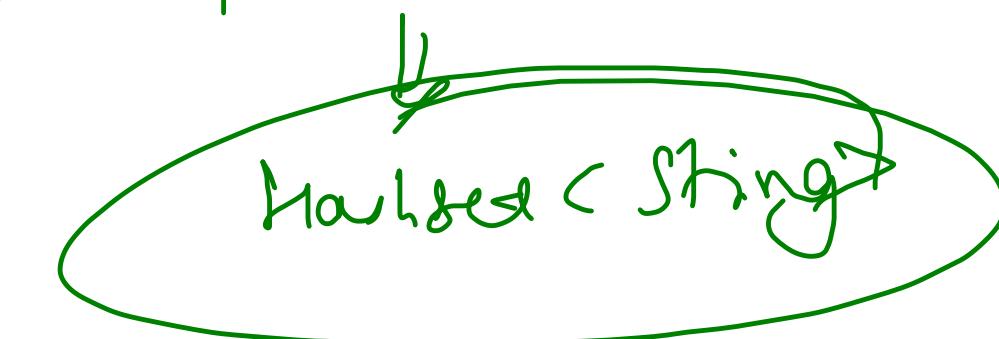
= $\frac{\text{no of nodes}}{\text{in one bucket}}$

= $\frac{\text{total nodes}}{\text{no of buckets}}$

$$= \frac{8}{4}$$

$$= 2$$

HashMap < String, Boolean >



```

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

public HashMap(){
    noOfBuckets = 4;
    noOfNodes = 0;
    buckets = new LinkedList[noOfBuckets];

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

```

```

public void put(K key, V value) throws Exception {
    // O(1)
    int bucketId = getBucketId(key);
    HMNode data = getData(bucketId, key);

    if(data == null){
        // Insertion
        HMNode node = new HMNode(key, value);
        buckets[bucketId].addLast(node);
        noOfNodes++;
    } else {
        // Updation
        data.value = value;
    }
}

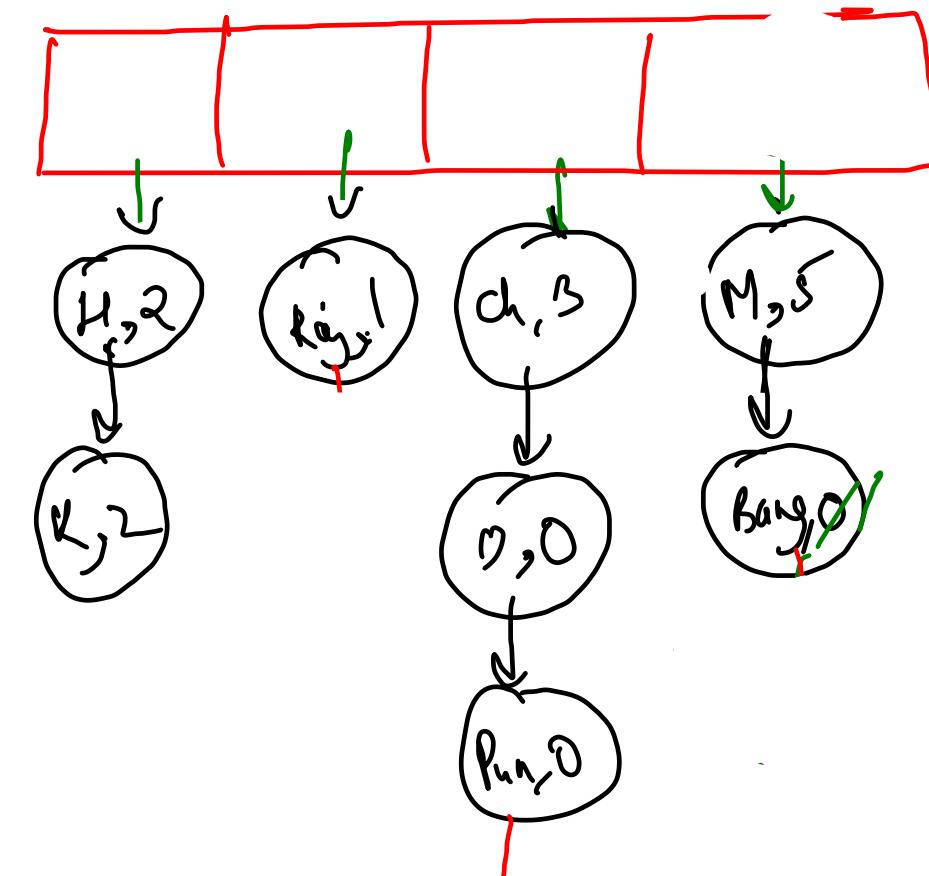
```

```

public int getBucketId(K key) throws Exception{
    // O(1)
    int hashCode = key.hashCode();
    int bucketId = (Math.abs(hashCode)) % noOfBuckets;
    return bucketId;
}

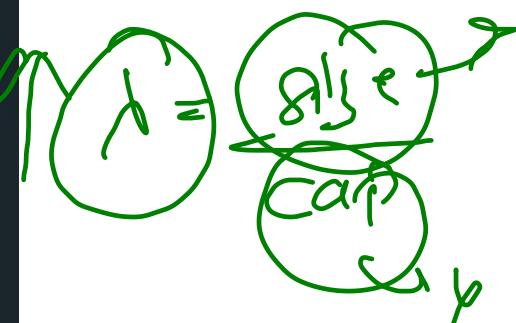
private HMNode getData(int bucketId, K key) throws Exception{
    for(HMNode node: buckets[bucketId]){
        if(node.key.equals(key) == true){
            return node; // data already exist
        }
    }
    return null; // data not exist
}

```



$\text{Cap} = 9$

$\text{Size} = 18$

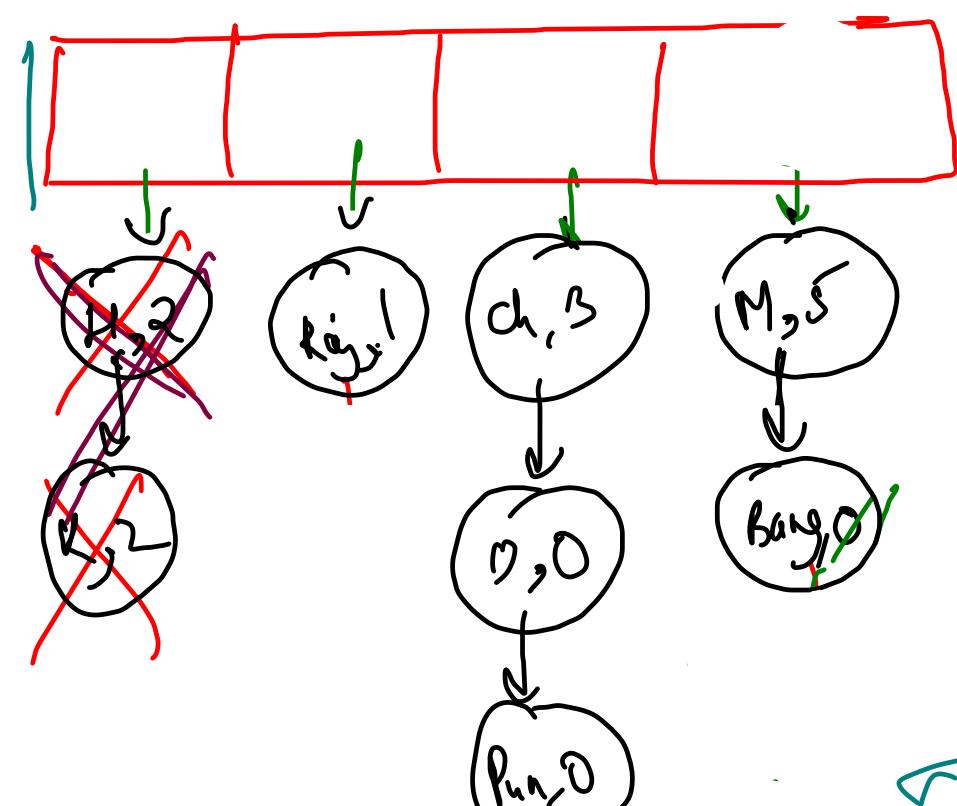


$O(2)$

$O(C)$

Ahm

$1/4 = O(1,2,3)$

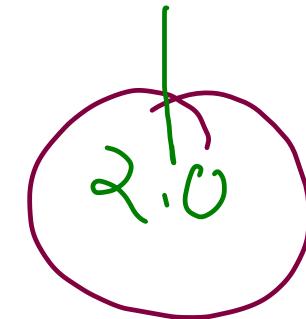


$\text{size} = 8$
 $\text{cap} = 9$

$O(N)$

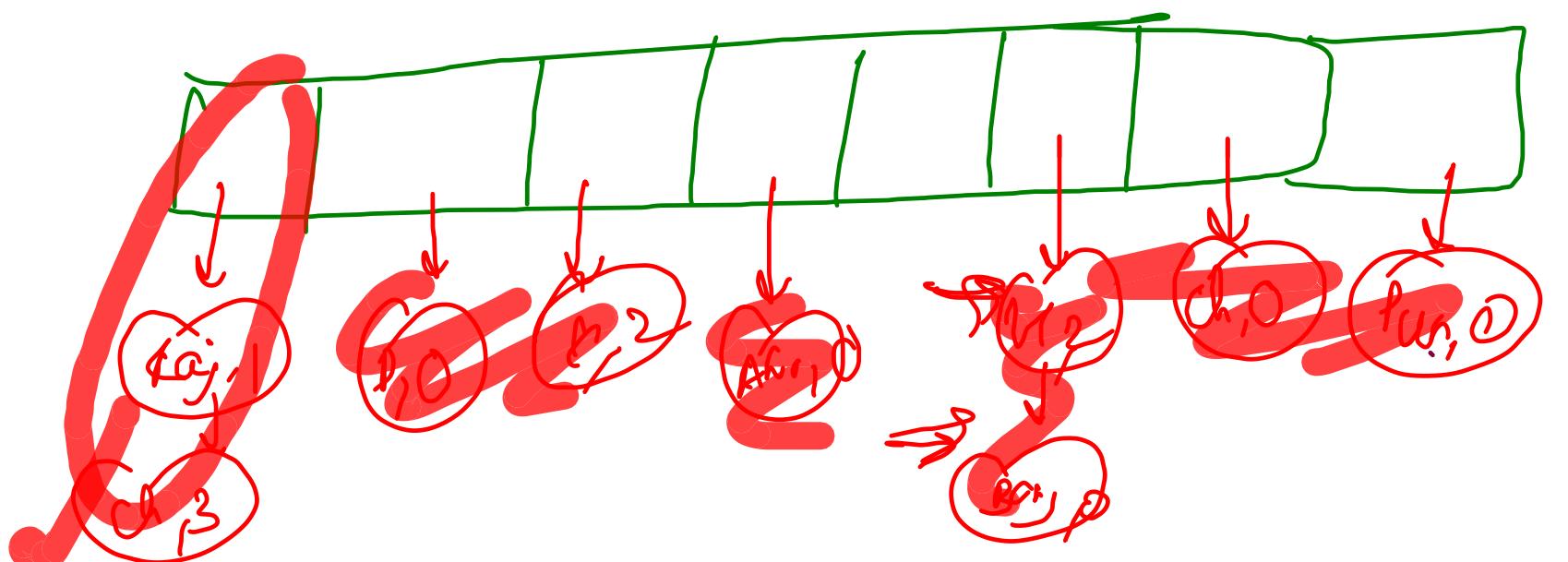
Rehashing

$$\lambda = \frac{8}{9} = 0.888\ldots$$



~~$O(c)$~~

hashcode = Hyp
 $(\text{hi}) \mod 4$



$$\lambda = \frac{9}{8} = 1.125\ldots$$

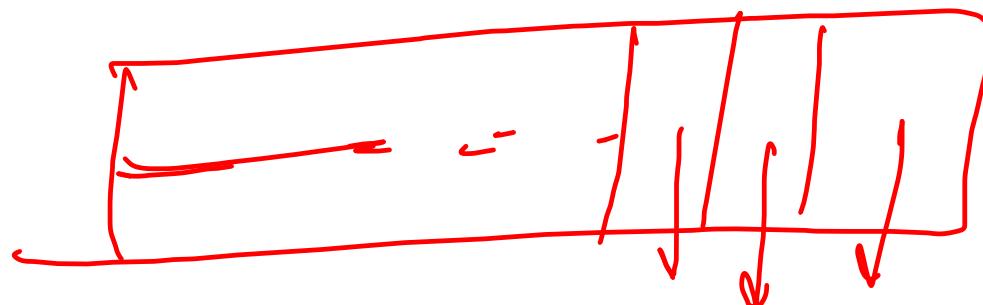
Time Comp

→ Normal
→ Rehashing

→ worst case

Leading factors

Collision $O(w)$



9^{..};
931.
92/1

C - 99°

Collisions

1

193

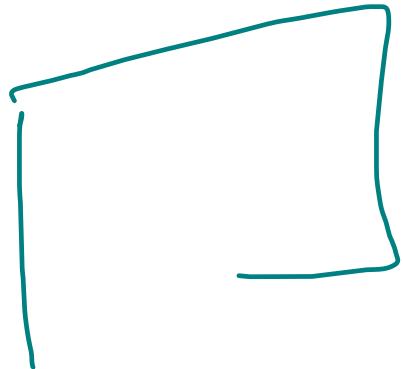
911 (8889)

Figure 10: The effect of the parameter α on the solution curves.

→ 9219

A red ink drawing on a white background. The main subject is a large, roughly rectangular shape with a thick red border. Inside this shape, there are several smaller, irregularly shaped figures. One figure at the top contains the number '9'. Another figure on the right side contains the number '1'. In the center, there is a figure containing the letter 'e' and the number '2'. At the bottom, there is a figure containing the number '7'. The entire drawing is done in a simple, hand-drawn style with red ink.

← OCI



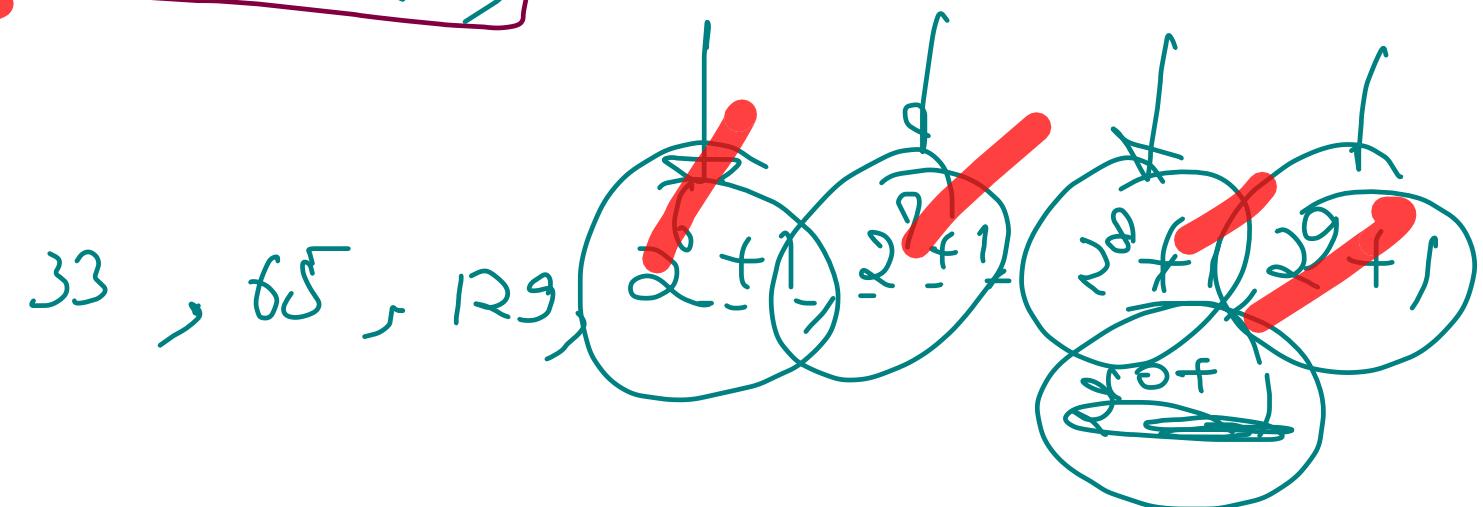
~~1st $\rightarrow O(1)$~~
~~2nd $\rightarrow O(1)$~~
~~3rd $\rightarrow O(1)$~~
~~4th $\rightarrow O(1)$~~
~~5th $\rightarrow O(8)$~~
~~6th $\rightarrow O(1)$~~
~~7th $\rightarrow O(1)$~~

~~8 $\rightarrow O(1)$~~
~~9 $\rightarrow O(16)$~~
~~10 $\rightarrow O(1)$~~
~~11 $\rightarrow O(32)$~~

Amortized

~~1 + 1 + 1 + 1 + 8 + 1 + ...~~
~~+ 16 + ...~~
~~+ 32 + ...~~

$$2^{10} + 1 = O(n)$$



✓ String \Rightarrow

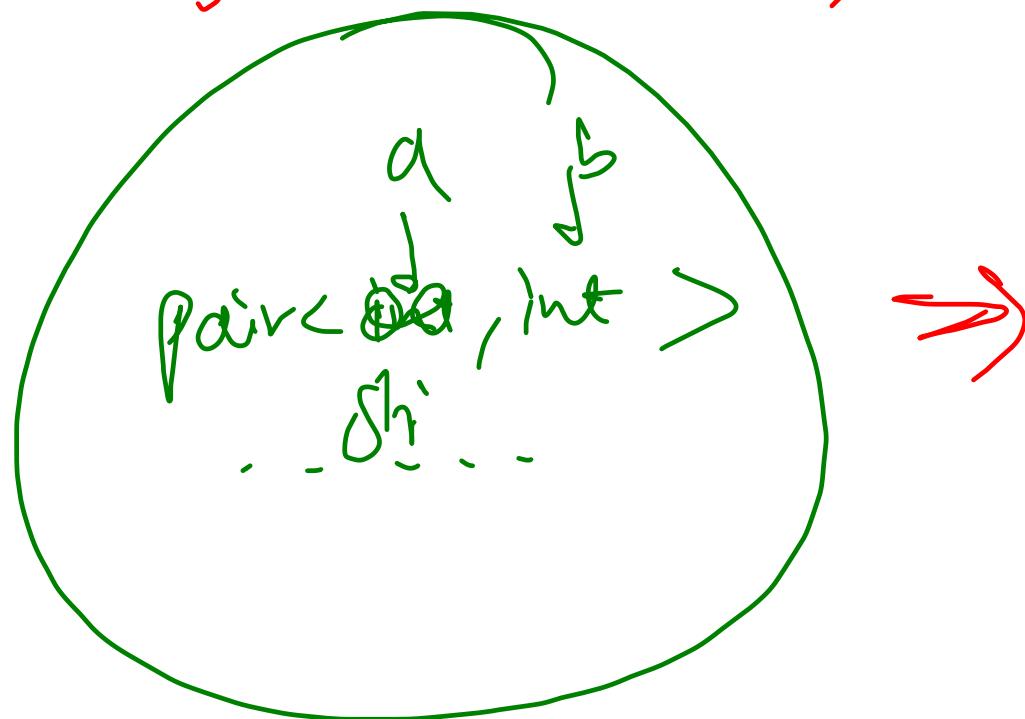
bepcoder
0 1 2 3 4 5 6 7

$O \times P$
 $+ 1 \times e$

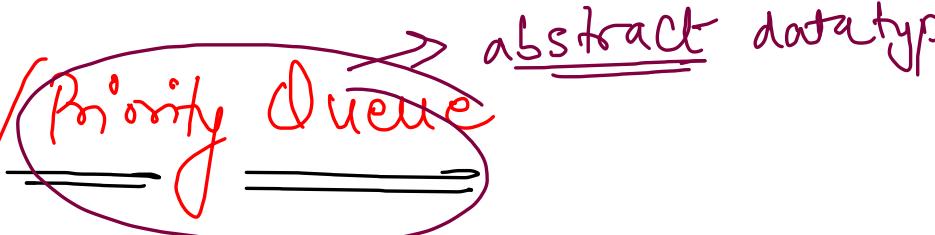
$+ 2 \times P + 3 \times C$
 $+ \dots) / f$

✓ int $\Rightarrow R_1, R_2$

$\rightarrow (\text{key } \times P_1) \times P_2$



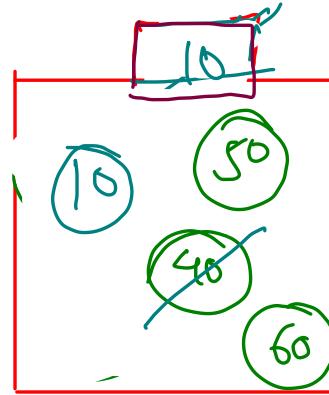
Data Structure



Priority Queue

- add → $O(\log n)$
- removal → $O(\log n)$
- peek → $O(1)$

f+ Min Heap



- pq.remove()
- pq.remove()
- pq.remove()
- pq.remove()

$$n! = n^n$$

Student
→ roll no
→ marks
→ height

- size { $O(1)$ }
- display { $O(n)$ }

{30, 50, 40, 20, 60}

(10)

Inserting n elements

$$\begin{aligned} & \log 1 + \log 2 + \log 3 \\ & + \dots \log n \\ & - \log (1 \times 2 \times 3 \times \dots \times n) \\ & = \log(n!) \\ & = \log(n^n) \end{aligned}$$

for inserting
 n elements
one by one

$n \log n$

Deleting n elements

$$\begin{aligned} & \log n + \log(n-1) + \log(n-2) \\ & + \dots \log 1 \quad \text{for removing } n \text{ elements} \\ & = n \log n \end{aligned}$$

Binary Heap :- Array

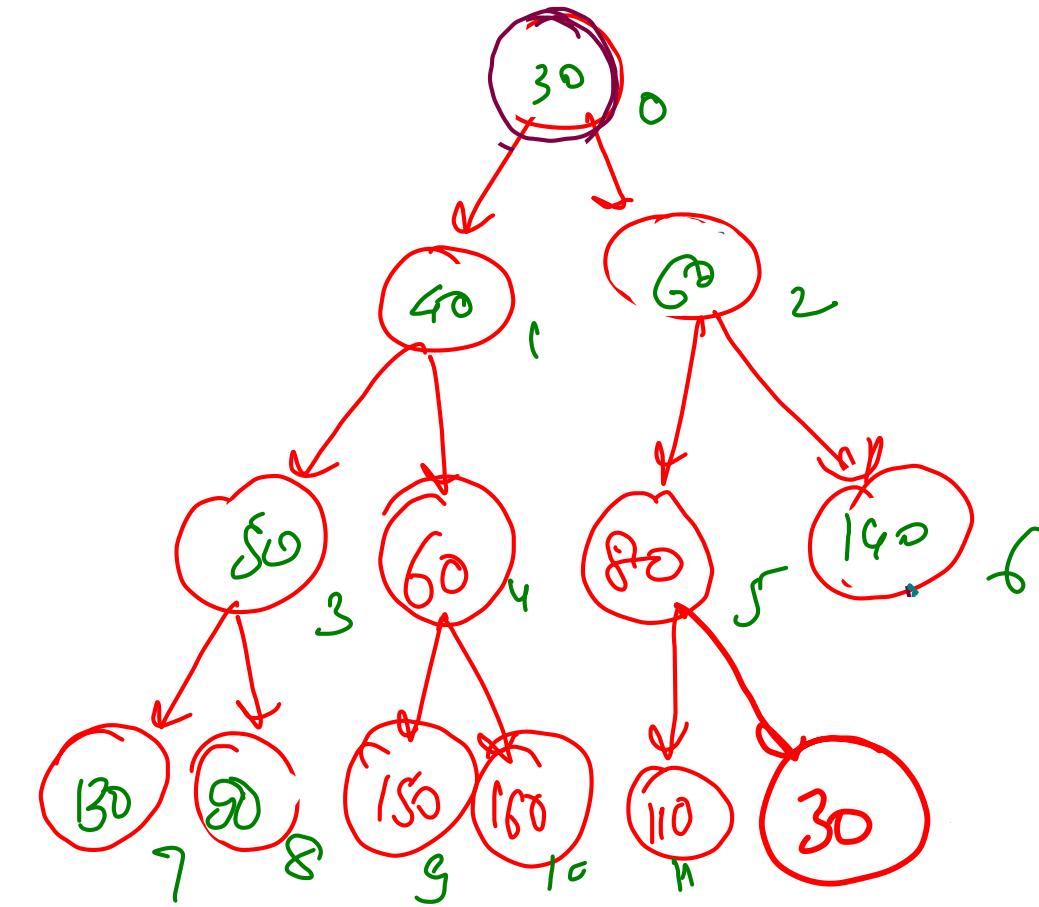
(\Rightarrow complete Binary Tree) \Rightarrow

\checkmark {Heap Order Property}

→ left child :- $2^i + 1$

→ right child :- $2^i + 2$

→ parent :- $(i-1)/2$



$$N \log(N) = \underline{N} \underline{\log N}$$

heapSort

- 5
- 10
- 20
- 30
- 40
- 60

Quick Sort :- ~~Stack~~

Merge Sort :- extra space

Heap Sort :- O(1) extra space

```

public void upheapify(int idx){
    int parIdx = (idx - 1) / 2;

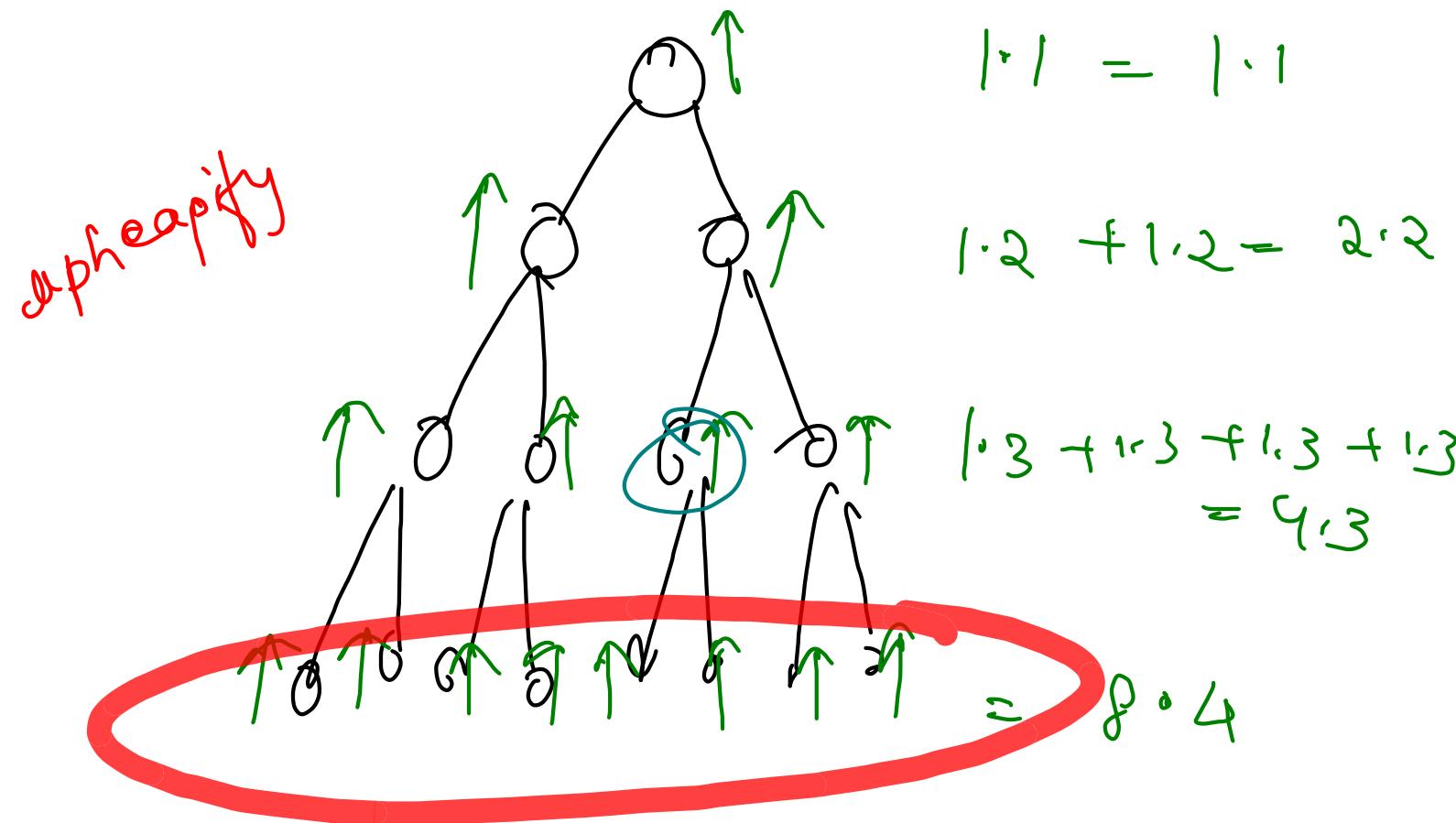
    if(isSmaller(idx, parIdx)){
        swap(idx, parIdx);
        upheapify(parIdx);
    }
}

```

```

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

```



Insetion of
N nodes

$$\left\{ 2^0 \cdot 1 + 2^1 \cdot 2 + 2^2 \cdot 3 + 2^3 \cdot 4 + 2^4 \cdot 5 + 2^5 \cdot 6 \dots \right\} = N \log N$$

3yoda
nodes,
3yada
room
↓
 $O(n \log n)$
3yada
T.C.

```

public void downheapify(int idx){
    int min = idx;
    int left = 2 * idx + 1;
    int right = 2 * idx + 2;

    if(left < size() && isSmaller(left, min)){
        min = left;
    }

    if(right < size() && isSmaller(right, min)){
        min = right;
    }

    if(min != idx){
        swap(idx, min);
        downheapify(min);
    }
}

```

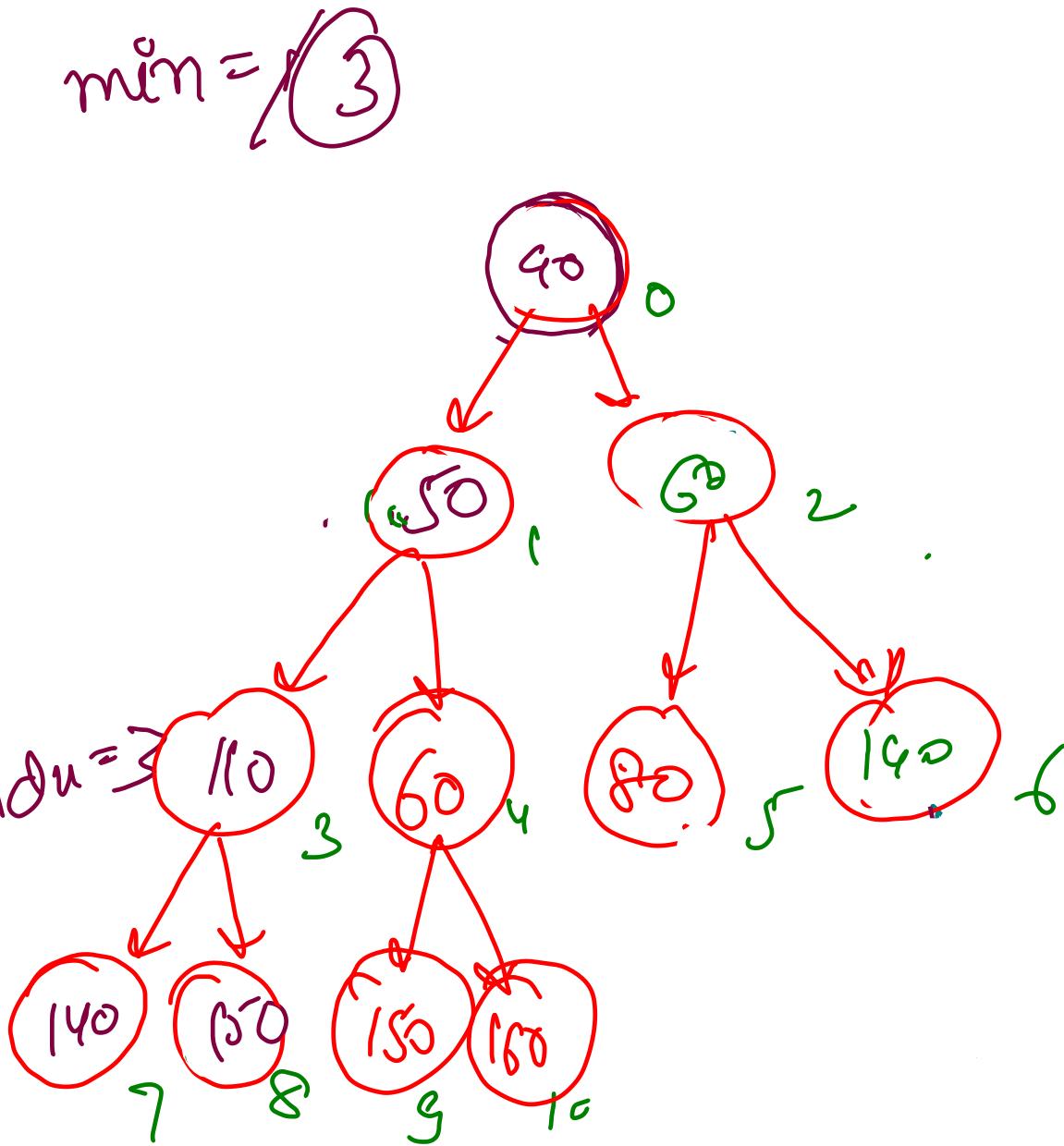
```

public int remove() {
    // O(log n)
    if(size() == 0){
        System.out.println("Underflow");
        return -1;
    }

    int val = peek();
    swap(0, size() - 1);
    data.remove(size() - 1);

    downheapify(0);
    return val;
}

```



opti nodes, zyada kaam

0. 2⁰ + 4

+ 2¹ + 3

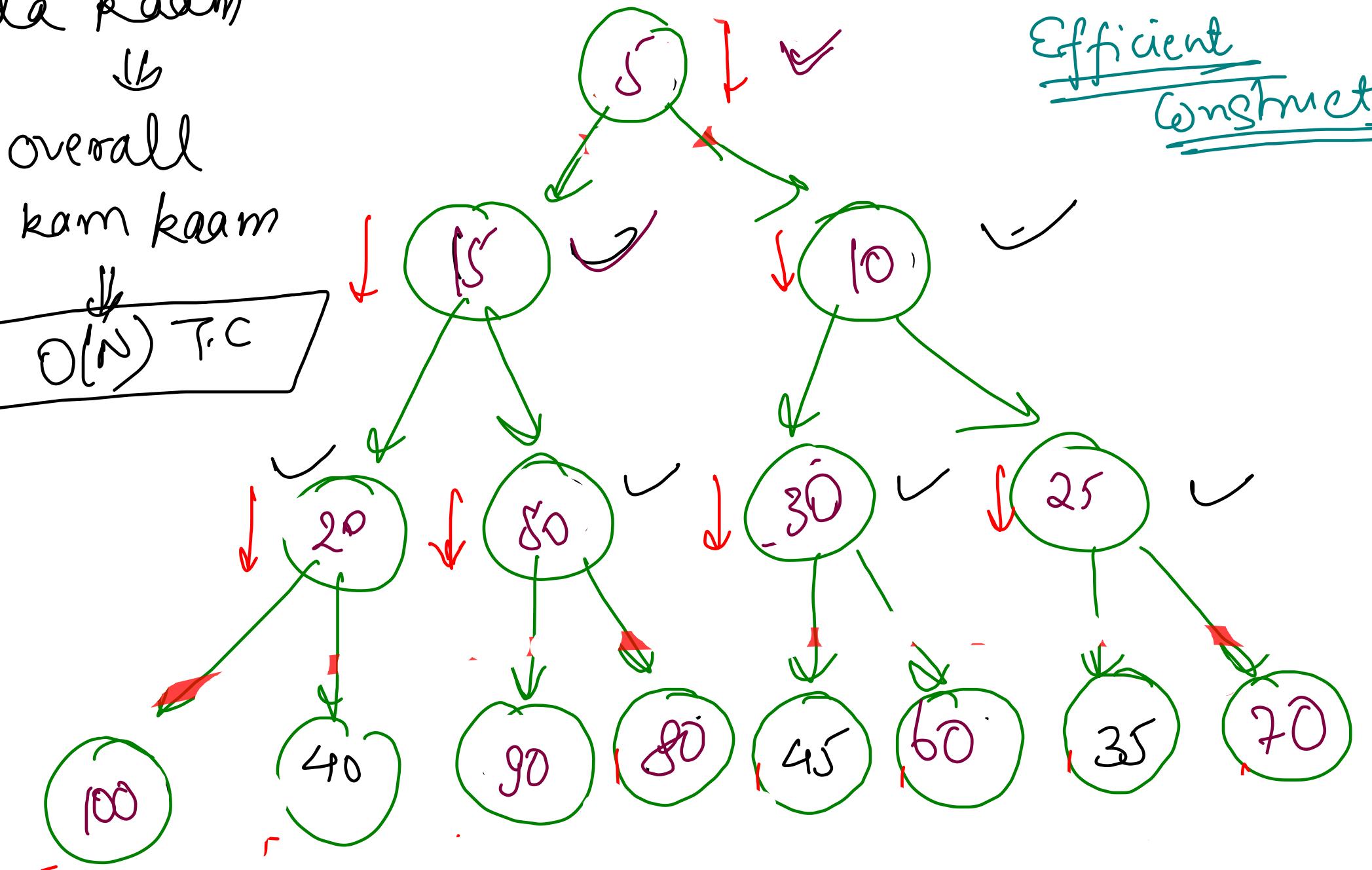
+ 2² + 2

+ 2³ + 1

overall

kam kaam

$O(N)$ T.C



Efficient Constructor

AGP

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

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

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

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

$$= 2^{\log_2 n} - 1 - \log_2 n$$

$$= n - \log n - 1$$

$$\boxed{h = \log_2 n}$$

$$\boxed{O(n)}$$

```
// O(n) for inserting n elements -> per element O(1)
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);
    }
}
```

Interview Prep

10 questions \Rightarrow 3-4 Ques

\Rightarrow 3-4 Ques Algo's

\Rightarrow 1-2 complete

~~360~~
~~660~~

3 monthly

5 classes * 4 weekly

= 20 classes

$20 \times 3 = 60$ classes

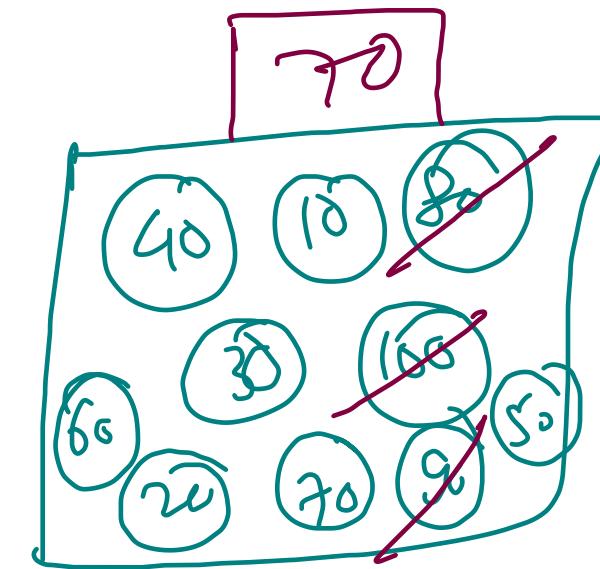
MTW

60 & 6 = 360

$\{ 40, 10, 80, 30, 100, 60, 20, 70, 90, 50 \}$ $k=3$

Approach 1 \rightarrow Max Heap

$\{ 100, 30, 80 \}$

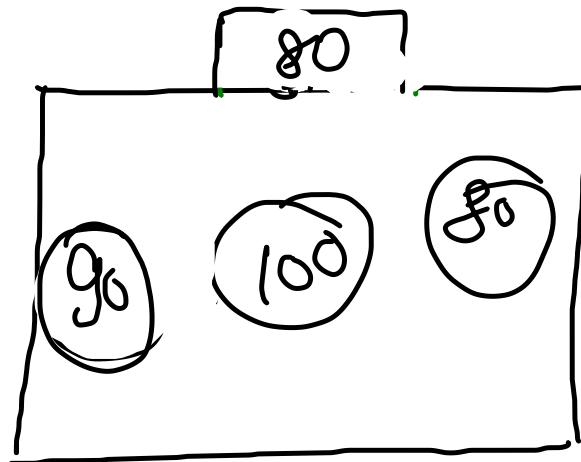


$O(N + K \log N)$

```
public static void approach1(ArrayList<Integer> arr, int k){  
    PriorityQueue<Integer> pq = new PriorityQueue<>(Collections.reverseOrder());  
  
    // O(n)  
    pq.addAll(arr);  
  
    // O(k log N)  
    ArrayList<Integer> res = new ArrayList<>();  
    while(k-- > 0)  
        res.add(pq.remove());  
  
    for(int i=res.size() - 1; i>=0; i--){  
        System.out.println(res.get(i));  
    }  
}
```

$\{ 40, 10, 80, 30, 100, 60, 20, 70, 90, 50 \}$ $\underline{k=3}$

f min heap



$$O(k \log k + (n-k) \log k) = O(n \log k)$$

```
// O(n log k)
public static void approach2(ArrayList<Integer> arr, int k){
    // Min heap
    PriorityQueue<Integer> pq = new PriorityQueue<>();

    // O(k log k)
    for(int i=0; i<k; i++){
        pq.add(arr.get(i));
    }

    // O((n-k) log k)
    for(int i=k; i<arr.size(); i++){

        if(arr.get(i) > pq.peek()){
            pq.remove();
            pq.add(arr.get(i));
        }
    }

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

Today's Evening Questions

- </> Sort K-sorted Array
- ☒ Heap - Comparable V/s Comparator
- </> Merge K Sorted Lists
- </> Median Priority Queue

level 2 & level 3

{ Level up (300 Questions) + Interview Prep (300 Ques) }

• Arrays & Strings { searching, sorting } (greedy, 2 pointers) $\Rightarrow \pm 15$ classes

• Recursion & Backtracking $\Rightarrow \pm 10$ classes

• Stack & Queue $\Rightarrow \pm 5$ classes

• linked list $\Rightarrow \pm 5$ classes

• Trees (Binary Tree, BST) $\Rightarrow \pm 10$ classes

• Hashmap & Heap $\Rightarrow \pm 5$ classes

• Graphs $\Rightarrow \pm 8$ classes

• Dynamic Programming $\Rightarrow \pm 15$ classes

80 classes

↓
5 classes per week

↓
4 months

• Bit Manipulation ② classes

• Tries ② classes

• Advanced Trees ① class

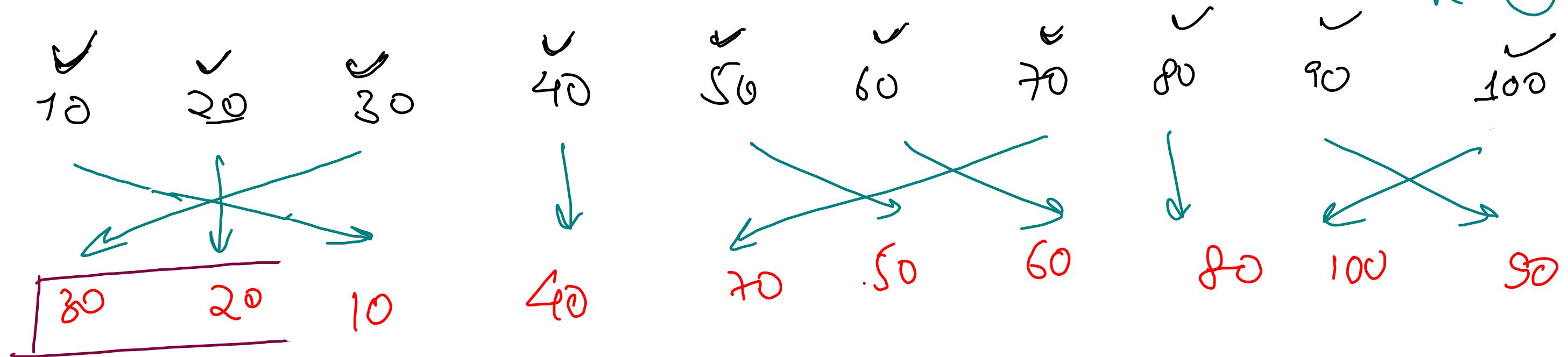
• Mathematics & Number Theory ② classes

• Range Query ② classes

• Geometry & Game Theory ① class

• String pattern matching ① class

Sort k Sorted Array [id_{n-k}, id_{n+k}]



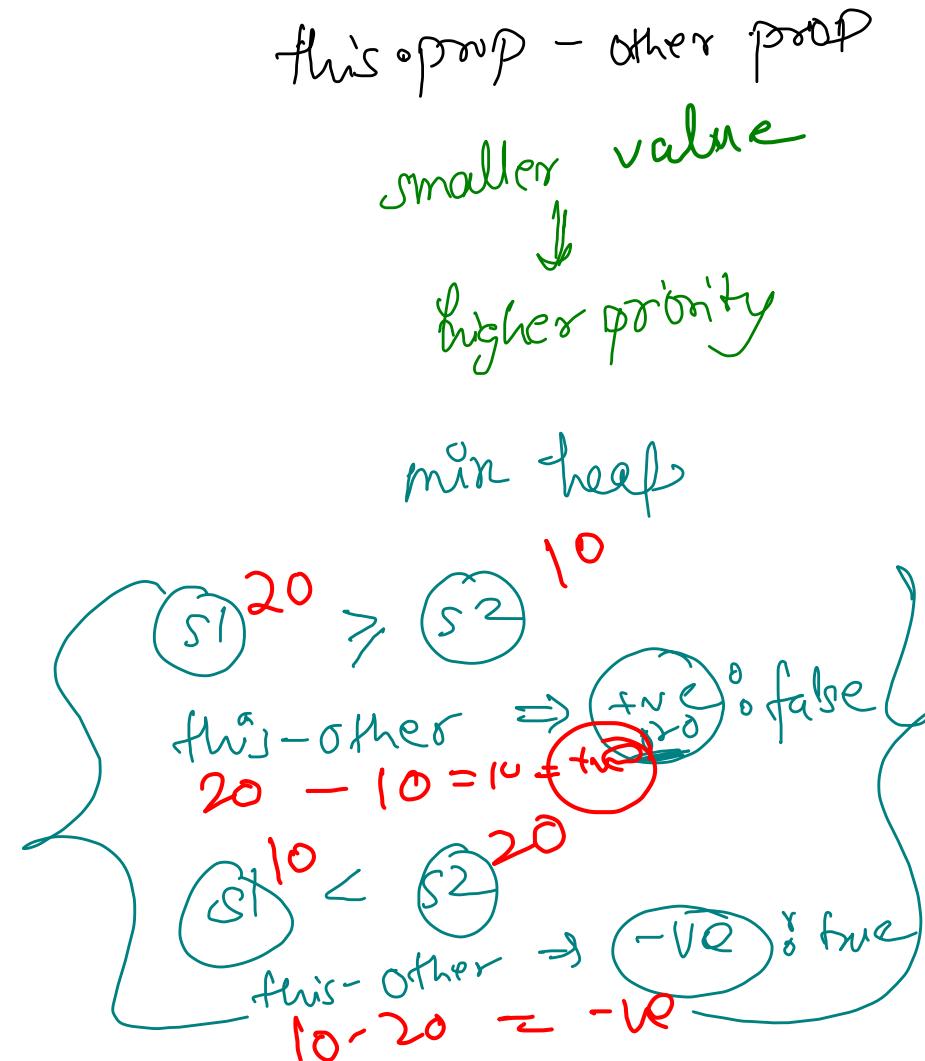
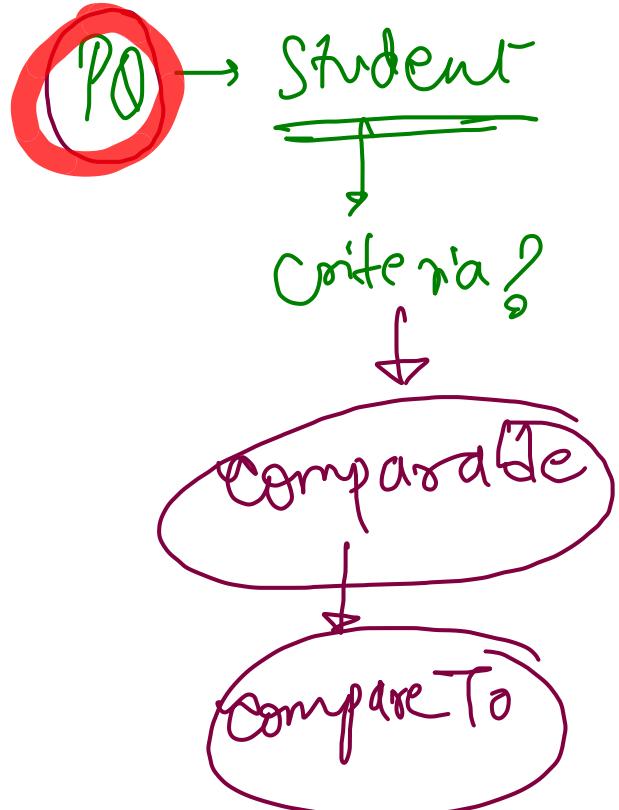
brute force :- Array sort : $N \log N$

optimized code :- $R \log k + (n-k) \log k = (k+n-k) \log k = n \log k$

→ Create Priority Queue of $(k+1)$ elements { sliding window }

{ push 1 ele }
pop 1 ele

Comparable & Comparator



others.prop - this.prop

higher value
↓

higher priority

min heap

$$S1^{10} \leq S2^{20}$$

$$\text{other} - \text{this} = \text{the if false}$$

$$S2^{20} - S1^{10} = 10 \text{ } +ve$$

$$S1^{10} > S2^{20}$$

$$\text{other} - \text{this} = \text{re: the}$$

$$(10 - 20) = -ve$$

Comparable & comparator

```
PriorityQueue<Student> pq = new PriorityQueue<>(new StudentMrksComparator());
```

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

    Student(){}
    Student(int rollNo, int marks, int weight){}
    public String toString(){}

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

public static class StudentWtComparator implements Comparator<Student>{
    public int compare(Student s1, Student s2){
        // Higher Weight - Higher Priority
        return s2.weight - s1.weight;
    }
}

public static class StudentMrksComparator implements Comparator<Student>{
    public int compare(Student s1, Student s2){
        // Higher Marks - Higher Priority
        return s2.marks - s1.marks;
    }
}
```

- ① Generic
 - ② Comparable
 - ③ Obj. compare(s1, s2)
- is smaller

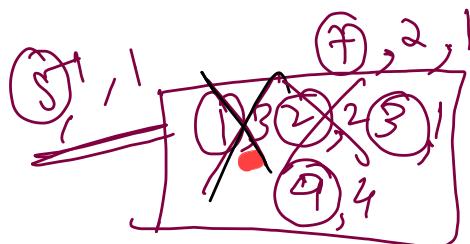
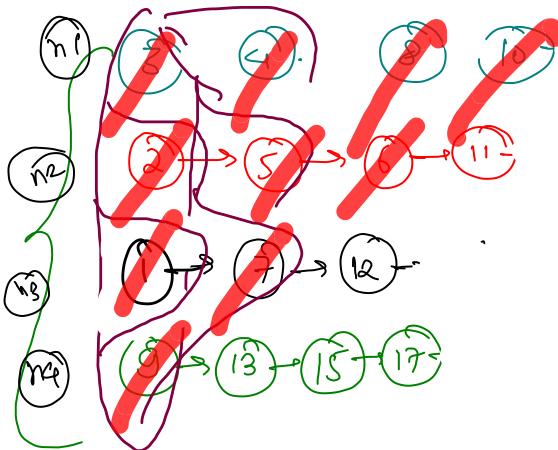
isSmaller

```
public boolean isSmaller(int i1, int i2){  
    K s1 = data.get(i1);  
    K s2 = data.get(i2);  
  
    if(comp == null){  
        // Comparable -> Compare To  
        Comparable s1c = (Comparable)s1;  
        Comparable s2c = (Comparable)s2;  
  
        // s1.compareTo(s2) will not work  
        if(s1c.compareTo(s2c) < 0)  
            return true;  
        return false;  
    } else {  
        // Comparator  
        if(comp.compare(s1, s2) < 0)  
            return true;  
        return false;  
    }  
}
```

Constructor

```
public ArrayList<K> data;  
private int size;  
private Comparator<K> comp;  
  
public Priorityqueue(){  
    data = new ArrayList<>();  
    size = 0;  
    this.comp = null;  
}  
  
public Priorityqueue(Comparator<K> comp){  
    data = new ArrayList<>();  
    size = 0;  
    this.comp = comp;  
}
```

Merge k sorted array



log k

① 2, ③, 4, 7, 8, 9, 10

\Rightarrow push k element : \rightarrow j^{th} element of each array(.)²

while (q.size()

Finite force

Brute-force
merge 2 at a time

$$O(n_1) + O(n_1 + n_2)$$

$$O(n_1 + n_2 + n^3)$$

+

$$O(n_1 + n_2 + n_3 + n_4)$$

1

$$O(n_1 + n_2 + \dots + n_C)$$

d → } data
 } listIdn
 } dataIdn

~~30-40 min~~

5-10 min
15 min

5 min

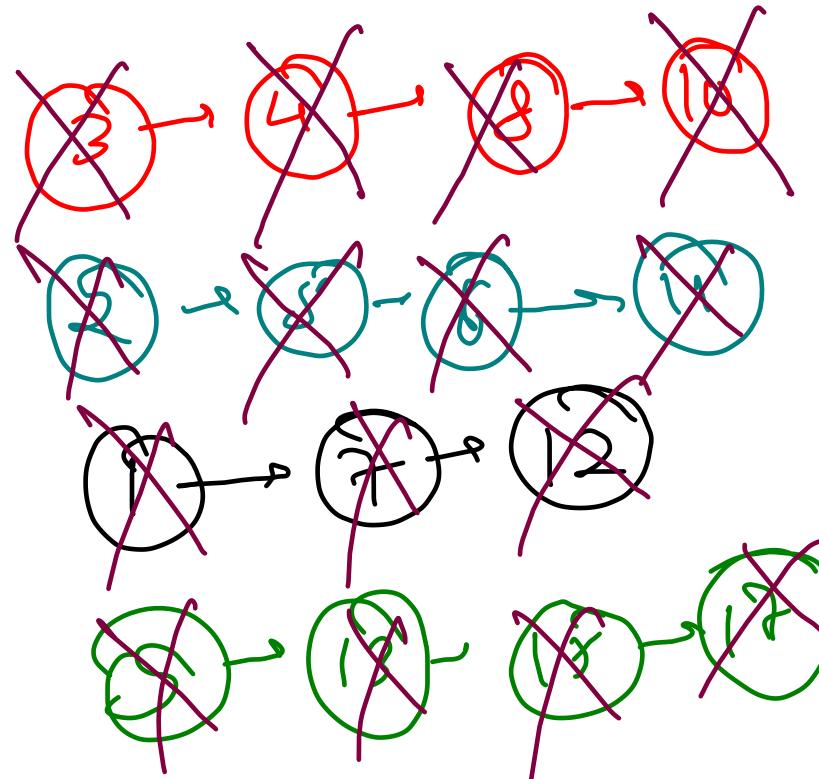
26

grob

$$O(n_1 * (\star) + n_2 (\star - 1) + n_3 * (\star - 2) + n_4 * (\star - 3) + \dots + n_k * 1)$$

$$N^*k + N^*(k-1) + \dots + N^* = N^*k$$

$$= N \neq \begin{cases} 1+2+\dots+n \\ 0(N^{\frac{n}{2}}) \end{cases}$$



```

public static ArrayList<Integer> mergeKSortedLists(ArrayList<ArrayList<Integer>> lists){
    PriorityQueue<Pair> q = new PriorityQueue<>();
    ArrayList<Integer> res = new ArrayList<>();

    // Insert 0th index element of each arraylist
    // O(k log k)
    for(int i=0; i<lists.size(); i++){
        q.add(new Pair(lists.get(i).get(0), i, 0));
    }

    // O(Nt * logk)
    while(q.size() > 0){
        // Pop one pair
        Pair top = q.remove();
        res.add(top.data);

        if(top.dataIdx + 1 < lists.get(top.listIdx).size()){
            // insert next pair
            q.add(new Pair(lists.get(top.listIdx).get(top.dataIdx + 1), top.listIdx, top.dataIdx + 1));
        }
    }

    // O(Nt * logk + k * logk) where Nt = n1 + n2 + n3 + ... nk
    return res;
}

```