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
Update → put ⇒ O(1)

Delete → remove ⇒ O(1)

Read → get ⇒ O(1)
↳ If key exist then return value else return null;

Display

Size ⇒ O(1)

ContainsKey → find ⇒ O(1)

keySet → keys

Key → value

eg IPL teams → trophies count
String → Int

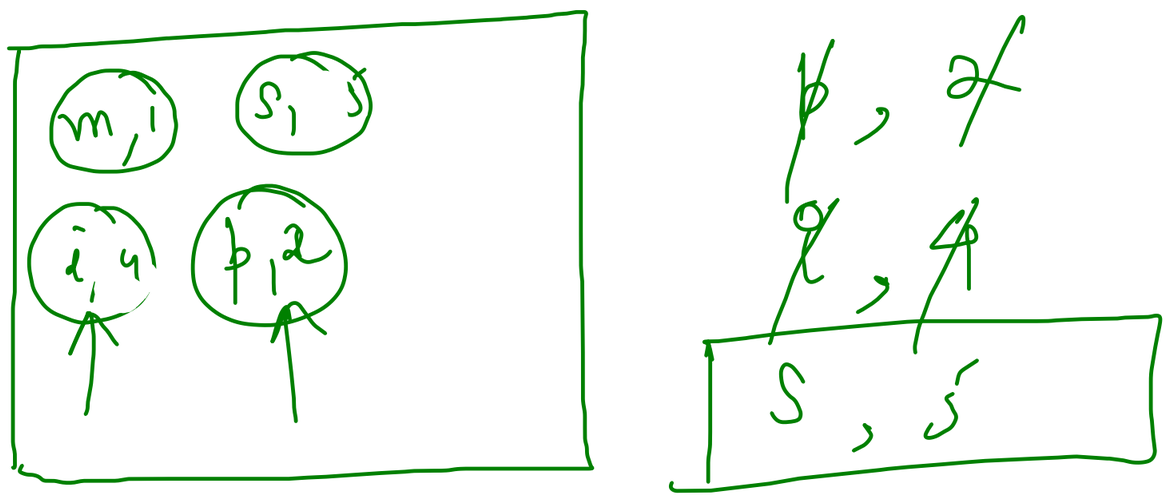
{ CSK → 4
MI → 5
SRH → 2
DC → 0

eg countries → population
String → Integer

{ India → 130
USA → 40
China → 200

Highest Frequency Character

m i s s i s s i p p i

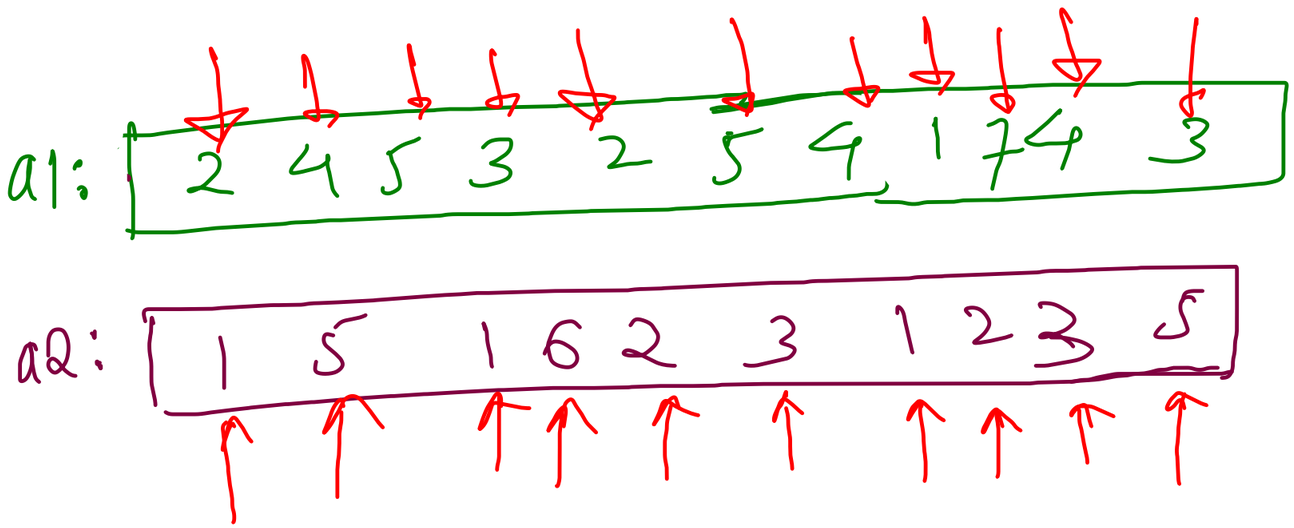


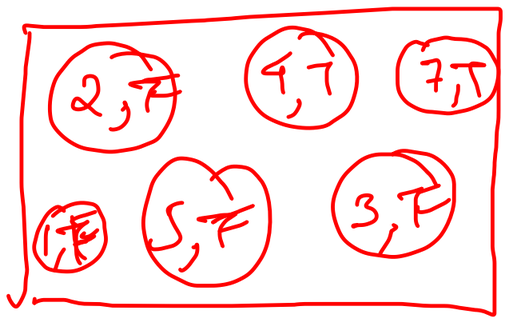

```java
HashMap<Character, Integer> freq = new HashMap<>();

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);
```

O(N)

O(keys) = O(256)

Get Common Elements - 1





<Integer, Boolean>

1, 5, 2, 3

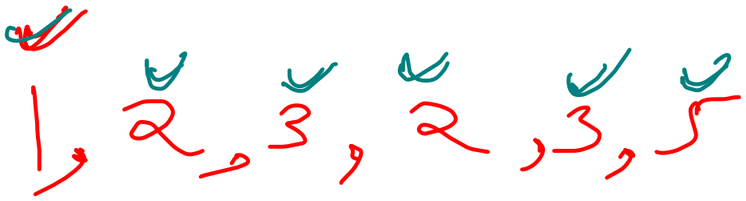
```java
HashMap<Integer, Boolean> hm = new HashMap<>();
for(int i=0; i<n1; i++)
    hm.put(arr1[i], true);

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);
    }

}
```

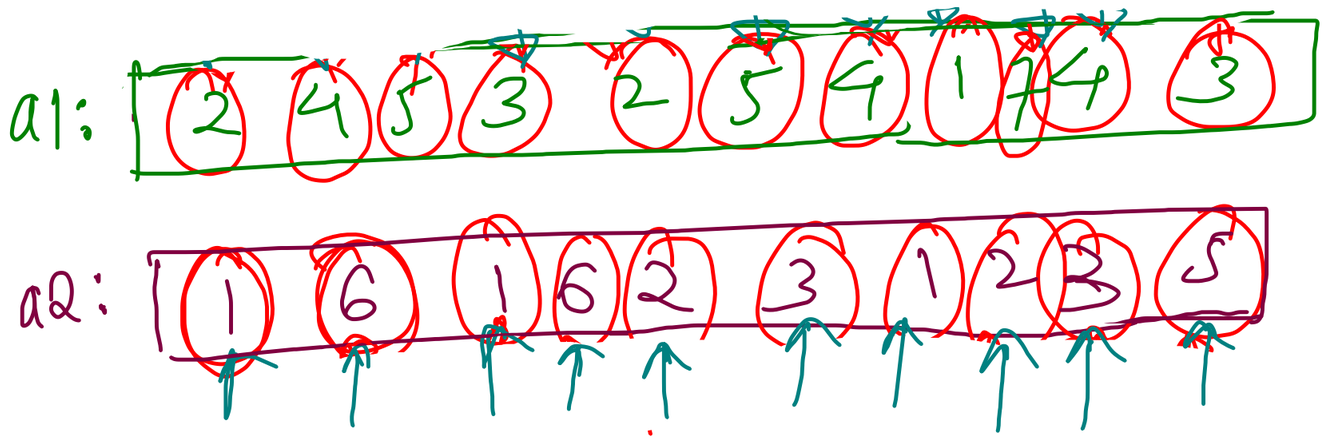
O(n1)

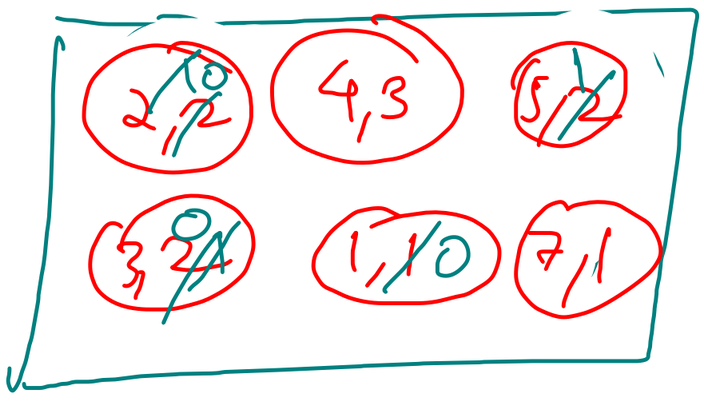
O(n2)

O(n1+n2)

Get Common Elements - 2

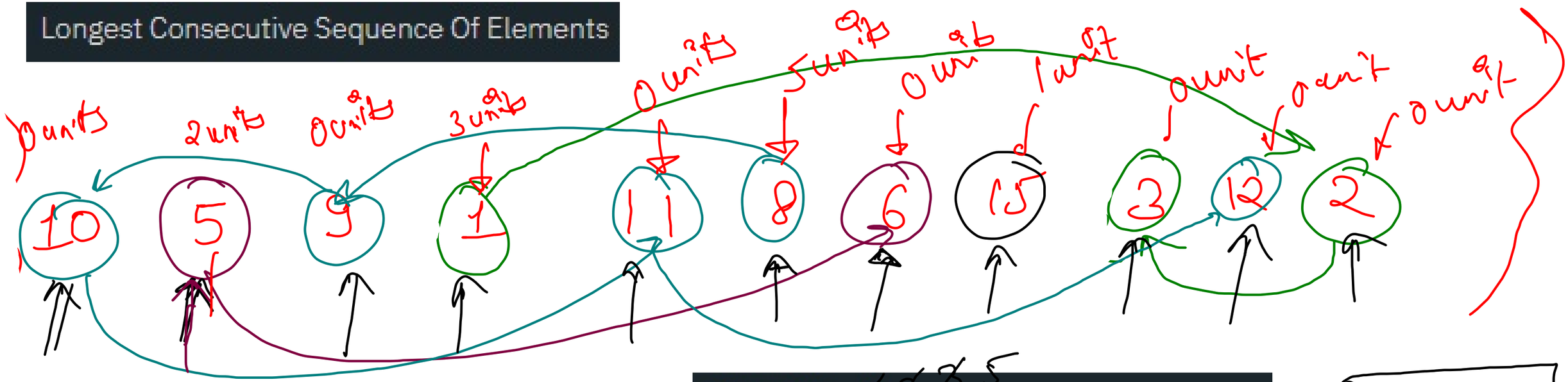


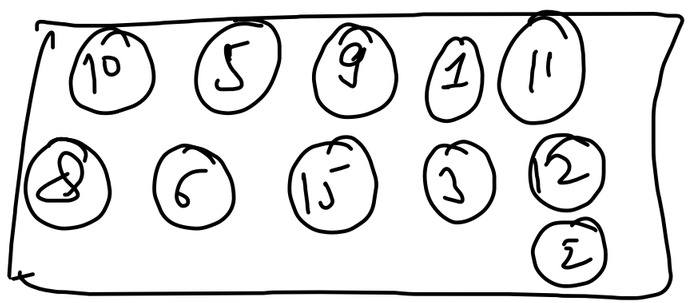





```java
HashMap<Integer, Integer> hm = new HashMap<>();
for(int i=0; i<n1; i++){
    if(hm.containsKey(arr1[i])){
        hm.put(arr1[i], hm.get(arr1[i]) + 1);
    } else {
        hm.put(arr1[i], 1);
    }
}

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






```java
int maxChain = 0;
int startingPt = 0;

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 → 2 → 3

8 → 9 → 10 → 11 → 12

5 → 6

15

O(N) Time

Hashmap Construction

get → O(1)

put → O(1)

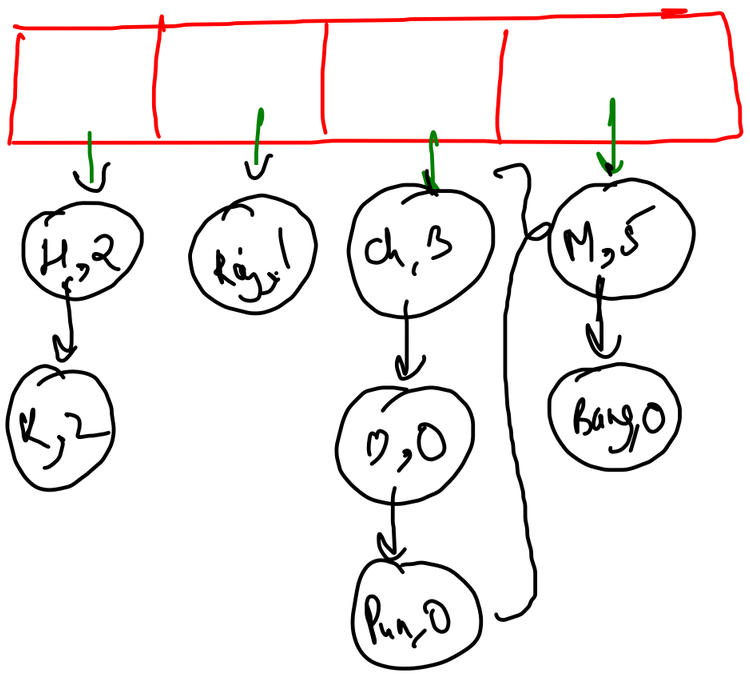
containsKey → O(1)

remove → O(1)

keySet

size

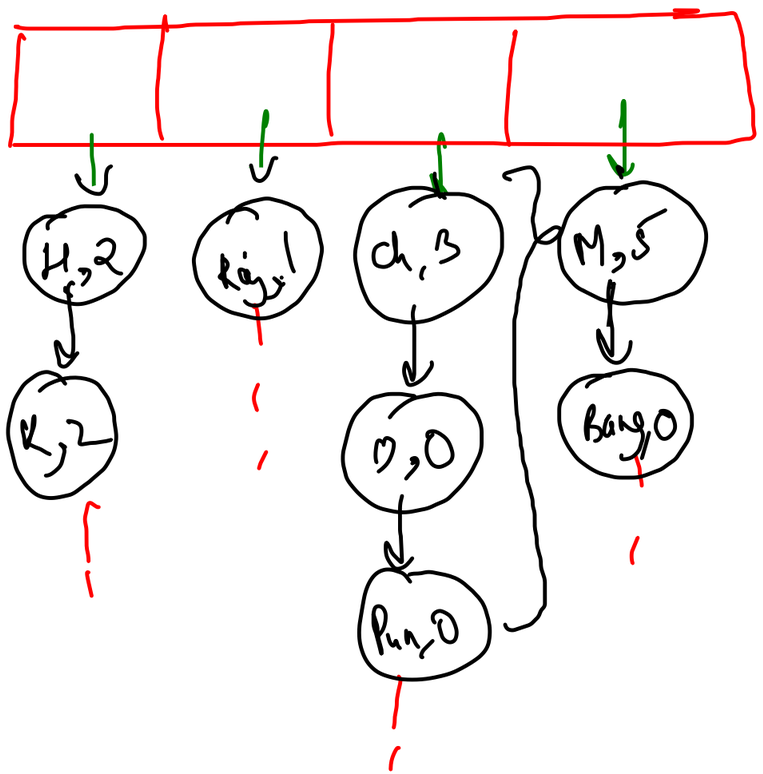
display



hashfunction

→ Two keys can have same hashfn

→ A given key will never have two hashfn values



cap = 4

$O(\lambda)$

Const < C

loading factor (λ)

= avg no of nodes in one bucket

$= \frac{\text{total nodes}}{\text{no of buckets}} = \frac{8}{4}$

$= 2$

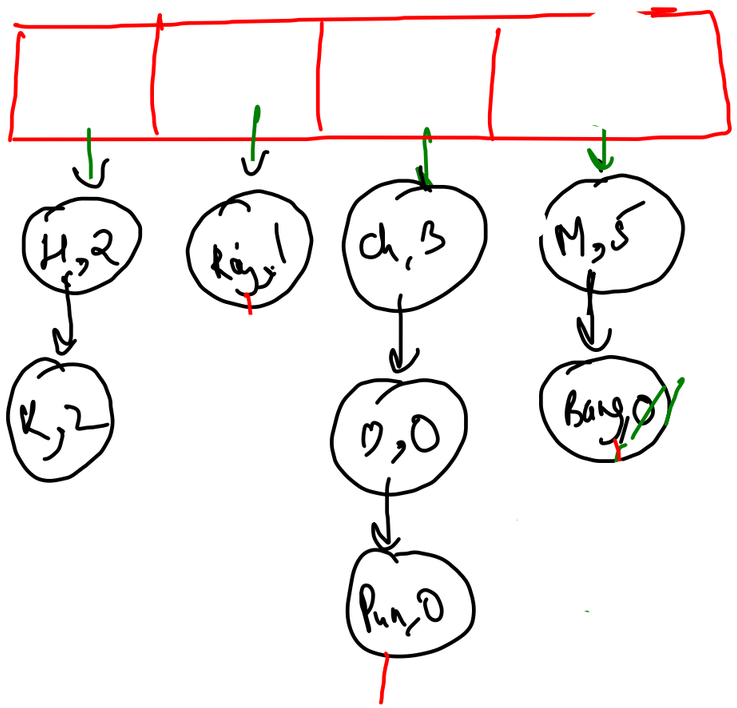
Hashmap < String, Boolean >

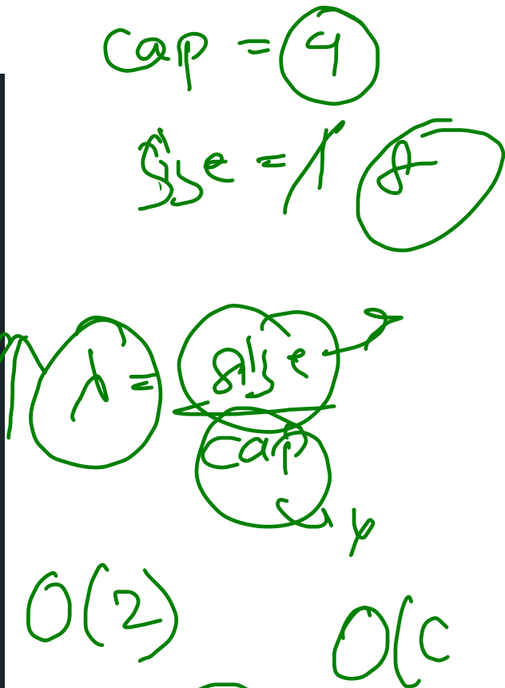
HashSet < String >

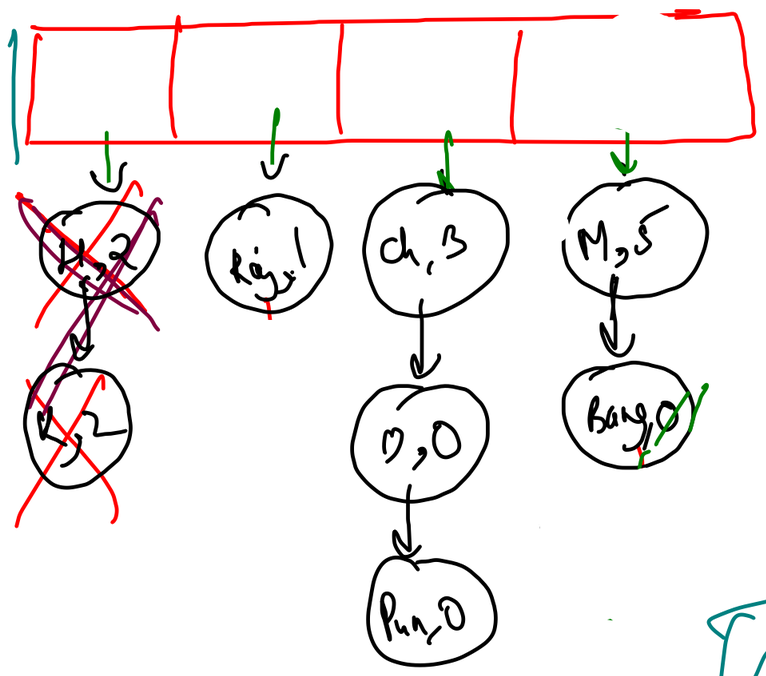
```java
private LinkedList<HMNode>[] buckets;
private int noOfNodes;
private int noOfBuckets;

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

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

```java
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;
    }
}
```

```java
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
}
```







O(N)

size = 8

cap = 4

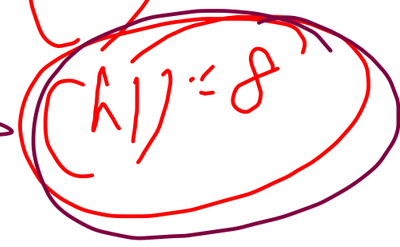
Rehashing

$\lambda = \frac{8}{4} = 2.0$

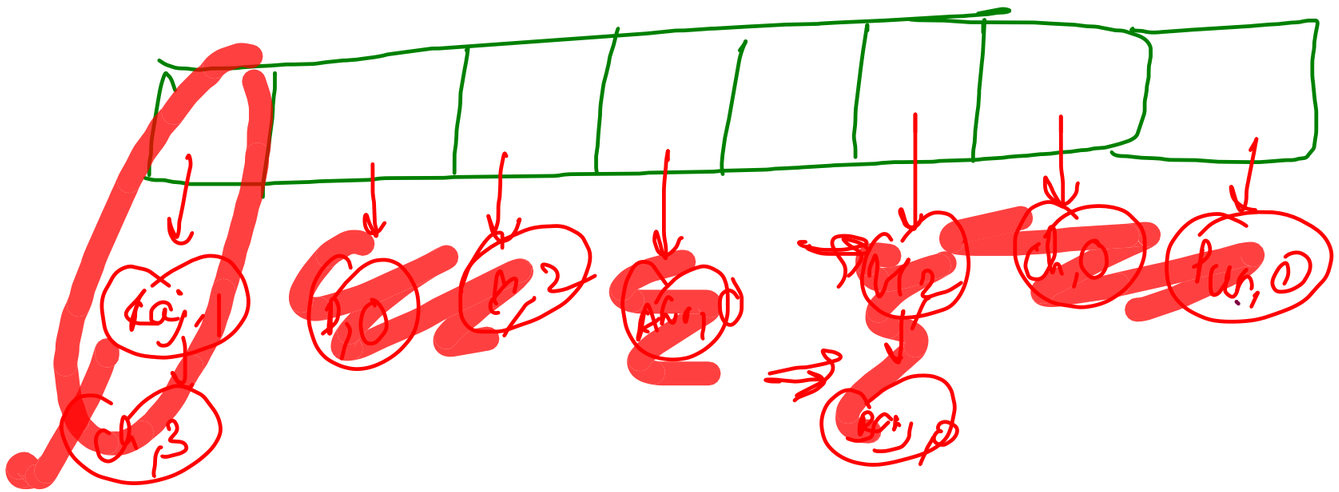
O(C)

hashcode = Hyp

(h1) % 4



O(N)



$\lambda = \frac{9}{8} = 1.\ldots$

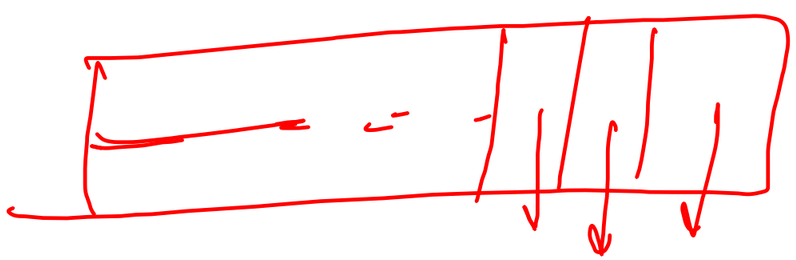
→ Time Comp
- → Normal
- → Rehashing

→ worst case

→ loading factor

0 - 999

collisions O(N)



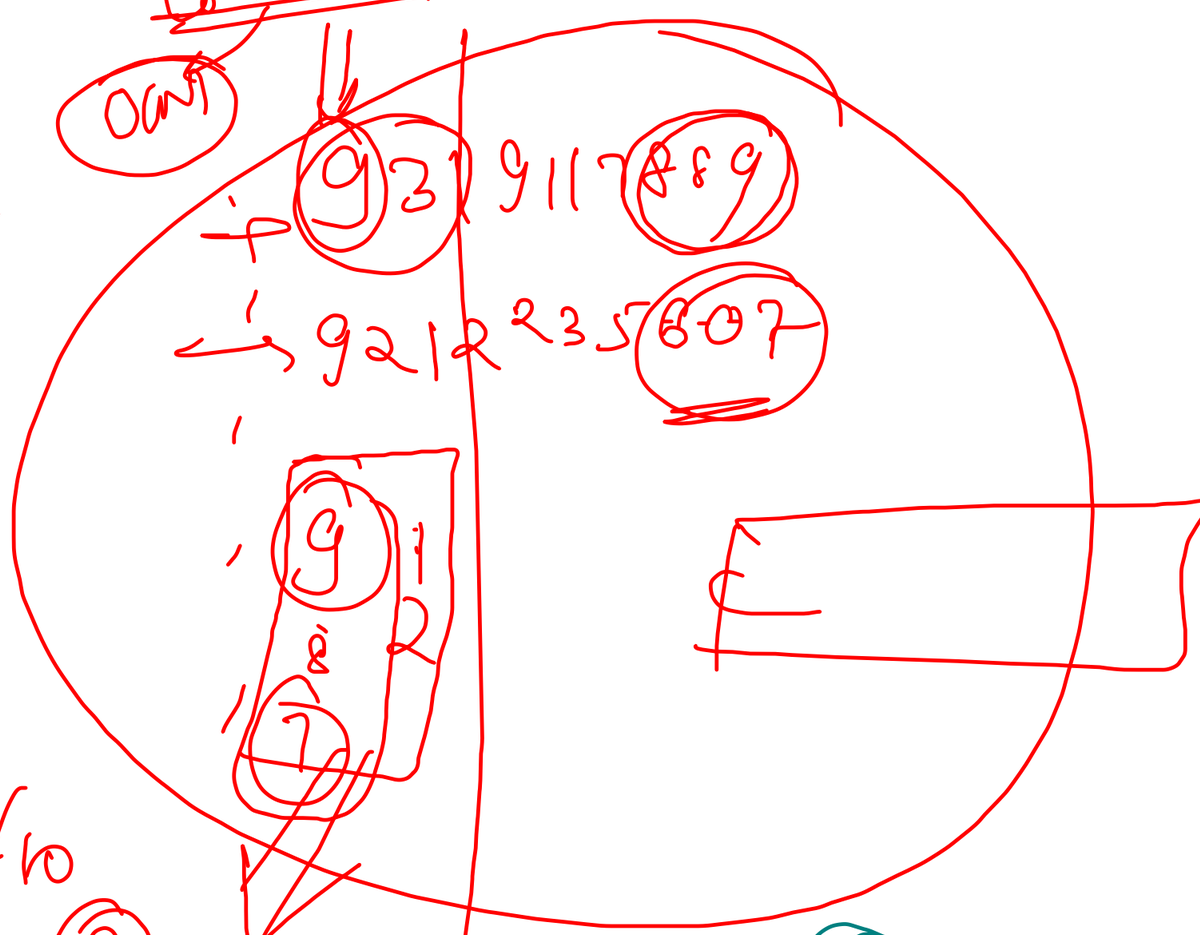
1000

9...

931

921

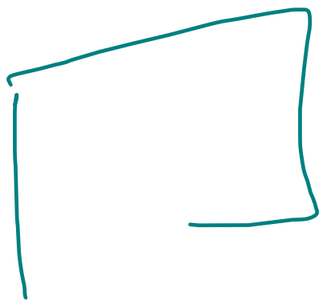
collisions



19%10

1,8,9

O(1)



1st → ~~O(n)~~ O(1)

2nd → ~~O(n)~~ O(1)

3rd → O(1)

4th → O(1)

5th → O(8)

6th → O(1)

7 → O(1)

8 → O(1)

9 → O(16)

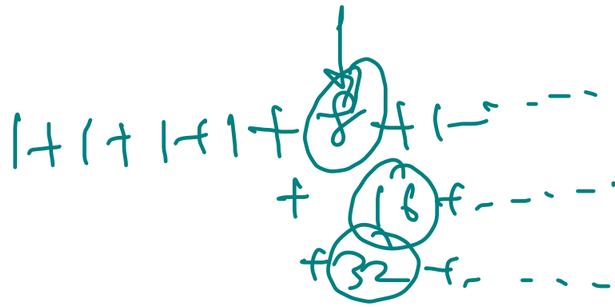
⋮

16 → O(1)

17 → O(32)

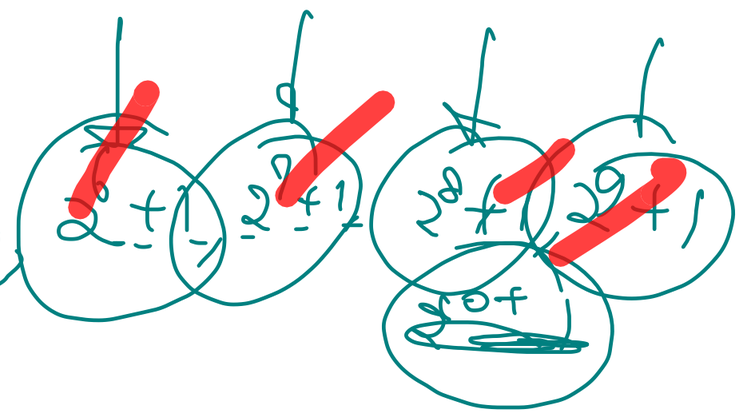
33 , 65 , 129 ,

Amortized



$2^{10}+1$

$= O(1)$

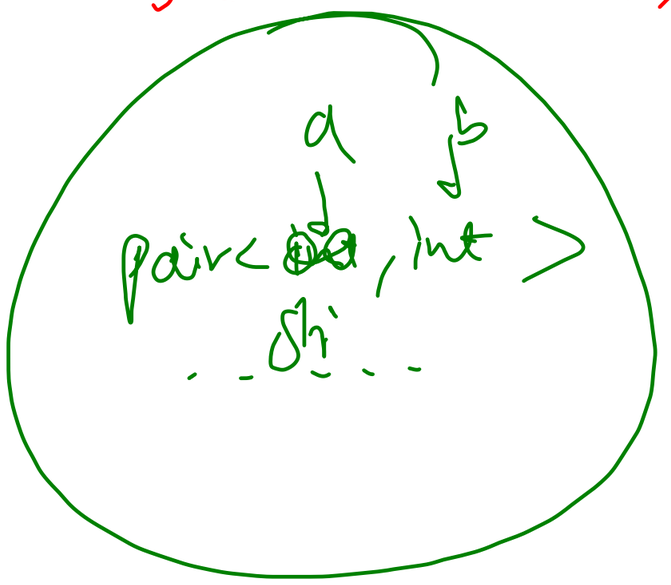


- String ⇒

pepcoder
0 1 2 3 4 5 6 7

= (0 x p + 1 x e + 2 x p + 3 x c + ---) % f

- int ⇒ P1, P2 ⇒ (key x P1) % P2



Data structure

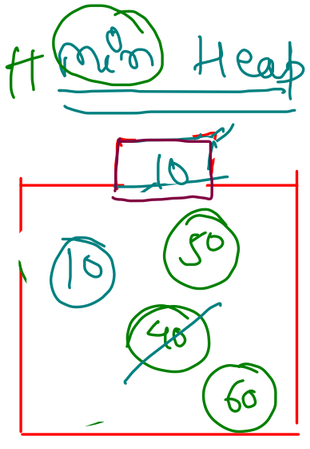
Binary Heap / Priority Queue → abstract datatype

$n! = n^n$

Priority Queue

- add → O(log n)
- removal → O(log n)
- peek → O(1)
- size {O(1)}
- display {O(N)}

Min Heap



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

{30, 50, 40, 20, 60}
10

Student
- roll no
- marks
- height

Inserting n elements

$\log 1 + \log 2 + \log 3 + \cdots \log n$

$= \log(1 \times 2 \times 3 \times \cdots n)$

$= \log(n!)$

$= \log(n^n) = n \log n$

for inserting n elements one by one

Deleting n elements.

$\log n + \log(n-1) + \log(n-2) + \cdots \log 1$ for removing n ele

$= n \log n$

Binary Heap :→ Array

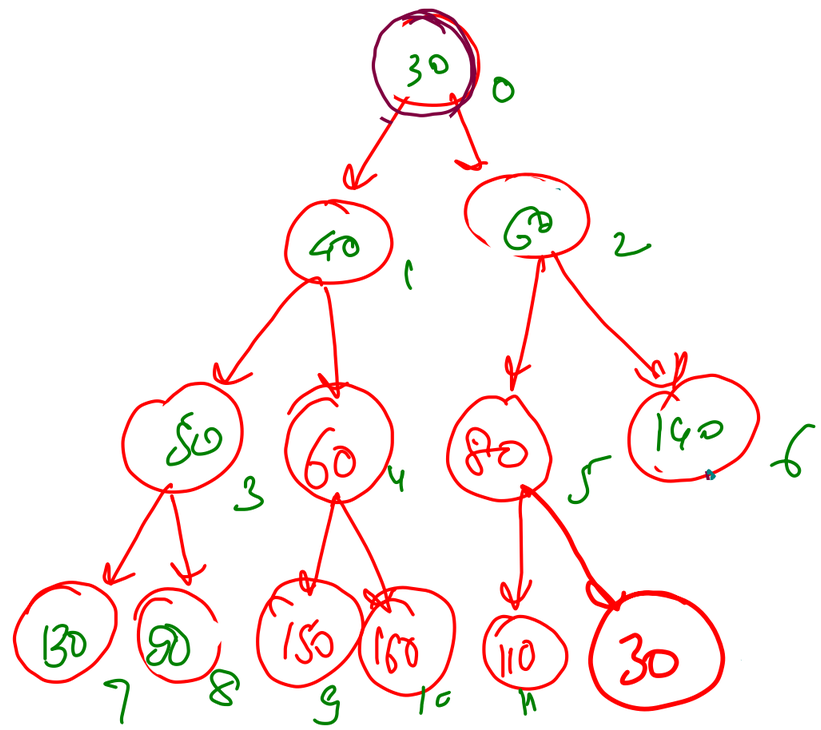
① {Complete Binary Tree} ⇒

② {Heap order Property}

→ left child :- $2*i+1$

→ right child :- $2*i+2$

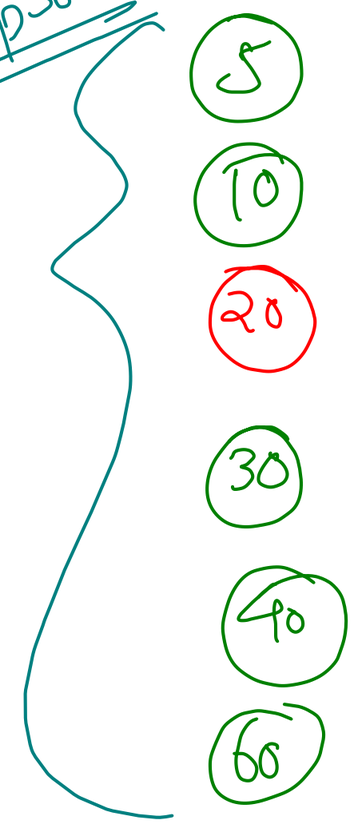
→ parent :- $(i-1)/2$



N*O(logN) = NlogN

↑

HeapSort-



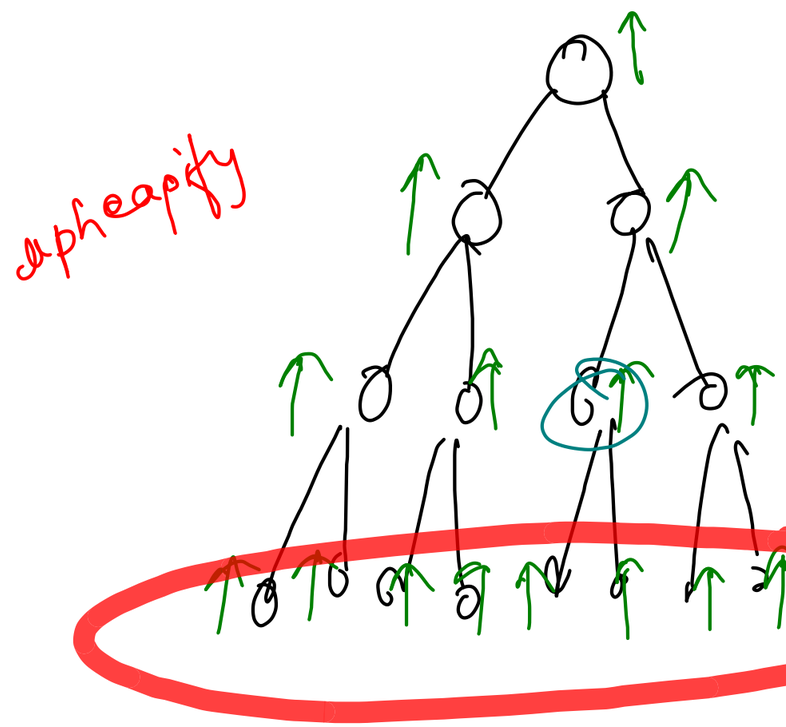
Quicksort :→ ~~Stable~~

MergeSort :- extra space

Heap Sort :- O(1) extra space

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

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

```java
public void add(int val) {
    // O(log n)

    data.add(val);
    upheapify(size() - 1);
}
```



$1 \cdot 1 = 1 \cdot 1$

$1 \cdot 2 + 1 \cdot 2 = 2 \cdot 2$

$1 \cdot 3 + 1 \cdot 3 + 1 \cdot 3 + 1 \cdot 3 = 4 \cdot 3$

$= 8 \cdot 4$

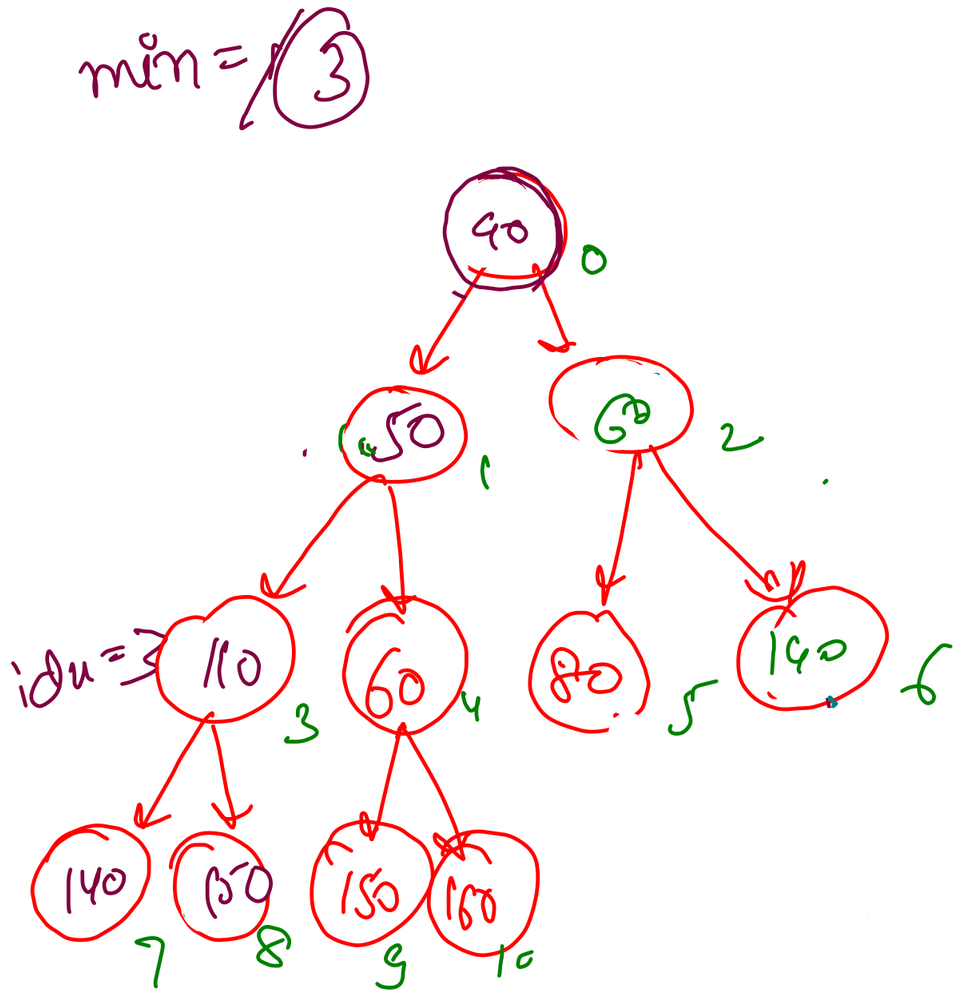
Zyada nodes, zyada kaam ↓ O(nlogN) Zyada T.C.

Insertion of N nodes

$$\{2^0 \cdot 1 + 2^1 \cdot 2 + 2^2 \cdot 3 + 2^3 \cdot 4 + 2^4 \cdot 5 + 2^5 \cdot (\text{+..})\} = N \log N$$

```java
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);
    }
}
```

```java
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;
}
```



~~NN~~ nodes, zyada kaam

$2^0 \cdot 4$

$+$

$2^1 \cdot 3$

$+$

$2^2 \cdot 2$

$+$

$2^3 \cdot 1$

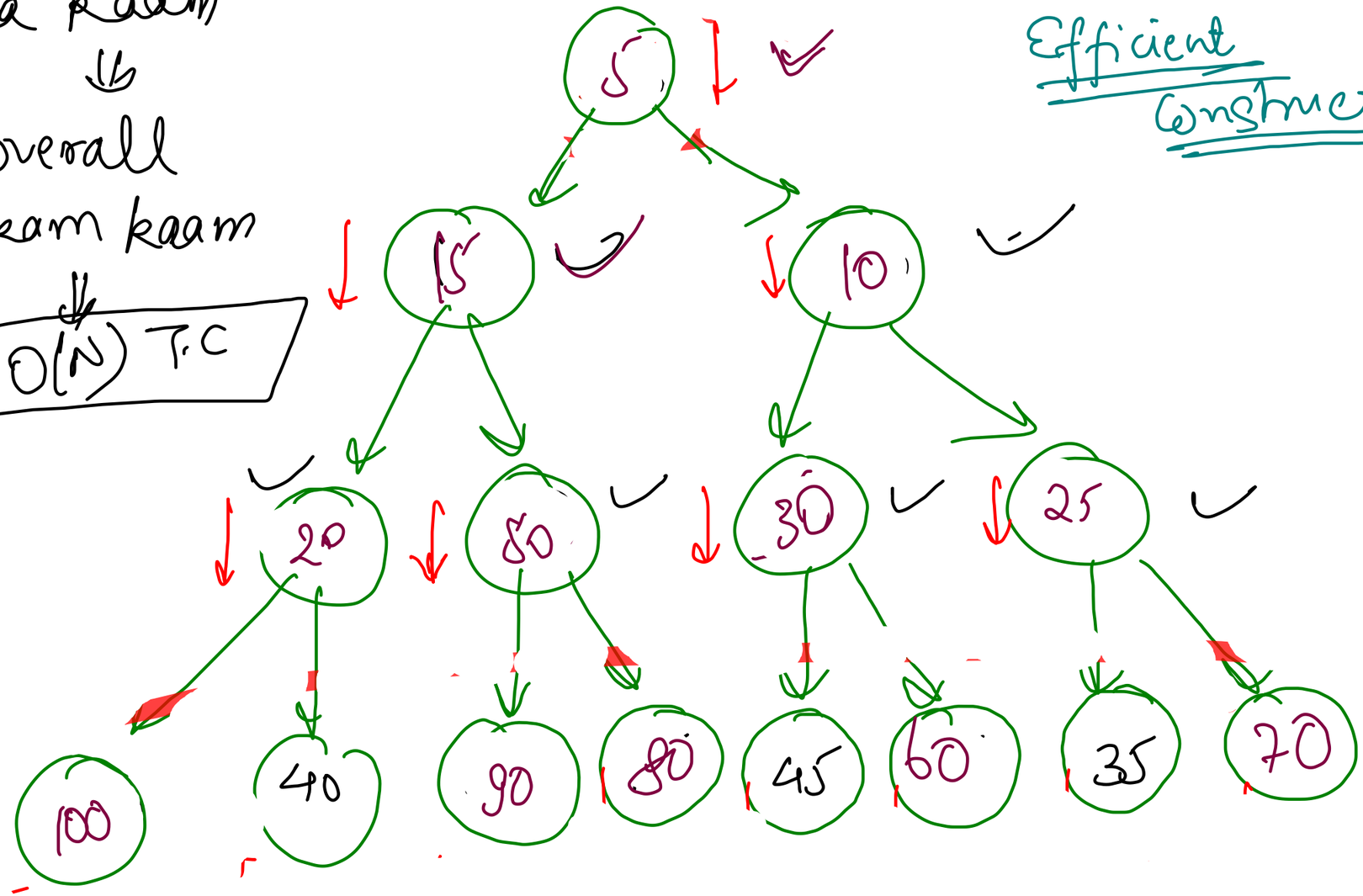
⇓

overall kam kaam

⇓

O(N) T.C

Efficient Constructor



AGP

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

$$+2 * T(n) = \quad 2 \cdot h + 2^2 \cdot (h-1) + 2^3 \cdot (h-2) + \dots + 2^h \cdot 1$$

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

$h = \log_2 n$

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

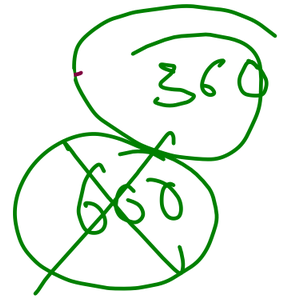
$$= 2^{\log_2 n} - 1 - \log_2 n$$

$$= n - \log_2 n - 1$$

$O(n)$

```java
// 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 ⇒ 3-4 ques

⇒ 3-4 ques Algodiss

⇒ 1-2 complete

MTW

3 months

5 classes * 4 weeks

= 20 classes

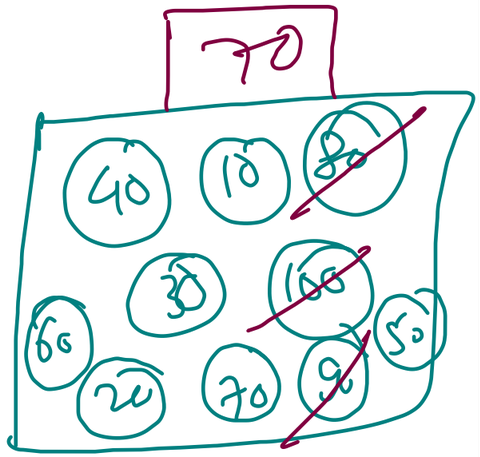
20 * 3 = 60 classes

60 + 6 = 360

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

Approach1 :- Max Heap

{ 100, 90, 80 }



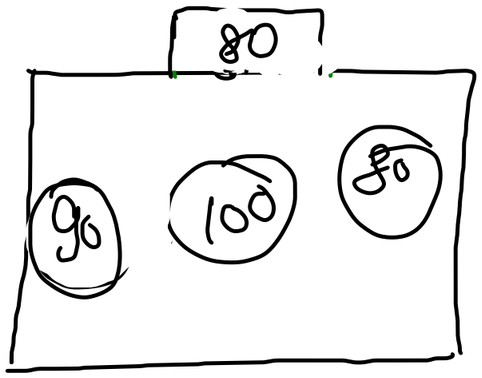
$O(N + K \log N)$

```java
public static void approach1(ArrayList<Integer> arr, int k){
    PriorityQueue<Integer> pq = new PriorityQueue<>(Collections.reverseOrder());

    // o(n)
    pq.addAll(arr);

    // o(klogN)
    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 } $k=3$

min heap

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




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

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

    // O((n-k) logk)
    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());
    }
}
```

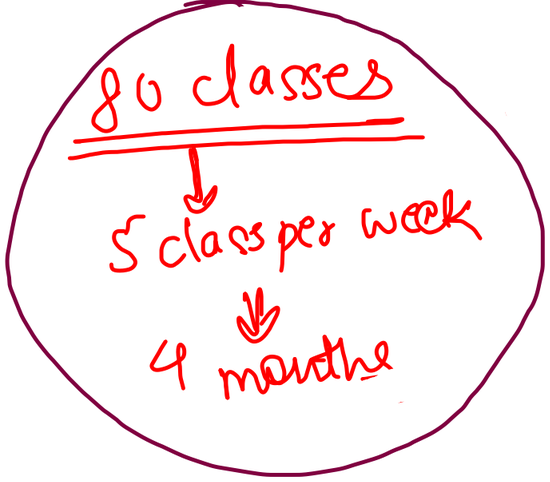
Today's Evening Questions

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

level 2 & level 3

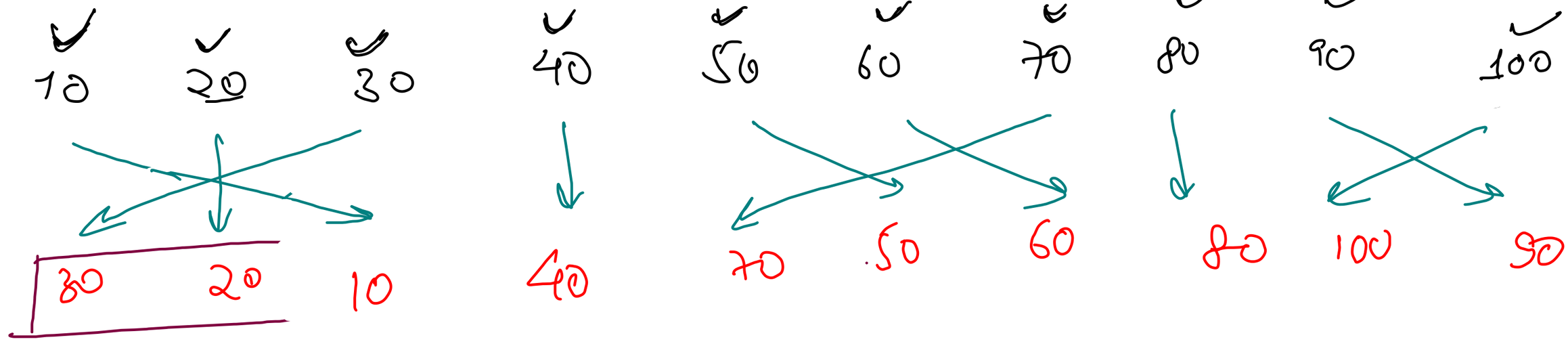
{ level up (300 Questions) + Interview Prep (300 Ques) }

- Arrays & Strings { searching, sorting, greedy, 2 pointers } ⇒ ±15 classes
- Recursion & Backtracking ⇒ ±10 classes
- Stack & Queue ⇒ ±5 classes
- linked list ⇒ ±5 classes
- Trees (Binary Tree, BST) ⇒ ±10 classes
- Hashmap & Heap ⇒ ±5 classes
- Graphs ⇒ ±8 classes
- Dynamic Programming ⇒ ±15 classes



- 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 patter matching (1) class

k Sorted Array [idx-k, idx+k]



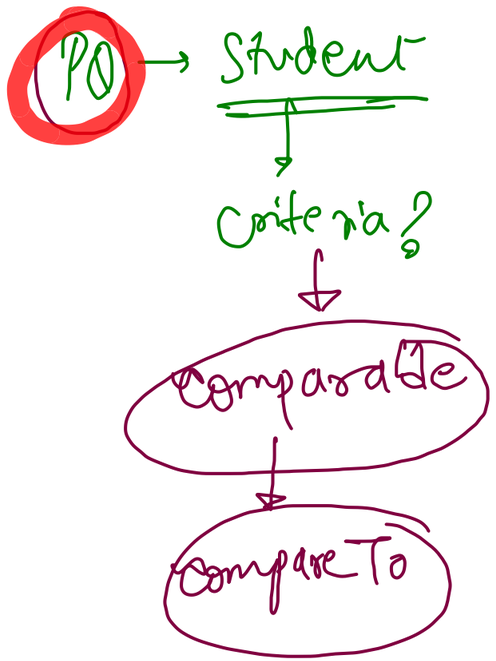
brute force :- Array sort : NlogN

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

→ create priority queue of (k+1) elements {sliding window}

{push 1 ele
pop 1 ele}

Comparable & Comparator



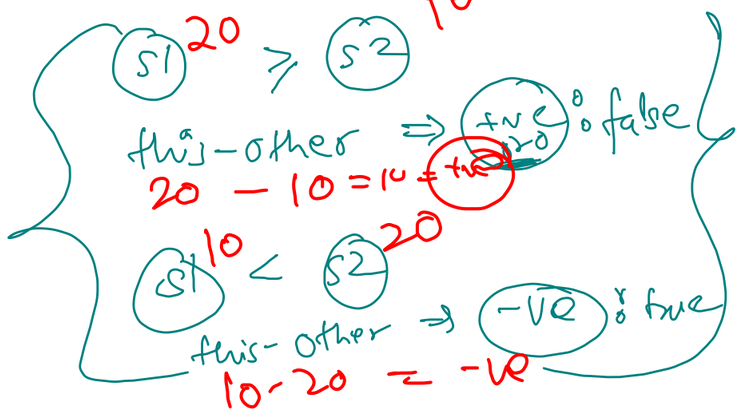
this.prop - other.prop

smaller value

⇓

higher priority

min heap



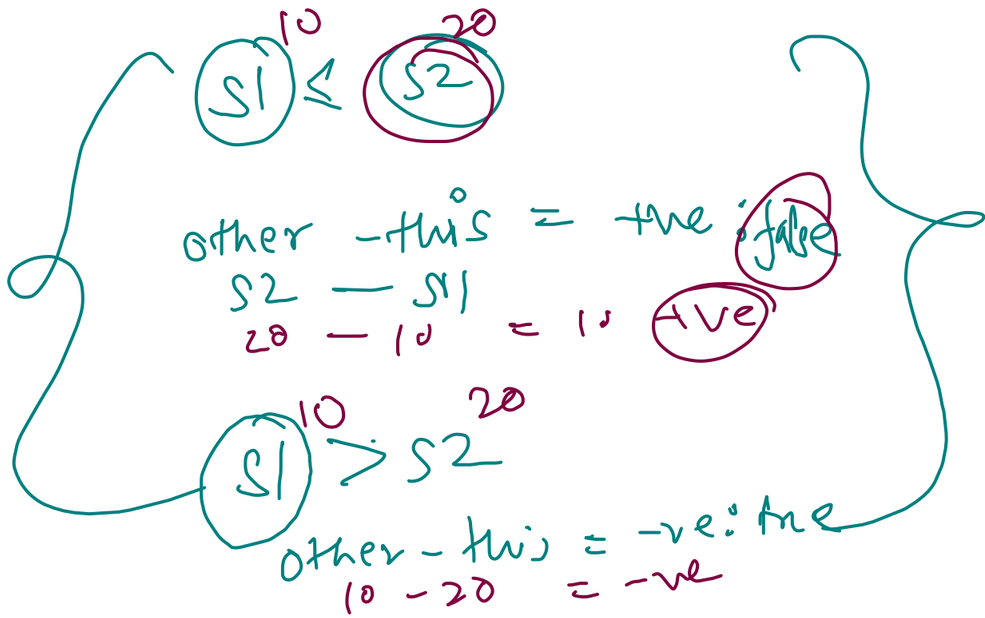
other.prop - this.prop

higher value

⇓

higher priority

max heap



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) Comparator

isSmaller

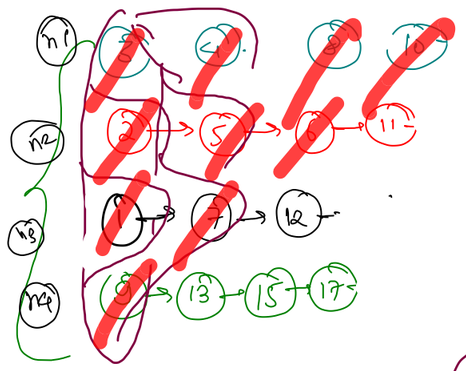
isSmaller

```java
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

```java
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 arrays



Brute force:
merge 2 at a time

$O(n_1) + O(n_1+n_2) + O(n_1+n_2+n_3) + O(n_1+n_2+n_3+n_4) + \dots + O(n_1+n_2+\dots+n_c)$

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

$N*k + N*(k-1) + N*(k-2) + \dots N*1$

$= N*\{1+2+3+\dots k\}$

$= O\left(\frac{N*k^2}{2}\right)$

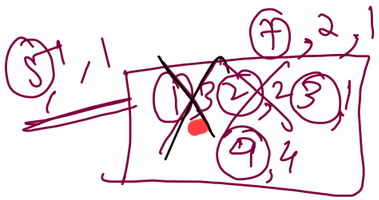
30-40 mins

5-10 mins
15 mins

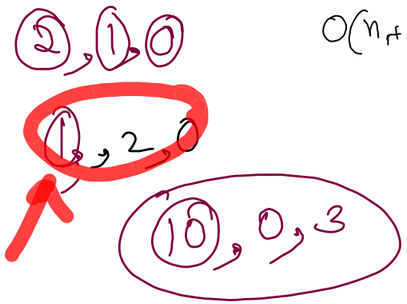
15 min

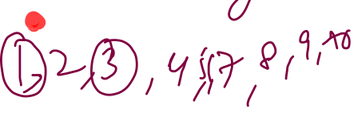
2 hr

9-10 hr



log k

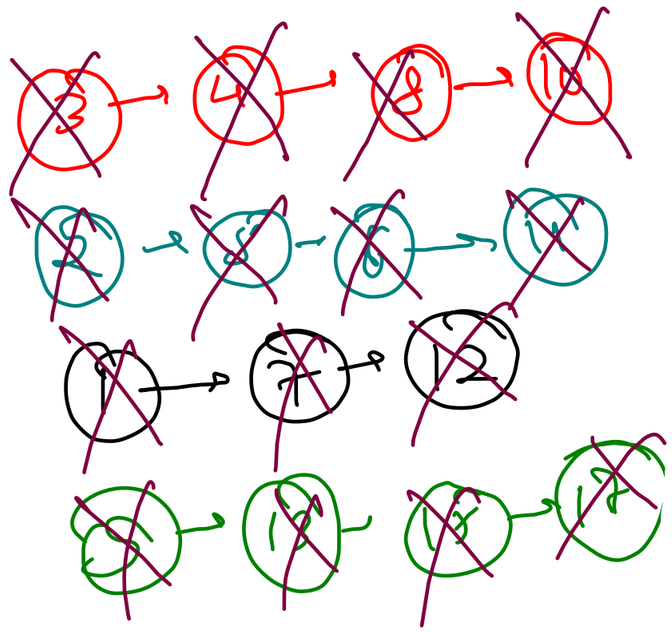




cl → { data, listIdx, dataIdx }

⇒ push k element :→ 0^{th} element of each array(:)z

```
while( q.size(
```



```java
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;
}
```