Priority Queue

Queue → FIFO — highest priority → first insert

Stack → LIFO — highest priority → last insert

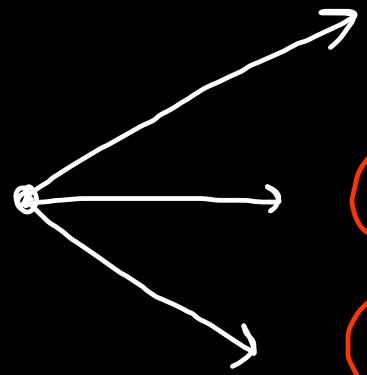
highest priority → custom sorting (max/min)

Priority Queue (abstract datatype) → Binary Heap (concrete data & structure)

Binary Heap → Min Heap (by default)

peek()	add()	remove()
get highest priority element O(1) avg/worst	insert elements acc to priority O(logn) avg/worst	deletes highest priority element O(logn) avg/worst

Binary Heap

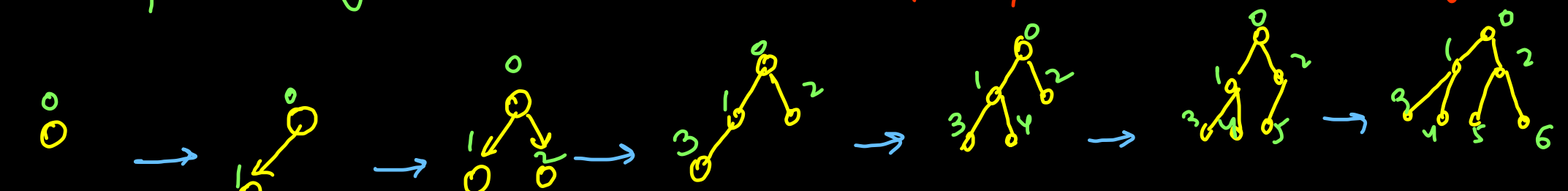


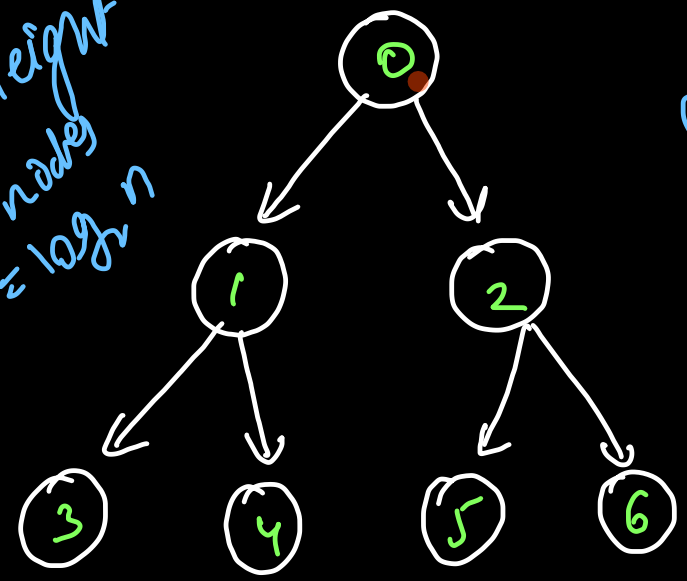
① Nodes will have at max 2 children

② Complete binary tree [Array]

③ Heap order Property

Complete binary tree { All levels (maybe except last) are full and in last level, all nodes are as left as possible }





Min height of n nodes = $\log_2 n$

0-based indexing = indices on CBT

root → idx

left = 2*idx + 1

right = 2*idx + 2

root → idx ⇝ parent = (idx-1)/2

Insertn

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

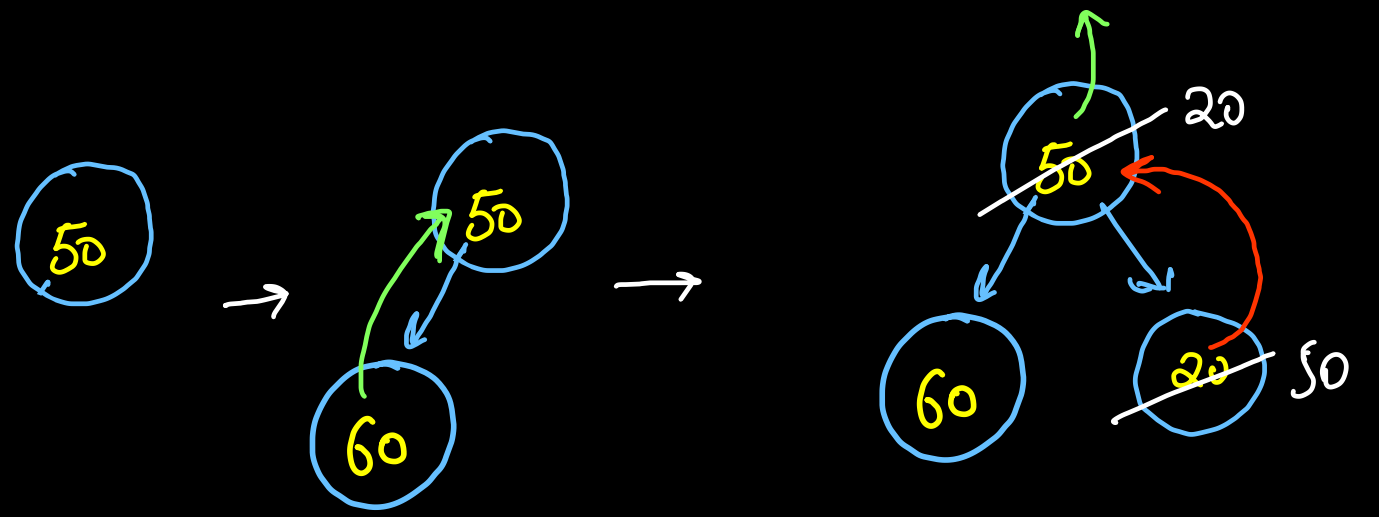
Heap order property

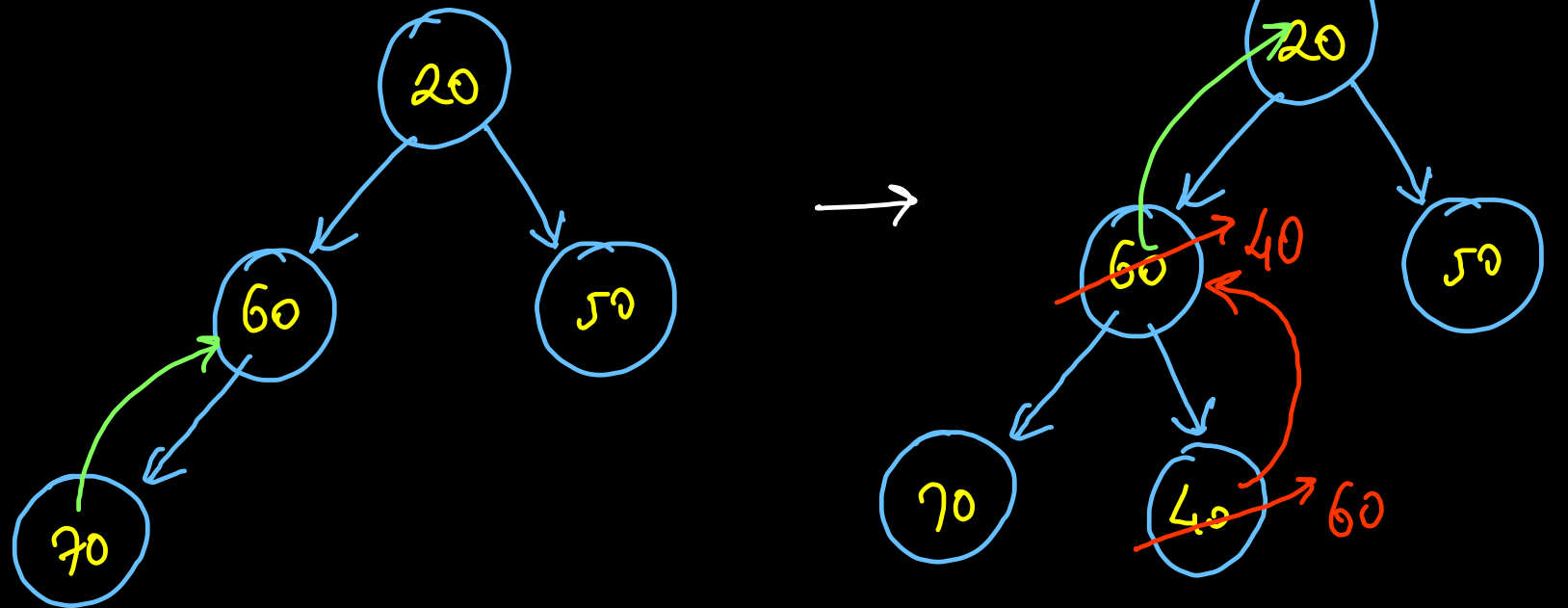
⇓

out of all nodes in a subtree

⇓

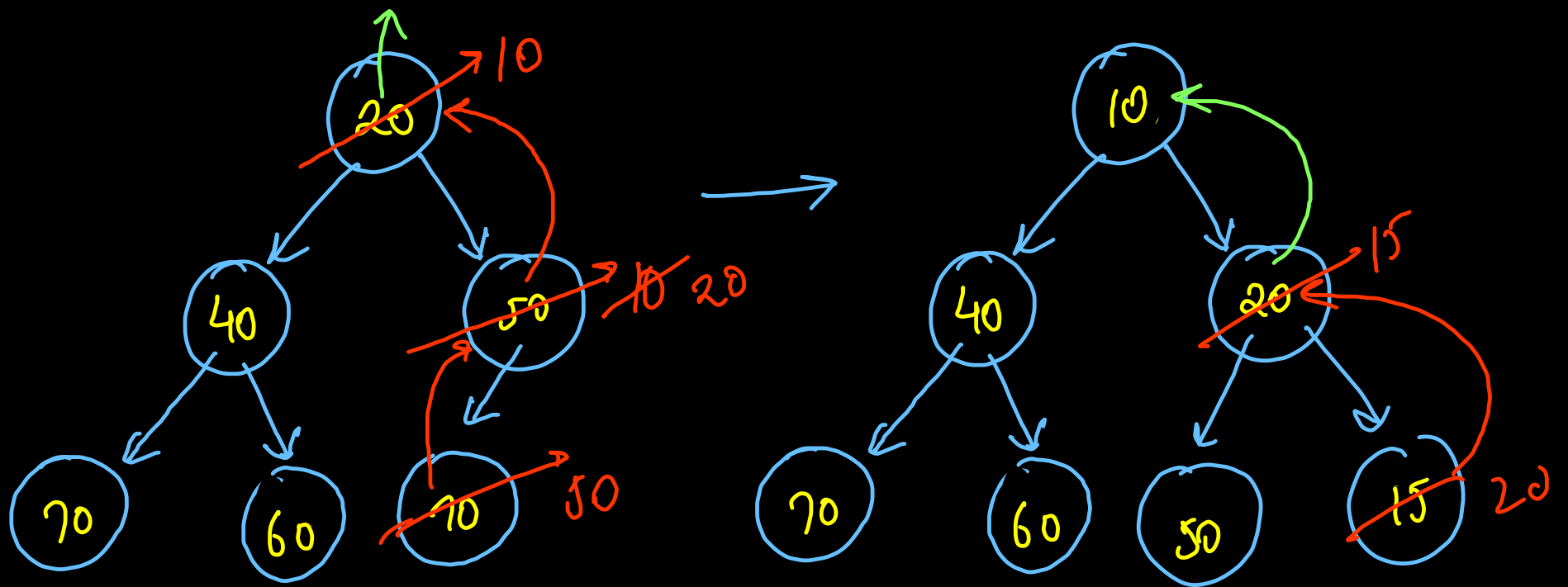
highest priority (minm value) should be at root





Insertn

0	1	2	3	4	5	6
50,	60,	20,	70,	40,	10,	15



upheapify()

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

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

  void upheapify(int idx){
      if(idx == 0) return;

      int par = (idx - 1) / 2;
      if(data.get(par) < data.get(idx)) return;

      Collections.swap(data, idx, par);
      upheapify(par);
  }
```

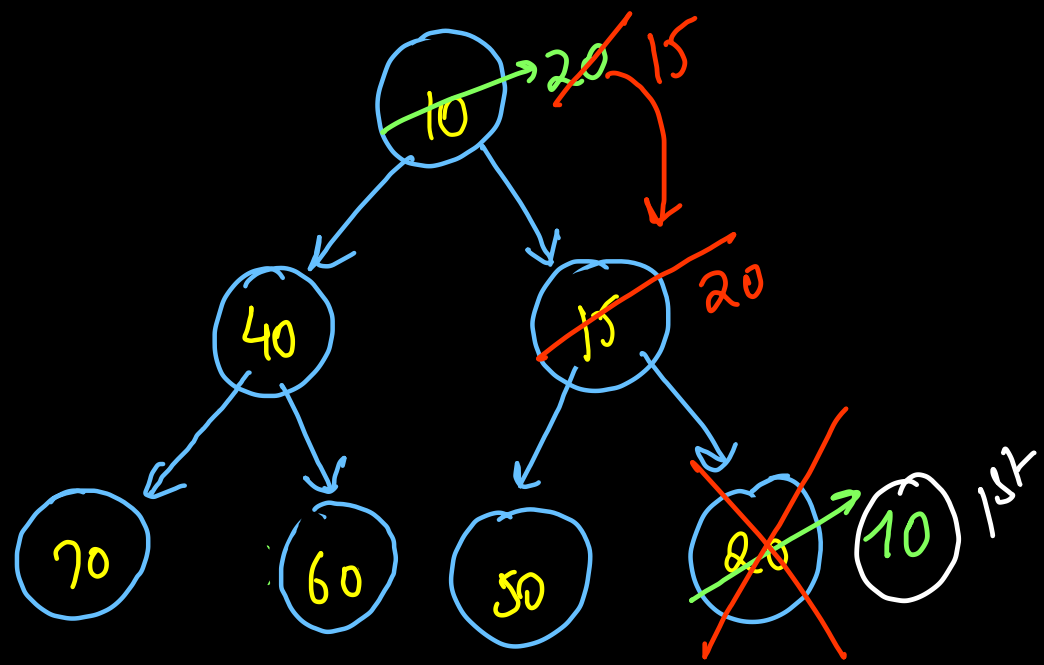
always balanced for CBT ★

O(height) → always balanced for CBT ★

= O(log n) → avg / worst

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

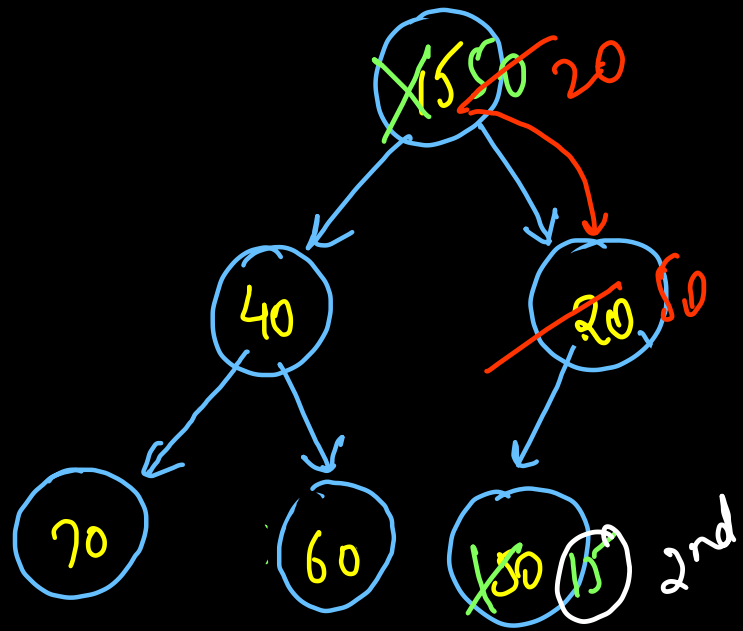
O(1) avg/worst

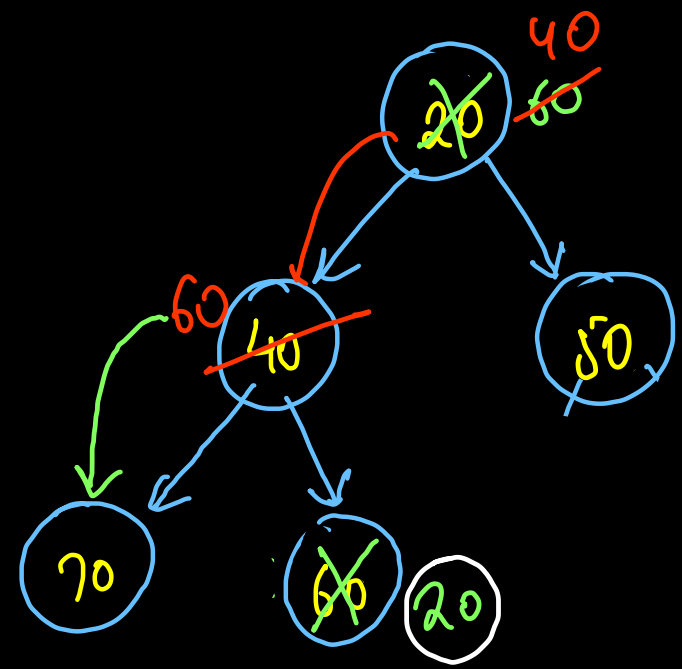


① swap(data[0], data[n-1]);

② delete the last node

③ down heapify(0)





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

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

  void upheapify(int idx){
      if(idx == 0) return;

      int par = (idx - 1) / 2;
      if(data.get(par) < data.get(idx)) return;

      Collections.swap(data, idx, par);
      upheapify(par);
  }
```

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

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

```java
public void downheapify(int idx){
    int l = 2 * idx + 1;
    int r = 2 * idx + 2;
    int min = idx;

    if(l < data.size() && data.get(l) < data.get(min))
        min = l;
    if(r < data.size() && data.get(r) < data.get(min))
        min = r;

    if(min == idx) return;
    Collections.swap(data, idx, min);
    downheapify(min);
}

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

    int val = data.get(0);
    Collections.swap(data, 0, data.size() - 1);
    data.remove(data.size() - 1);
    downheapify(0);
    return val;
}
```

Insertn data

0	1	2	3	4	5	6	7	8	9	10
50,	60,	20,	70,	40,	10,	30,	80,	25,	15,	35

```java
public PriorityQueue(int[] arr){
    // for(int val: data) add(val);
    // O(N * log N)

    for(int val: arr) data.add(val); // Complete Binary Tree
    for(int idx = data.size() - 1; idx >= 0; idx--)
        downheapify(idx); // Heap Order Property
}
```

O(n) time

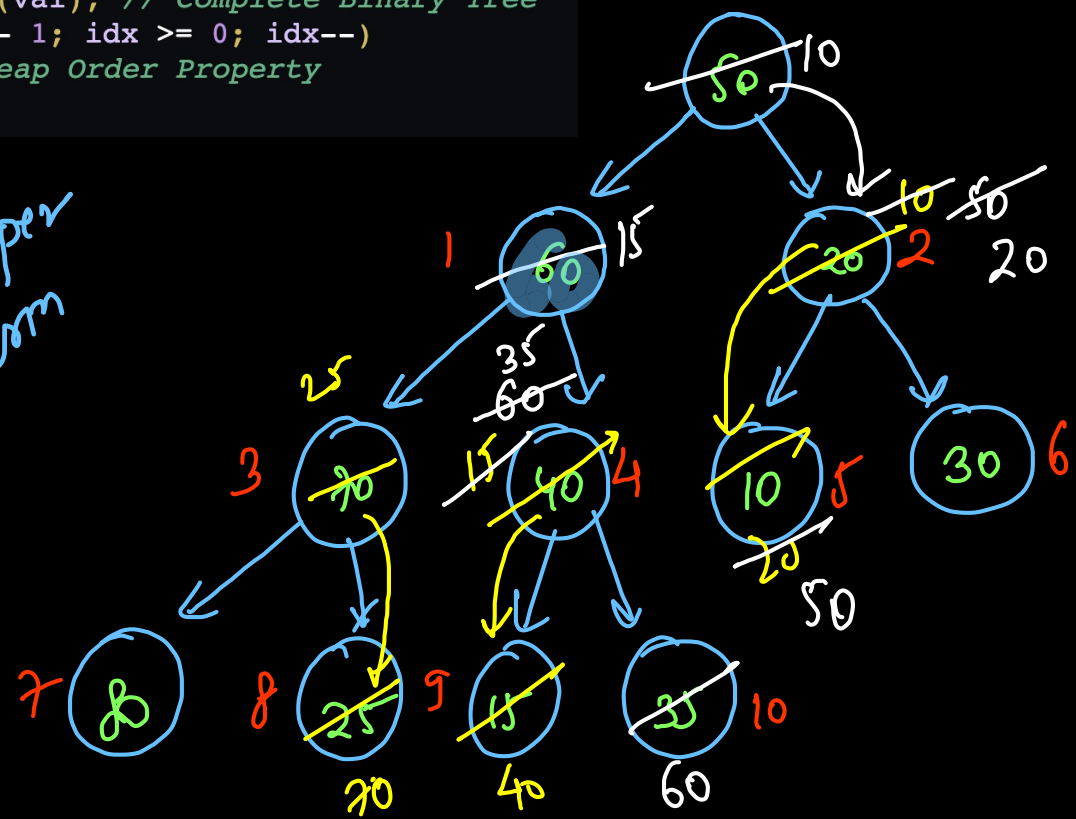
↳ O(1) const per insertn

insertn

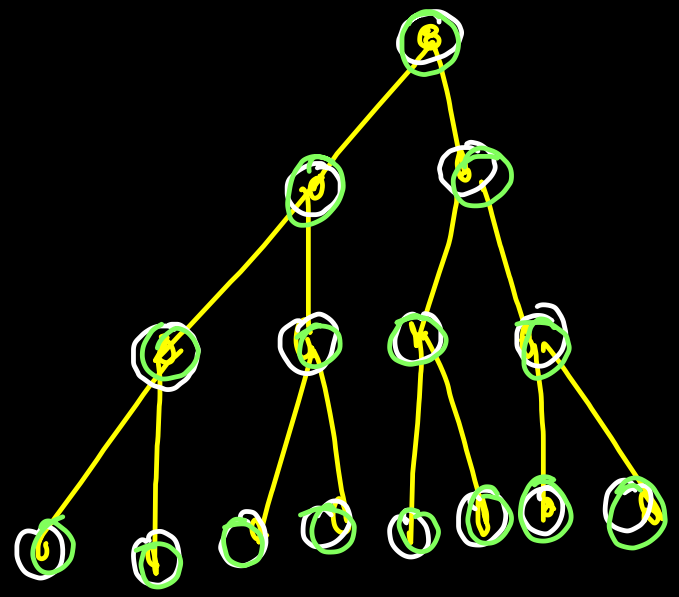
↓

upheapify nlogn

downheapify



downheapify(4)
downheapify(3)
downheapify(2)
downheapify(1)
downheapify(0)



Upheapify

$1+2+2+3+3+3+3+4+4+4+4+4+4+4+4$

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

$=$ AGP $\Rightarrow O(n\log n)$

downheapify

$= 8\times 1 + 4\times 2 + 2\times 3 + 1\times 4$

$= 2^h\times 1 + 2^{h-1}\times 2 + 2^{h-2}\times 3 + 2^{h-4}\times 4 + \cdots$

$\approx O(n)$

Heap Sort → Based on heap

③ Construct binary heap (Insert all n times)
↳ nlogn
+
Pop n elements (sorted order)
↳ nlogn

① Inplace (can be made) ② not stable

```java
class Solution
{
    ArrayList<Integer> data;

    public void downheapify(int idx){}

    public int delete() {
        int val = data.get(0);

        Collections.swap(data, 0, data.size() - 1);
        data.remove(data.size() - 1);

        downheapify(0);
        return val;
    }

    public void heapSort(int arr[], int n) {
        data = new ArrayList<>();

        for(int val: arr) data.add(val);

        for(int idx = n - 1; idx >= 0; idx--)
            downheapify(idx);

        for(int idx = 0; idx < n; idx++)
            arr[idx] = delete();
    }
}
```

Divide & Conquer

QuickSort

- O(nlogn) avg, $O(n^2)$ worst
- not stable (partitioning)
- in-place (excluding recursion)

MergeSort

- O(nlogn) avg/worst
- not inplace (merging)
- stable

```
void upheapify(int idx){
    if(idx == 0) return;

    int par = (idx - 1) / 2;
    if(data.get(par).compareTo(data.get(idx)) < 0) return;

    Collections.swap(data, idx, par);
    upheapify(par);
}
```

```
public void downheapify(int idx){
    int l = 2 * idx + 1;
    int r = 2 * idx + 2;
    int min = idx;

    if(l < data.size() && data.get(l).compareTo(data.get(min)) < 0)
        min = l;
    if(r < data.size() && data.get(r).compareTo(data.get(min)) < 0)
        min = r;

    if(min == idx) return;
    Collections.swap(data, idx, min);
    downheapify(min);
}
```

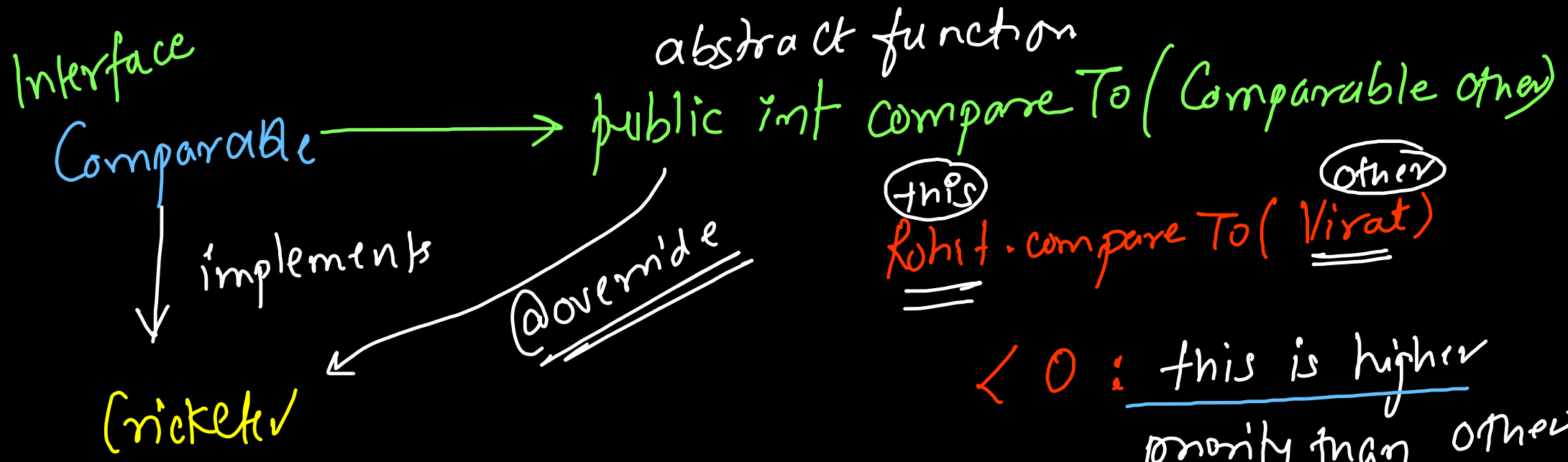
```java
public static class PriorityQueue<T> {
  ArrayList<T> data = new ArrayList<>();

  public PriorityQueue(){}

  public PriorityQueue(T[] arr){
      // for(int val: data) add(val);
      // O(N * log N)

      for(T val: arr) data.add(val); // Complete Binary Tree
      for(int idx = data.size() - 1; idx >= 0; idx--)
          downheapify(idx); // Heap Order Property
  }

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

  void upheapify(int idx){
      if(idx == 0) return;

      int par = (idx - 1) / 2;
      Comparable p1 = (Comparable)(data.get(idx));
      Comparable p2 = (Comparable)(data.get(par));

      if(p2.compareTo(p1) < 0) return;

      Collections.swap(data, idx, par);
      upheapify(par);
  }
```

```java
  public T peek() {
      if(data.size() == 0) {
          System.out.println("Underflow");
          return null;
      }
      return data.get(0);
  }

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

```java
public void downheapify(int idx){
    int l = 2 * idx + 1;
    int r = 2 * idx + 2;
    int min = idx;

    if(l < data.size()){
        Comparable p1 = (Comparable)(data.get(l));
        Comparable p2 = (Comparable)(data.get(min));
        if(p1.compareTo(p2) < 0) min = l;
    }

    if(r < data.size()){
        Comparable p1 = (Comparable)(data.get(r));
        Comparable p2 = (Comparable)(data.get(min));
        if(p1.compareTo(p2) < 0) min = r;
    }

    if(min == idx) return;
    Collections.swap(data, idx, min);
    downheapify(min);
}

public T remove() {
    if(data.size() == 0) {
        System.out.println("Underflow");
        return null;
    }

    T val = data.get(0);
    Collections.swap(data, 0, data.size() - 1);
    data.remove(data.size() - 1);
    downheapify(0);
    return val;
}
```

Comparable (by default) → CompareTo

```java
public static void main(String[] args) throws Exception {
  Integer[] arr = {50, 60, 20, 70, 40, 10};
  PriorityQueue<Integer> pq = new PriorityQueue(arr);

  while(pq.size() > 0)
      System.out.print(pq.remove() + " "); → Increasing order

  String[] arr2 = {"Rohit", "Virat", "Surya", "Dhoni", "Rishabh"};
  PriorityQueue<String> pq2 = new PriorityQueue(arr2);

  while(pq2.size() > 0)
      System.out.print(pq2.remove() + " "); → Lexicographical No
}
```




```java
class Cricketer implements Comparable<Cricketer>{
    String name;
    Integer runs;
    Integer age;

    public Cricketer(String name, Integer runs, Integer age){
        this.name = name;
        this.age = age;
        this.runs = runs;
    }

    @Override
    public int compareTo(Cricketer other){

    }
}
```

< 0 : this is higher priority than other

= 0 : both have same priority

> 0 : other is higher priority than this

rohit.compareTo(virat)

```java
@Override
public int compareTo(Cricketer other){
    // younger age -> higher priority
    // if(this.age > other.age){
    //     return +1;
    // } else if(this.age < other.age){
    //     return -1;
    // } else return 0;

    // more runs -> higher priority
    if(this.runs > other.runs){
        return -1;
    } else if(this.runs < other.runs){
        return +1;
    } else if(this.age < other.age){
        return +1;
    } else return -1;
}
```

age → young → more priority → +1

runs → more → more priority → −1

lexicographically sorted → −1

Java Inbuilt Priority queue

```java
PriorityQueue<Integer> pq = new PriorityQueue<>();
// Min Heap
pq.add(30);
pq.add(20);
pq.add(50);
pq.add(40);
pq.add(10);
while(pq.size() > 0)
    System.out.print(pq.remove() + " ");
System.out.println();

pq = new PriorityQueue<>(Collections.reverseOrder());
// Max Heap
pq.add(30);
pq.add(20);
pq.add(50);
pq.add(40);
pq.add(10);
while(pq.size() > 0)
    System.out.print(pq.remove() + " ");
System.out.println();
```

10,20,30,40,50

inbuilt comparator

50,40,30,20,10

```java
PriorityQueue<String> pq2 = new
        PriorityQueue<>(Collections.reverseOrder());
// Lexicographically Max Heap

pq2.add("Dhoni");
pq2.add("Virat");
pq2.add("Surya");
pq2.add("Ishan");
pq2.add("Rohit");

while(pq2.size() > 0){
    System.out.print(pq2.remove() + " ");
}
System.out.println();
```

```
Virat Surya Rohit Ishan Dhoni
```

*Interface

```java
class Cricketer implements Comparable<Cricketer>{
    String name;
    Integer runs;
    Integer age;

    public Cricketer(String name, Integer runs, Integer age){
        this.name = name;
        this.age = age;
        this.runs = runs;
    }

    @Override
    public int compareTo(Cricketer other){

    }
}
```

class CricketerAgeComparator implements Comparator<Cricketer>{

public int Compare(Cricketer t, Cricketer o){

}

}

<0 : t higher priority

=0 : equal priority

>0 : o higher priority

```java
class Cricketer implements Comparable<Cricketer>{
    String name;
    Integer runs;
    Integer age;

    public Cricketer(String name, Integer runs, Integer age){
        this.name = name;
        this.age = age;
        this.runs = runs;
    }

    // By Default Sorting Mechanism
    @Override
    public int compareTo(Cricketer other){
        // Higher Runs -> Higher Priority
        if(this.runs > other.runs){
            return -1;
        } else if(this.runs < other.runs){
            return +1;
        } return 0;
    }
}
```

```java
PriorityQueue<Cricketer> pq3 = new PriorityQueue<>();
pq3.add(new Cricketer("Rohit", 264, 34));
pq3.add(new Cricketer("Virat", 183, 33));
pq3.add(new Cricketer("Surya", 150, 32));
pq3.add(new Cricketer("Dhoni", 183, 40));

while(pq3.size() > 0){
    System.out.println(pq3.remove().name + " ");
    // Default Sorting: Comparable: Higher Runs
}
System.out.println();
```

↓ Rohit 264, Dhoni 183, Virat 183, Surya 150

```java
class CricketerAgeComparator implements Comparator<Cricketer>{
    // Young Age -> higher priority
    @Override
    public int compare(Cricketer t, Cricketer o){
        if(t.age < o.age) {
            return -1;
        } else if(t.age > o.age){
            return +1;
        } else return 0;
    }
}

class CricketerNameComparator implements Comparator<Cricketer>{
    // Lexicopgraphical Order
    @Override
    public int compare(Cricketer t, Cricketer o){
        if(t.name.compareTo(o.name) < 0) {
            return -1;
        } else if(t.name.compareTo(o.name) > 0){
            return +1;
        } else return 0;
    }
}
```

```java
PriorityQueue<Cricketer> pq4 = new PriorityQueue<>(new CricketerAgeComparator());
pq4.add(new Cricketer("Rohit", 264, 34));
pq4.add(new Cricketer("Virat", 183, 33));
pq4.add(new Cricketer("Surya", 150, 32));
pq4.add(new Cricketer("Dhoni", 183, 40));

while(pq4.size() > 0){
    System.out.println(pq4.remove().name + " ");
    // Default Sorting: Age Comparator: Young Age
}
System.out.println();
```

Surya, Virat, Rohit, Dhoni
32 33 34 40

```java
PriorityQueue<Cricketer> pq5 = new PriorityQueue<>(new CricketerNameComparator());
pq5.add(new Cricketer("Rohit", 264, 34));
pq5.add(new Cricketer("Virat", 183, 33));
pq5.add(new Cricketer("Surya", 150, 32));
pq5.add(new Cricketer("Dhoni", 183, 40));

while(pq5.size() > 0){
    System.out.println(pq5.remove().name + " ");
    // Default Sorting: Age Comparator: Young Age
}
```

Dhoni, Rohit, Surya, Virat

```java
class PriorityQueue<T> {
    ArrayList<T> data = new ArrayList<>();
    Comparator comparator;

    public PriorityQueue() {
    }

    public PriorityQueue(Comparator comparator) {
        this.comparator = comparator;
    }
```

```java
boolean swap(int i1, int i2){
    T p2 = data.get(i2);
    T p1 = data.get(i1);

    if(comparator == null){
        return ((Comparable)p1).compareTo((Comparable)p2) < 0;
    }
    return comparator.compare(p1, p2) < 0;
}
```

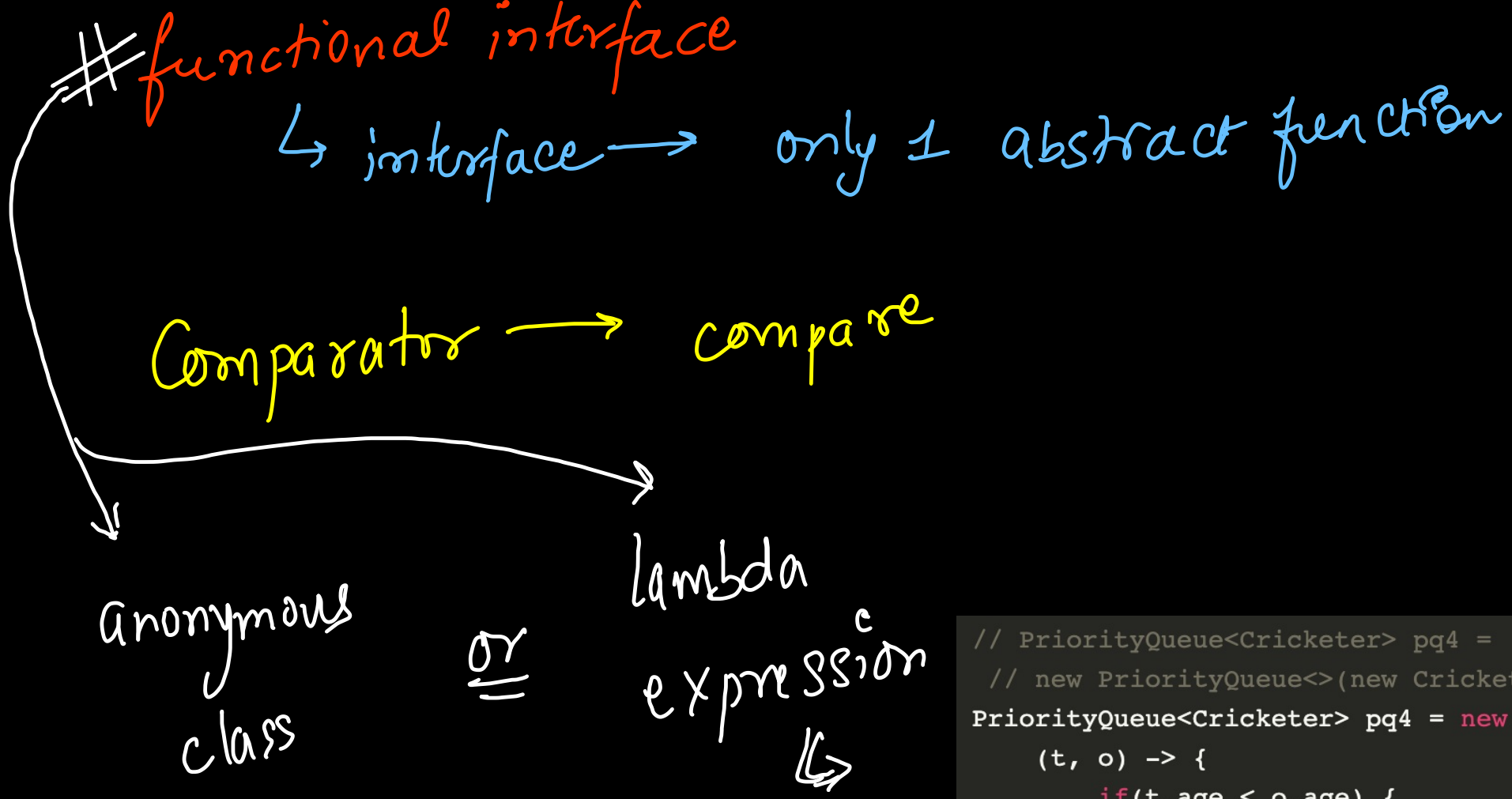
```java
void upheapify(int idx) {
    if (idx == 0)
        return;

    int par = (idx - 1) / 2;

    if (swap(idx, par) == false)
        return;

    Collections.swap(data, idx, par);
    upheapify(par);
}
```

You, 3 minutes ago • added codes

```java
public void downheapify(int idx) {
    int l = 2 * idx + 1;
    int r = 2 * idx + 2;
    int min = idx;

    if (l < data.size()) {
        if (swap(l, min) == true)
            min = l;
    }

    if (r < data.size()) {
        if (swap(r, min) == true)
            min = r;
    }

    if (min == idx)
        return;
    Collections.swap(data, idx, min);
    downheapify(min);
}
```

Difference Between Comparable and Comparator in Java

Comparable	Comparator
Comparable is an interface in Java.	Comparator is a functional interface in Java.
Comparable provides compareTo() method to sort objects.	Comparator provides compare() method to sort objects.
Comparable is a part of the Java.lang package.	Comparator is a part of the java.util package.
Comparable can be used for natural or default ordering.	Comparator can be used for custom ordering.
Comparable provides a single sorting sequence. Ex: Sort either by id or name	Comparator provides multiple sorting sequences. Ex. Sort by both id and name.
Comparable modifies the class that implements it.	Comparator doesn't modify any class.




```java
// PriorityQueue<Cricketer> pq4 =
 // new PriorityQueue<>(new CricketerAgeComparator());
PriorityQueue<Cricketer> pq4 = new PriorityQueue<>(
    (t, o) -> {
        if(t.age < o.age) {
            return -1;
        } else if(t.age > o.age){
            return +1;
        } else return 0;
    }
);
```

Greedy algorithm ⇒ local optimum

Fractional Knapsack

	salt	sugar	rice	cereal	Ghee
value	2000	500	100	200	600
weight	40	20	2	5	1

capacity = ~~25~~ ~~24~~ 17

secondary → minimize weight

primary → maximize value

1325

600 + 100 + 200 + $\frac{500}{20} \times 17$

① + ② + ⑤ + ⑰

$\frac{2000}{40} \times \frac{5}{25} = 1250$

㉕

	salt	sugar	rice	cereal	Ghee
value	200	500	100	2000	600
weight	40	20	2	50	1
value/weight	200/40 = 5	500/20 = 25	100/2 = 50	2000/50 = 40	600/1 = 600
			I_2	I_3	I_1

total capacity ⇒ 600rs + 100rs + 40*22rs

25kg — 1kg — 2kg — 22kg

```java
class VByWComparator implements Comparator<Item> {
    public int compare(Item t, Item o) {
        double p1 = (t.value * 1.0) / t.weight;
        double p2 = (o.value * 1.0) / o.weight;

        if(p1 < p2) return +1;
        else if(p1 > p2) return -1;
        else return 0;
    }
}

class Solution
{
    double fractionalKnapsack(int capacity, Item arr[], int n) {
        PriorityQueue<Item> pq = new PriorityQueue<>(new VByWComparator());
        for(int i = 0; i < n; i++) pq.add(arr[i]);

        double profit = 0.0;
        while(capacity > 0 && pq.size() > 0){
            Item i = pq.remove();

            if(capacity > i.weight){
                profit += i.value;
                capacity -= i.weight;
            } else {
                double currProfit = ((i.value * 1.0) / i.weight) * capacity;
                profit += currProfit;
                capacity = 0;
            }
        }

        return profit;
    }
}
```

Space → O(n) PQ

Time → O(nlogn) Greedy

nlogn

nlogn

→ higher priority → higher profit/weight

profit ↑ → weight ↓