Priority Queue

Queue → FIFO — highest priority → first insert

Stack → LIFO — highest priority → last insert

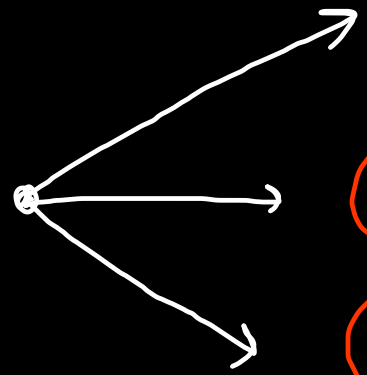
highest priority → custom sorting (max/min)

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

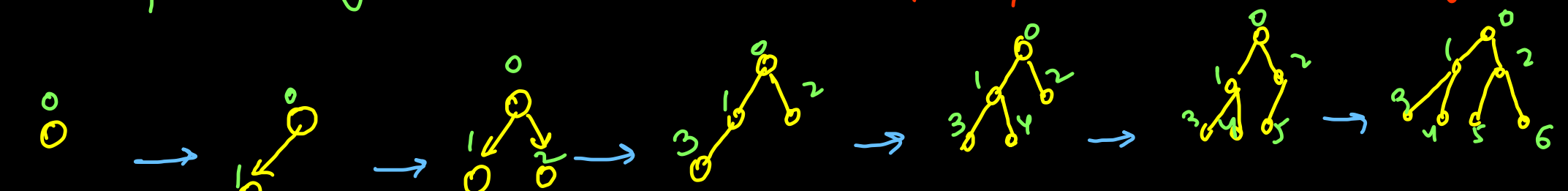


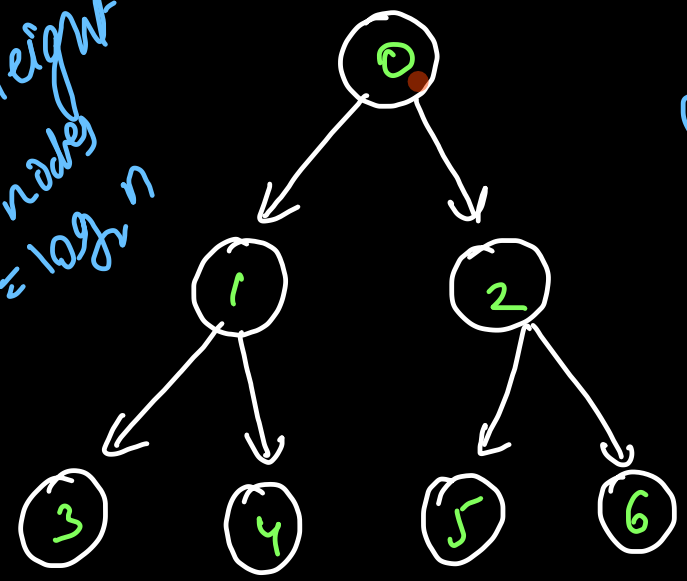
① Nodes will have at max 2 children

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

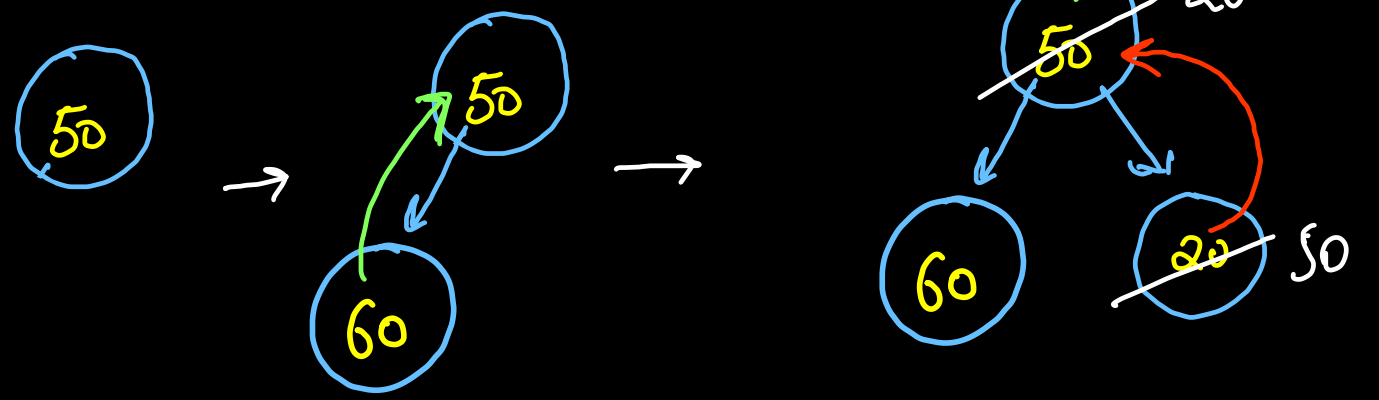
Heap order property

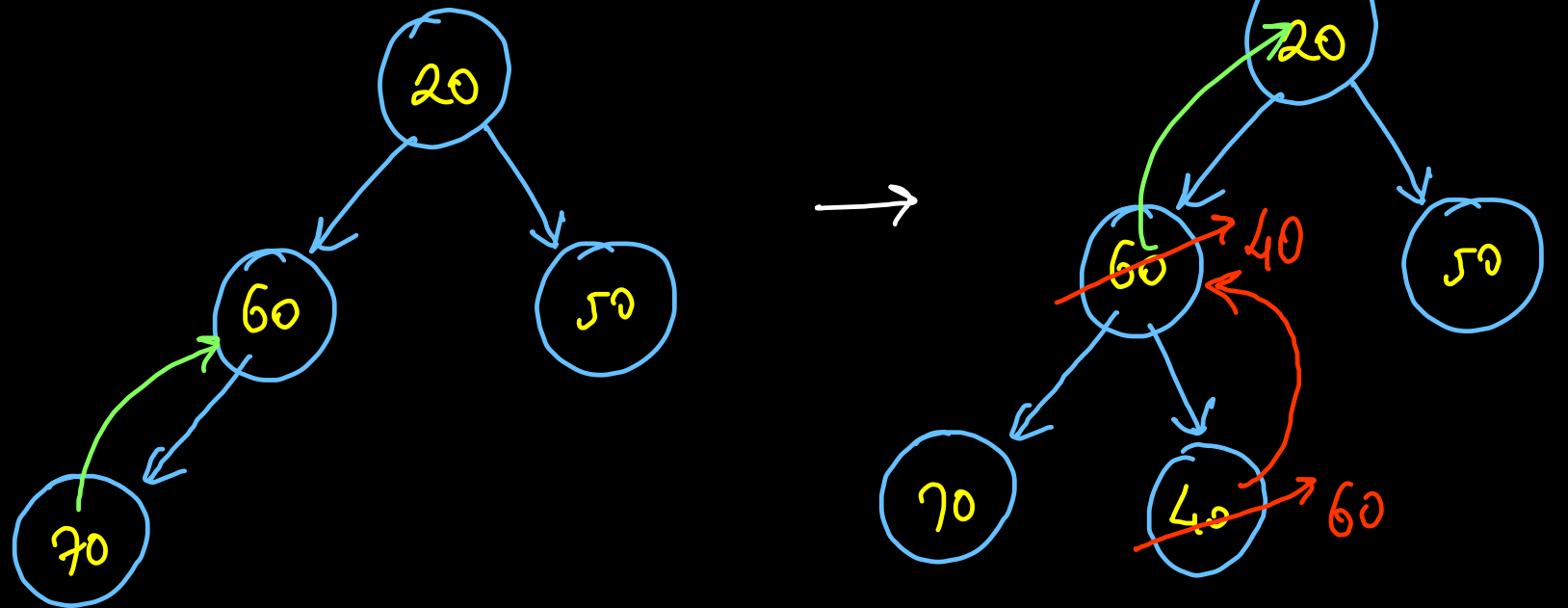
⇓

out of all nodes in a subtree

⇓

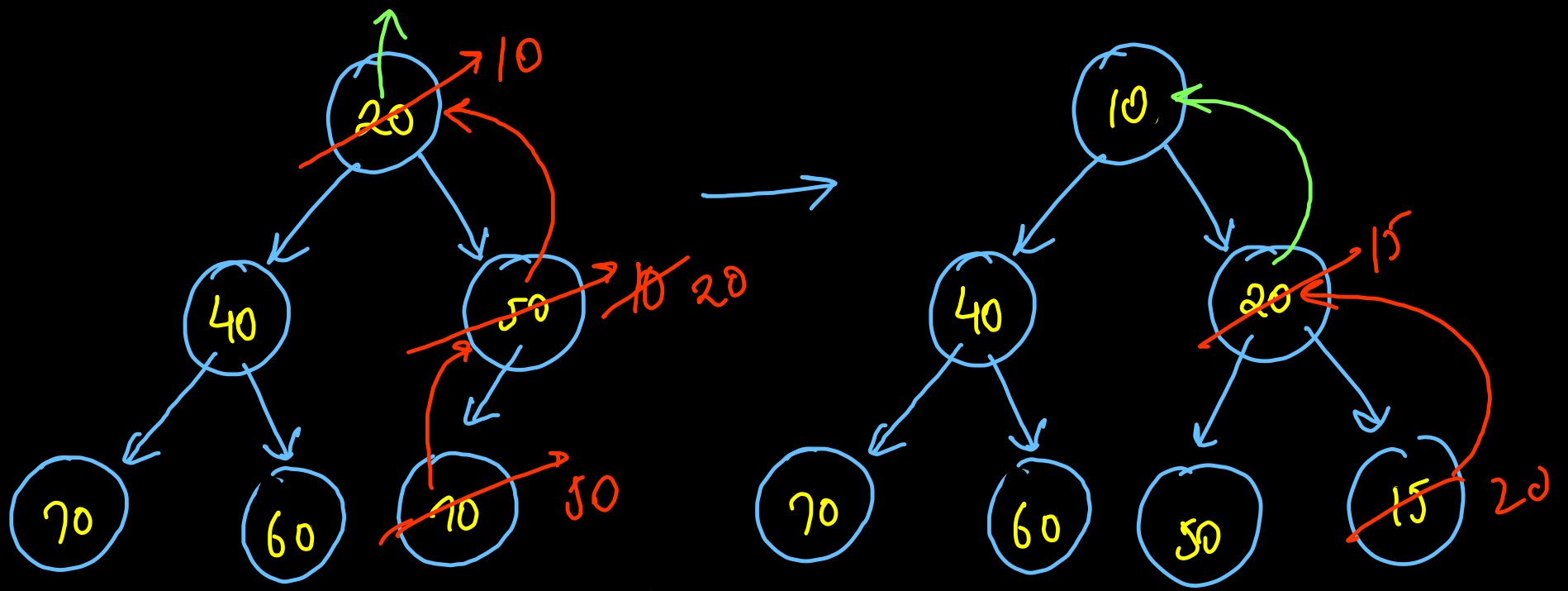
highest priority (minm value) should be at root





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);
  }
```

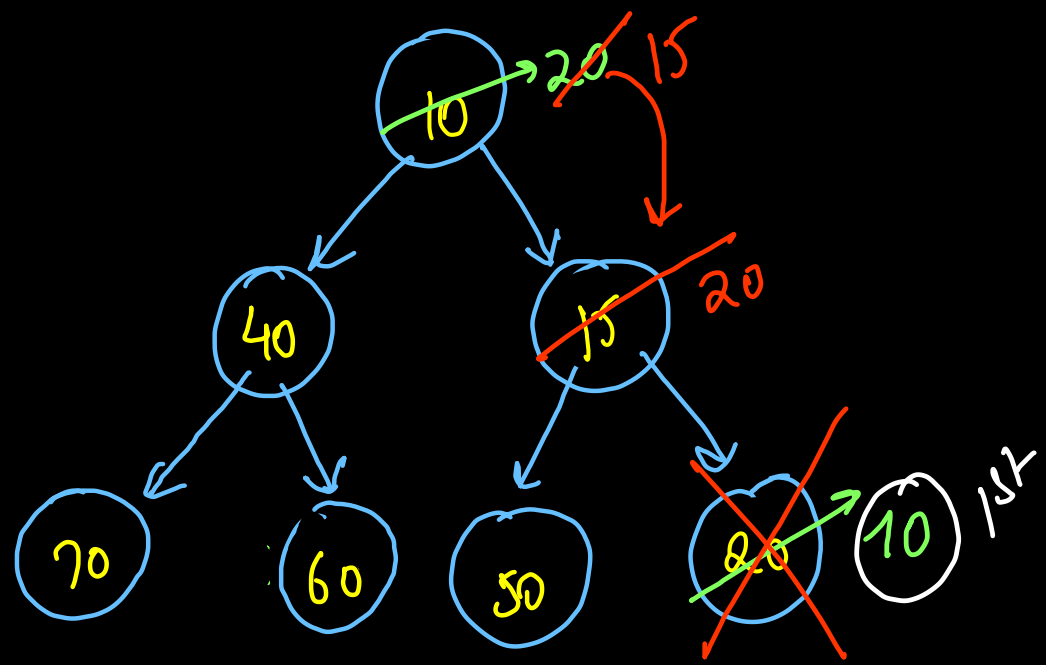
always balanced for CBT ★

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);
}
```

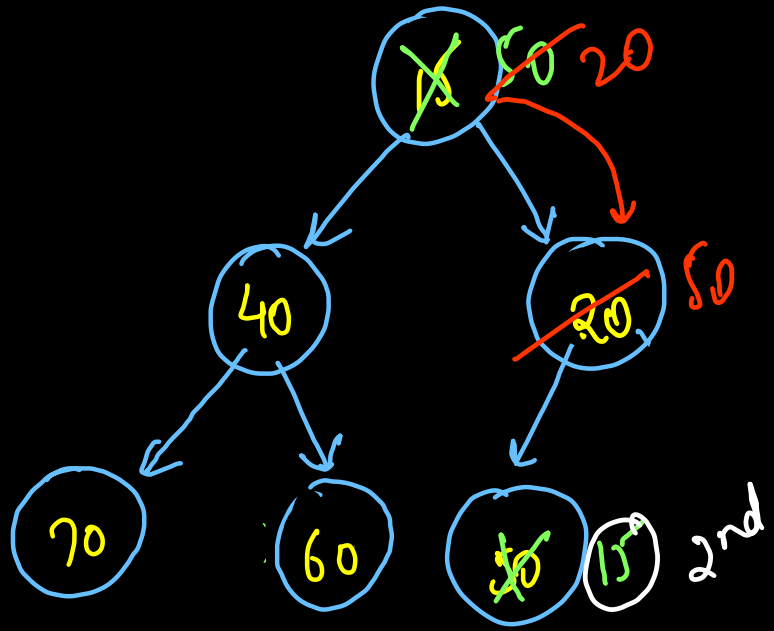
O(1) avg/worst

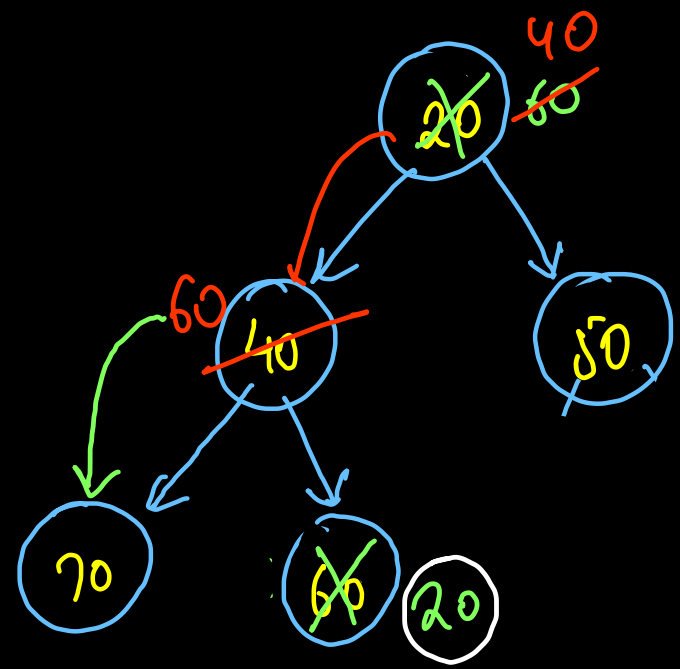


① swap(data[0], data[n-1]);

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;
}
```