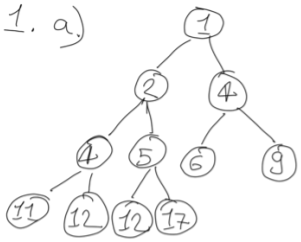
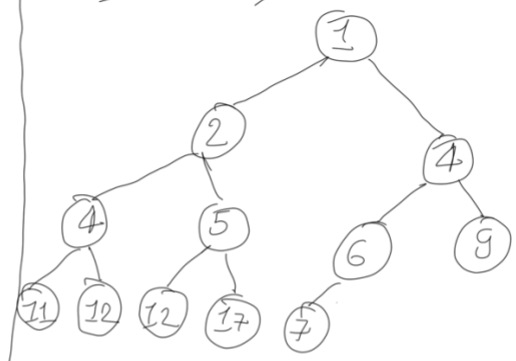
Answer of 1

1. a)



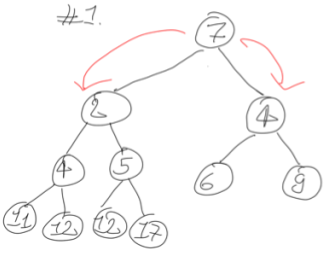
1. b) insert(7)



7 > 6, so everything is valid.

c) removeMin()

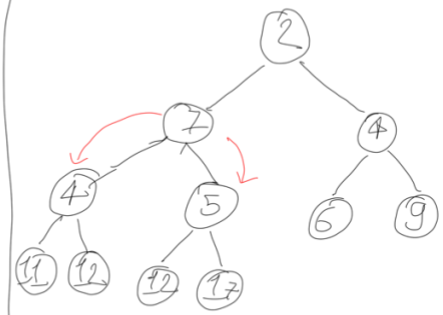
#1.



Remove 1 from heap.

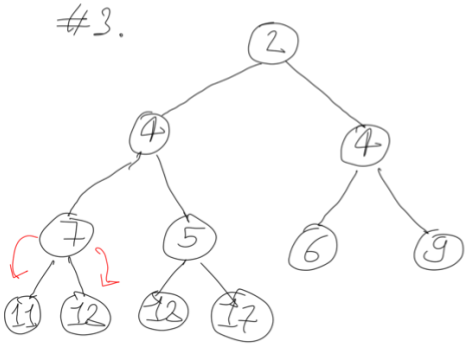
Heap is unbalanced, perform downheap as needed

#2.



Perform downheap as needed as 7 is still greater.

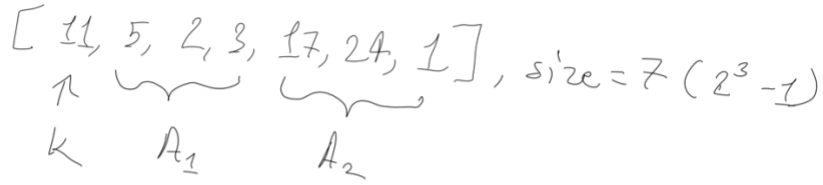
#3.

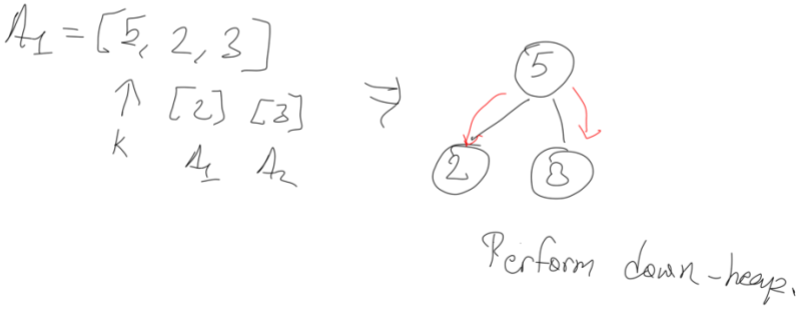


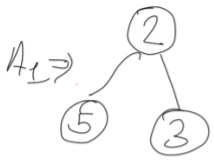
Now, the heap is balanced,
as 7 is less than its children.

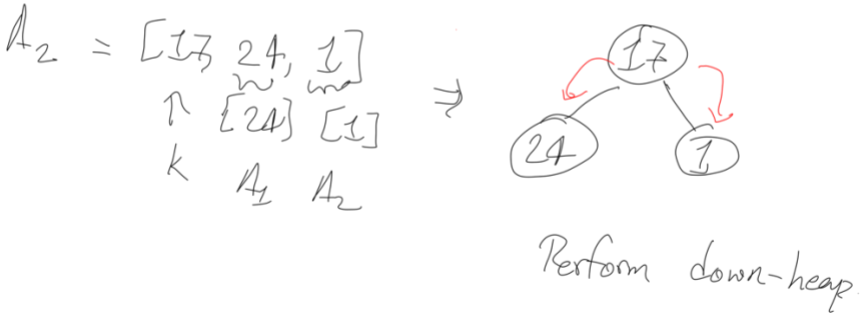
Answer of 2

2. Recursive Bottom up Heap for:

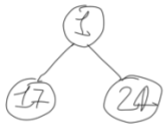




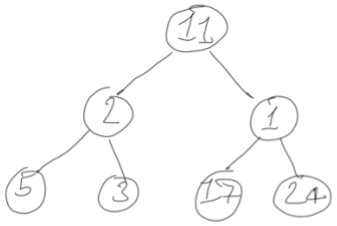




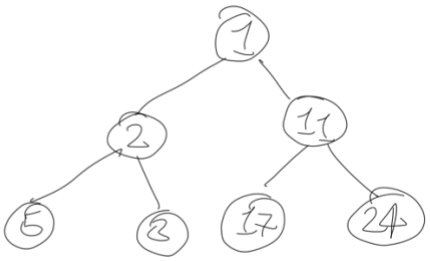
$A_2 \Rightarrow$



Now, create the tree with k, A_1 and A_2.

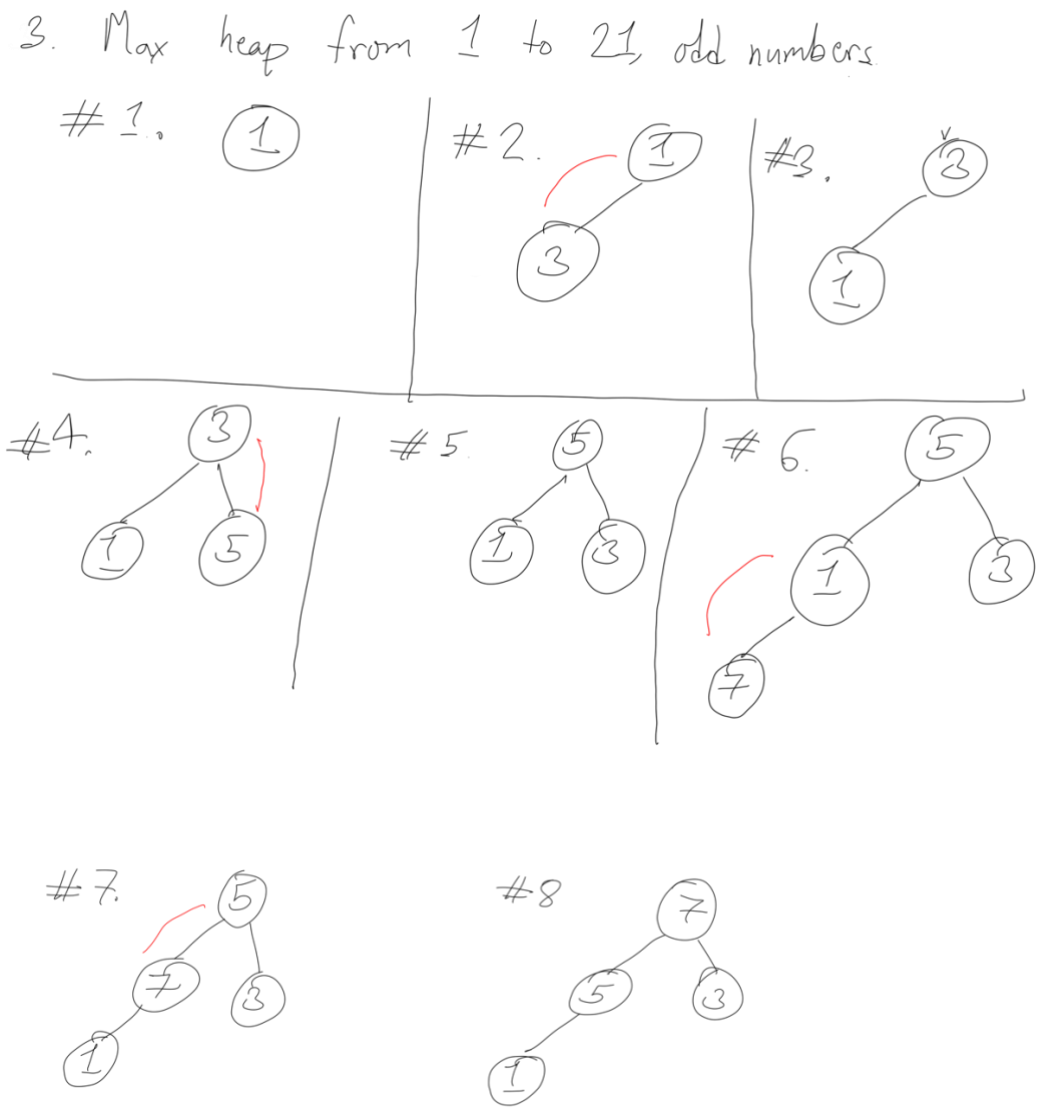


Perform down-heap.

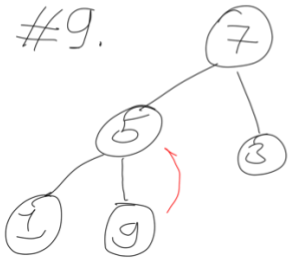


This will be the final heap.

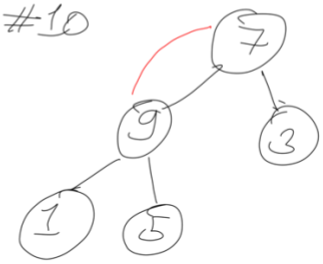
Answer of 3



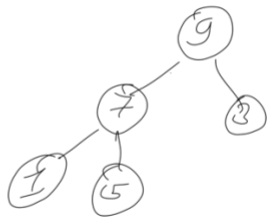
#9.



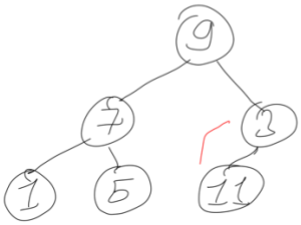
#10



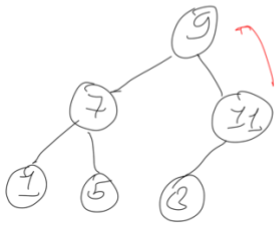
#11.



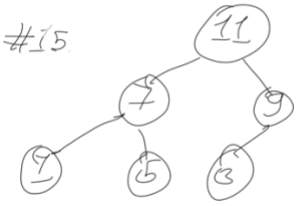
#12. insert(11)



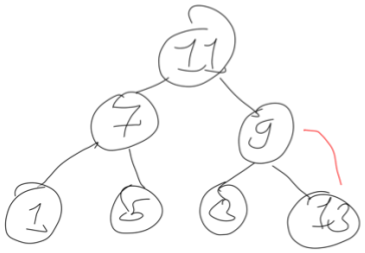
#14



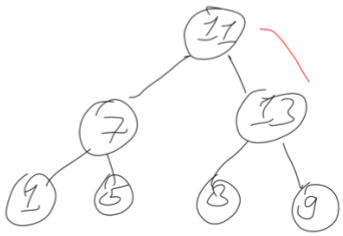
#15



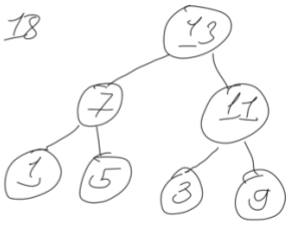
#16. insert(13)



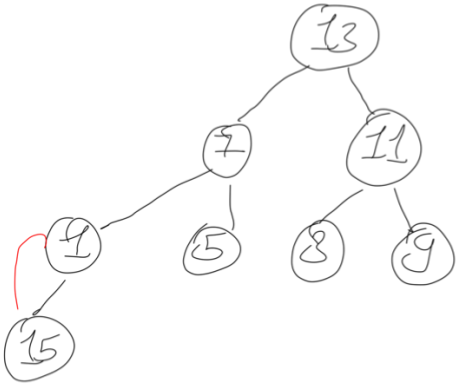
#17.



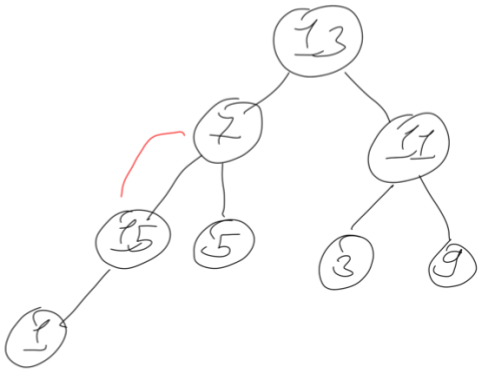
#18



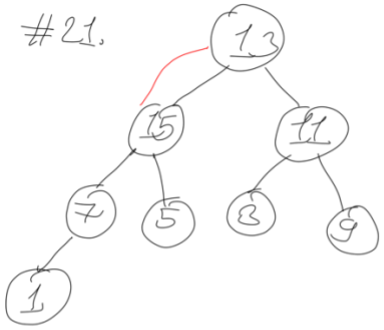
#19 insert(15)



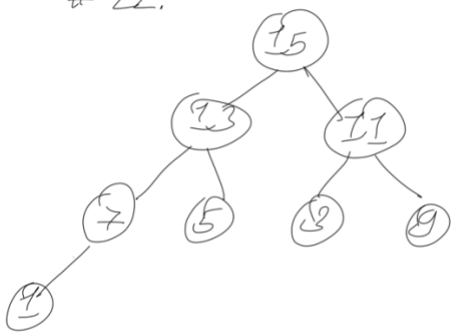
#20



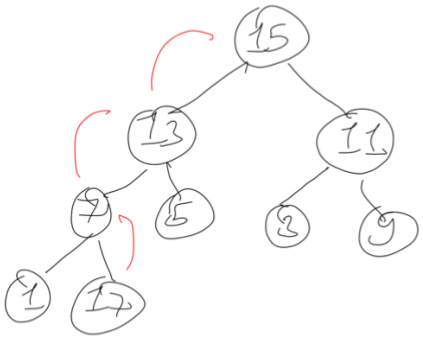
#21.



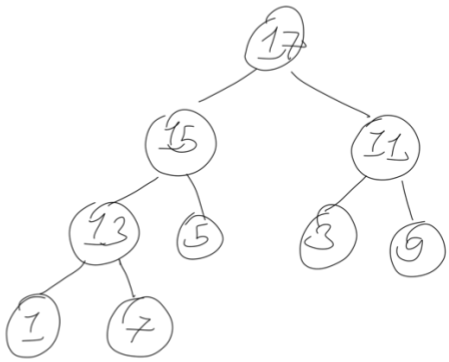
#22.



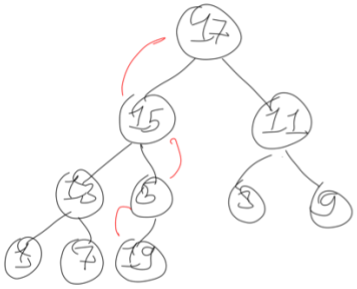
#23. insert(17)



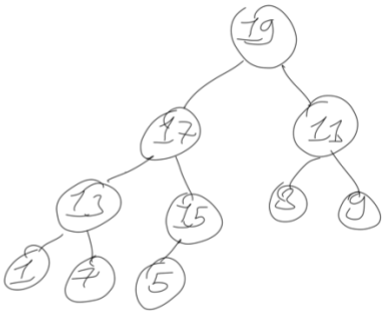
#24



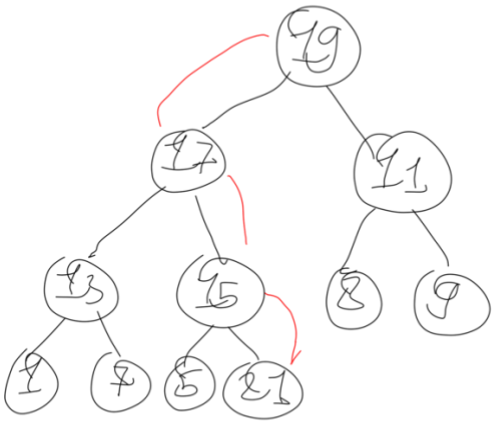
#25. insert(19)



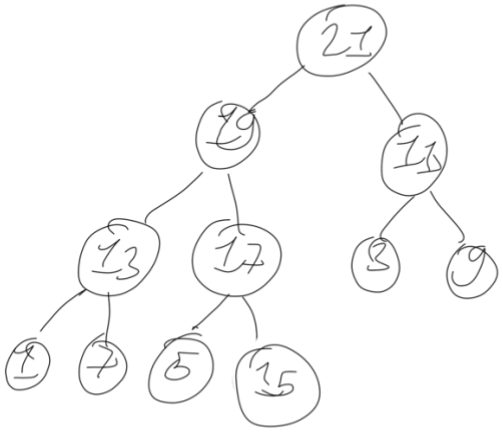
#26



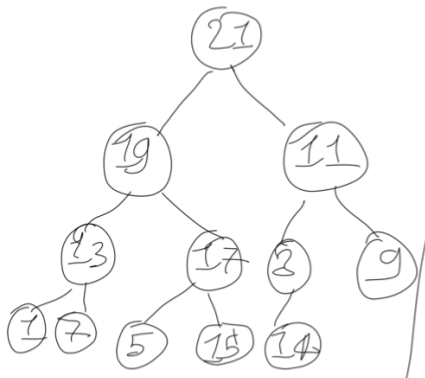
#27. insert(21)



#28.

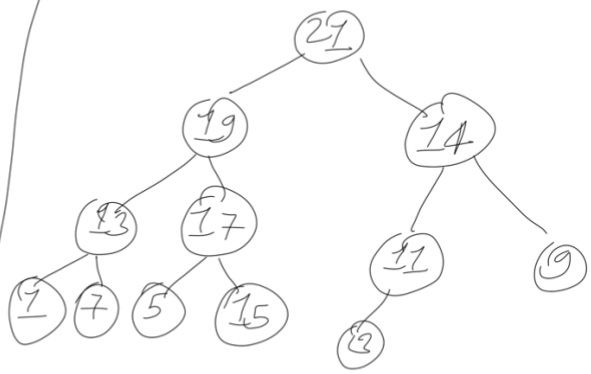


29 insert(14)



Perform upheap until balanced.

#30



And, this is the final heap.

Answer of 4

```
class Solution {

    static class Point {
        public int x;
        public int y;

        public Point(int x, int y) {
            this.x = x;
            this.y = y;
        }

        public double distanceFromOrigin() {
            return Math.sqrt(x * x + y * y);
        }

        public int[] toArray() {
            return new int[] { x, y };
        }
    }

    public int[][] kClosest(int[][] points, int k) {
        PriorityQueue<Point> pq = new PriorityQueue<>(
                (p1, p2) ->
Double.compare(p2.distanceFromOrigin(),
p1.distanceFromOrigin()));

        for (var point : points) {

            pq.add(new Point(point[0], point[1]));

            if (pq.size() > k) {
                pq.poll();
            }

        }

        int[][] result = new int[k][2];
        int index = 0;

        while (!pq.isEmpty()){
            result[index] = pq.poll().toArray();
            index++;
        }
```

```
        return result;
    }
}
```

Answer of 5

```
class Solution {

    static class CharacterFrequency{

        public char character;
        public int frequency;

        public CharacterFrequency(char character, int frequency
){
            this.character = character;
            this.frequency = frequency;
        }

        public int getFrequency(){
            return this.frequency;
        }
    }

    public String frequencySort(String s) {
        Map<Character, Integer> map = new HashMap<>();

        // count frequency
        for (var c: s.toCharArray()){
            map.put(c, map.getOrDefault(c, 0) + 1 );
        }

        PriorityQueue<CharacterFrequency> pq = new
PriorityQueue<>(

Comparator.comparing(CharacterFrequency::getFrequency).reversed(
)
        );

        for(Map.Entry<Character, Integer> entry:
map.entrySet()){
            pq.add( new CharacterFrequency(entry.getKey(),
entry.getValue()));
        }

        StringBuilder sb = new StringBuilder();
```

```
        while(!pq.isEmpty()){
           CharacterFrequency cf = pq.poll();

            for(int i = 0; i< cf.frequency; i++){
                sb.append(cf.character);
            }
        }

        return sb.toString();
    }
}
```