Heaps

Keys are assigned to its node (one key per node)

Prop. to be satisfied

1) Tree's structure: It should be a complete binary tree:-

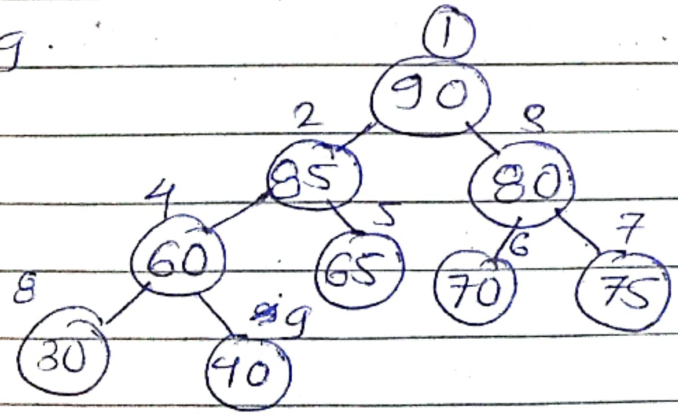
→ Completely filled on all levels ~~except the last one~~ with a possible exception that the last level is not completely filled

→ Lowest level filled from L to R

2) Heap order:
should follow ~~max~~ max-heap or min-heap property

Max-Heaps:
Key present at the root node must be greater than or equal to among keys present at all of its children.

E-g.



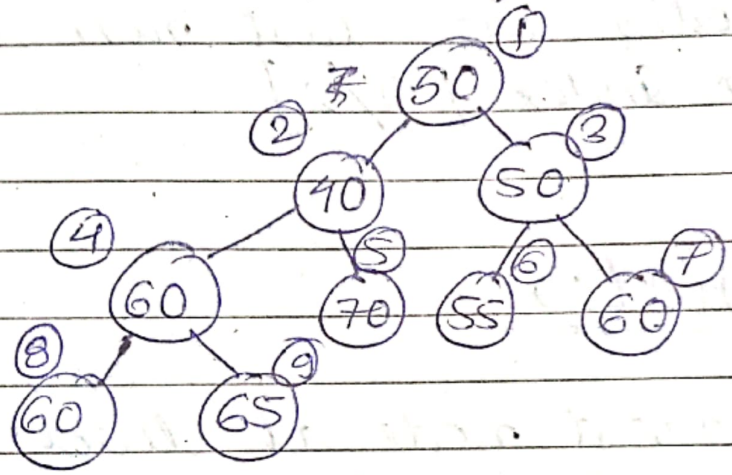
Array Representation:

90	85	80	60	65	70	75	30	40

~~Min Heap~~ : Min-Heap:

Key present at the root must be less than or equal to among keys present at all of its children.

E.g.



Representation:

30	40	50	60	70	55	60	60	65

Index Notes:

We begin indexing the root node at $i = 1$. (It makes the calculations easy)

If a node has index = i,
then index of it
Left child = $2i$
Right child = $2i+1$

If a node has index i:

i is Even: Parent = i/2

i is odd: Parent = (i-1)/2

At the end: for any index i

$$Parent = \lfloor i/2 \rfloor \rightarrow \text{G-I-F.}$$

⇒ For n nodes the last non-leaf node has index $\lfloor n/2 \rfloor$

Level notes

→ The leftmost node at level k (with level k=0 being the level of root) is node with index 2^k.

→ A node with index i is located at an level $\lfloor \lg i \rfloor$.

→ If there are n nodes in all, then max. level (the leaf level) equals $\lfloor \lg n \rfloor$.

→ The no. of levels betⁿ a node with index i & leaf layer, inclusive, is then $\lfloor \lg n \rfloor - \lfloor \lg i \rfloor + 1$

For a heap containing n nodes:

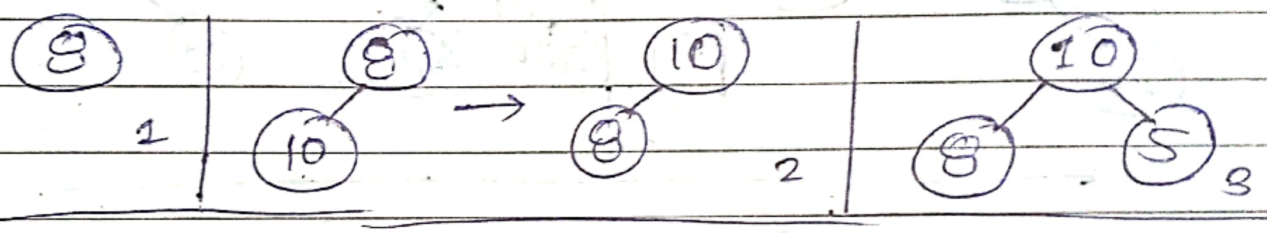
Leaf nodes lie from $\left(\frac{n}{2}+1\right)$ index to n^{th} index

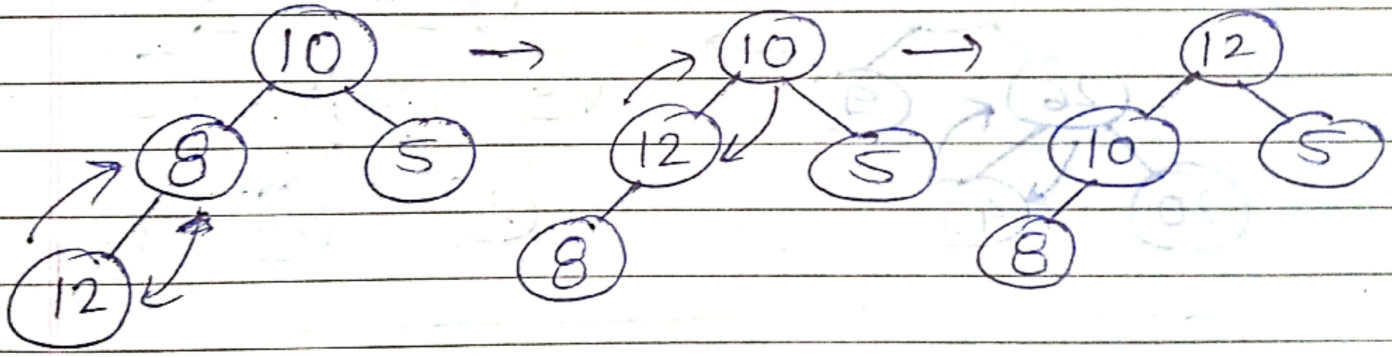
Top-Down approach or Upheapify

~~Array:~~

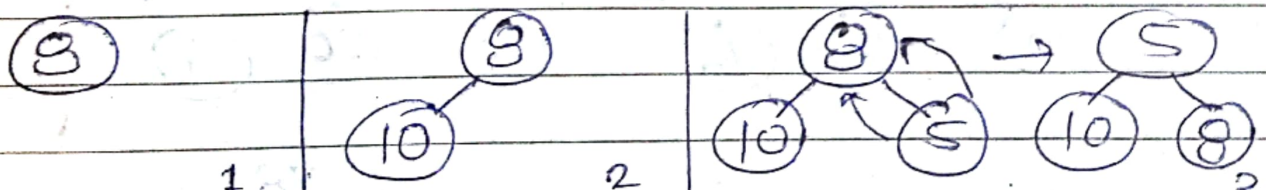
Elements: 8, 10, 5, 12,

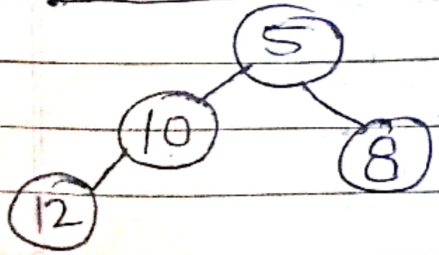
Max-Heap:





Min-Heap



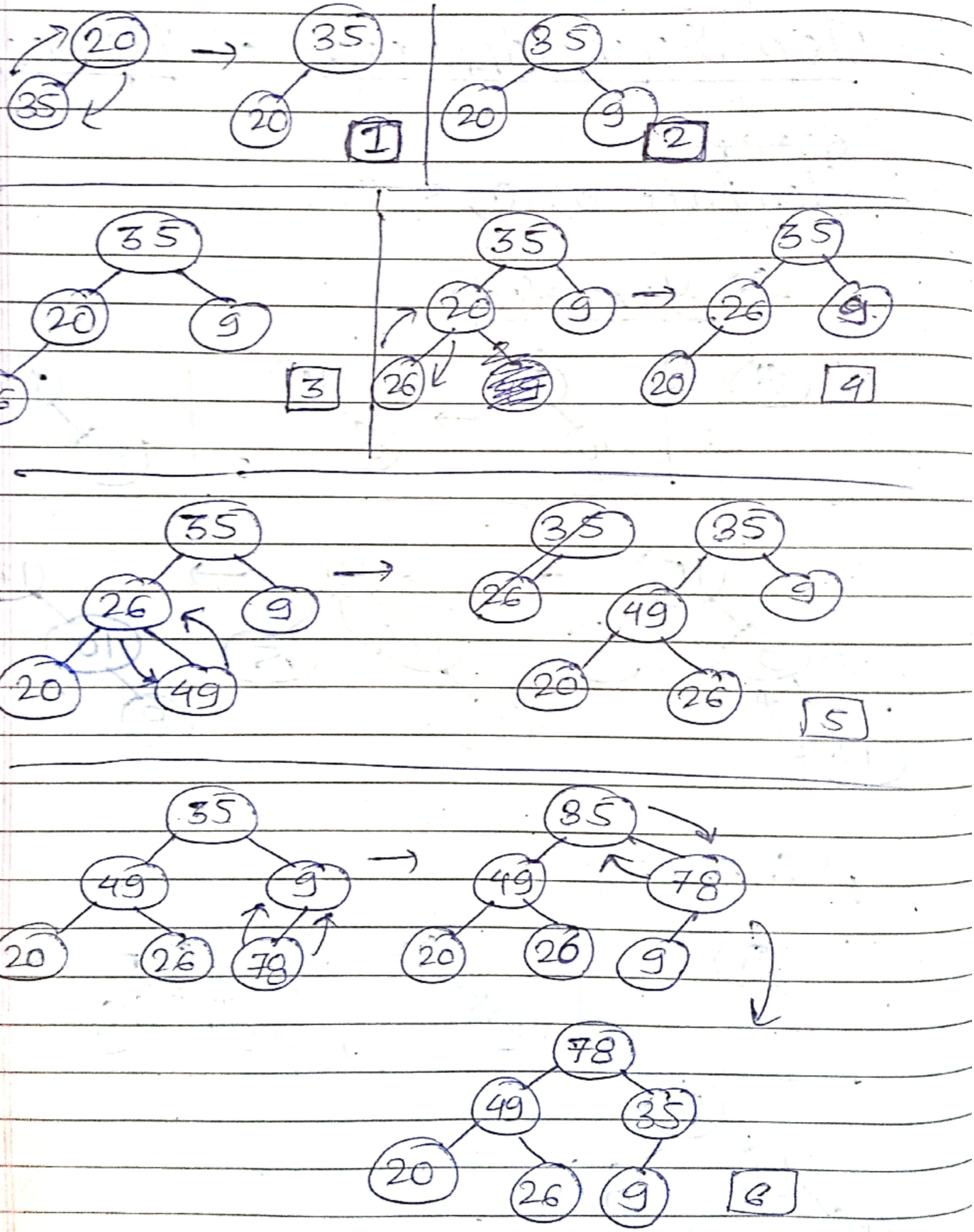


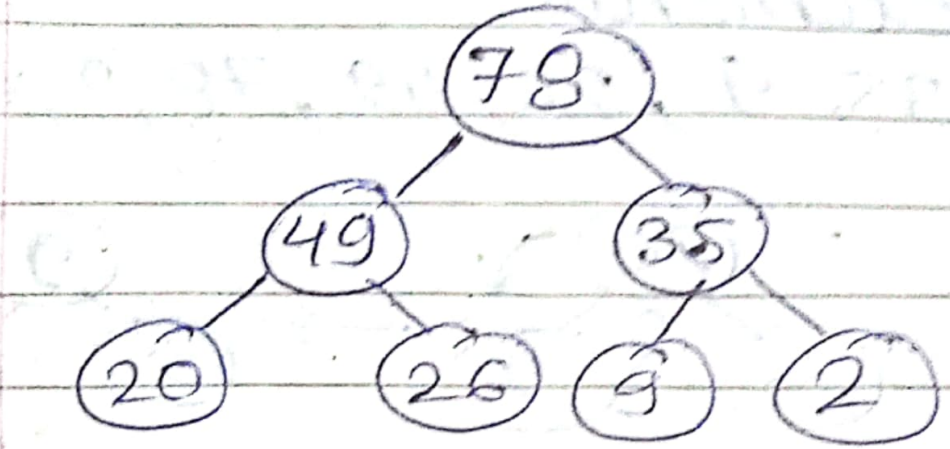
~~dsa~~

Elements:

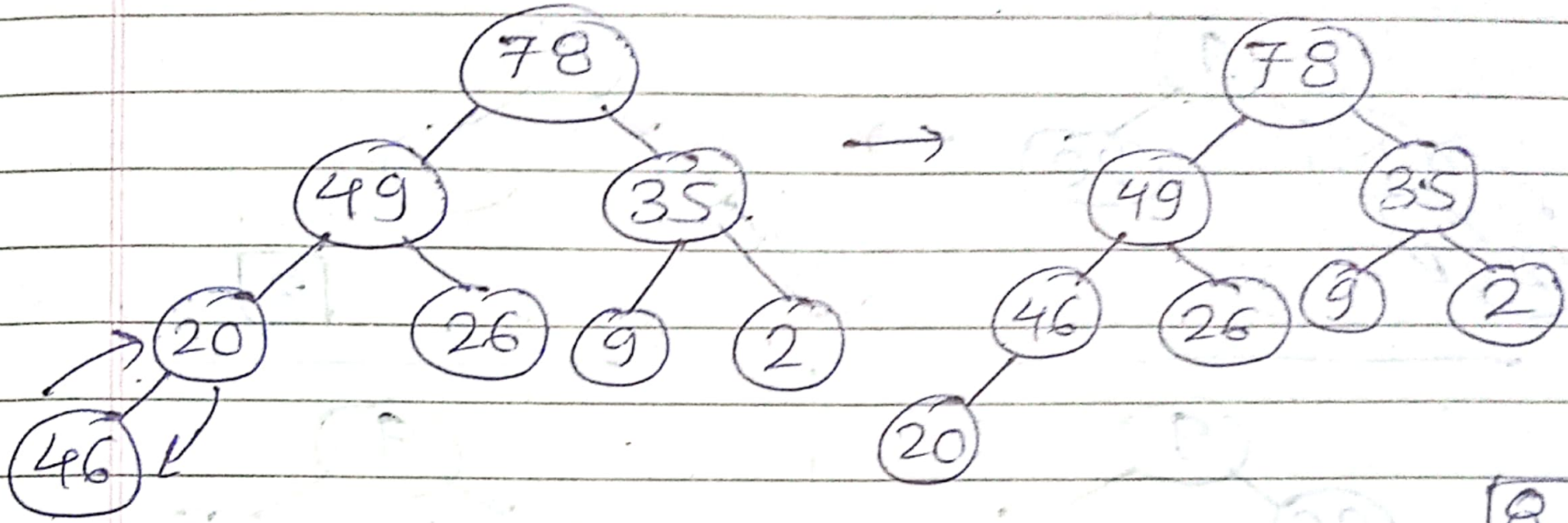
20, 35, 9, 26, 49, 78, 2, 46

Max-Heap:





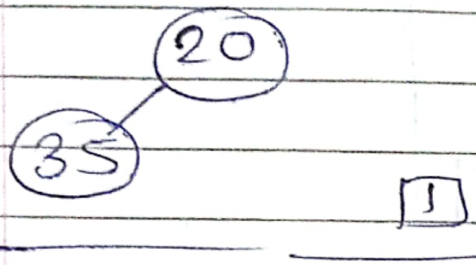
7



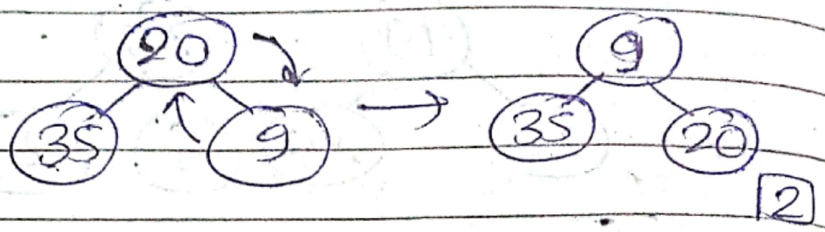
8

Min-Heap Elements

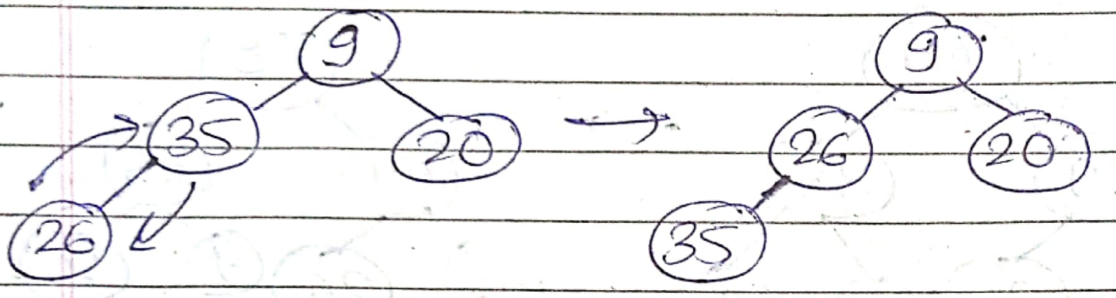
Elements: 20, 35, 9, 26, 49, 78, 2, 46



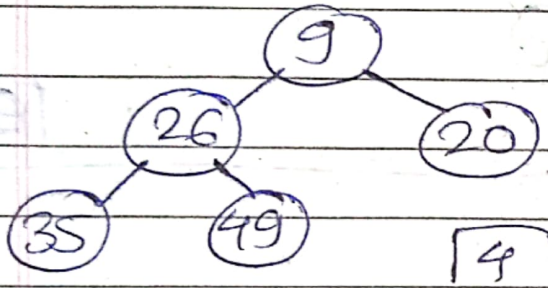
1



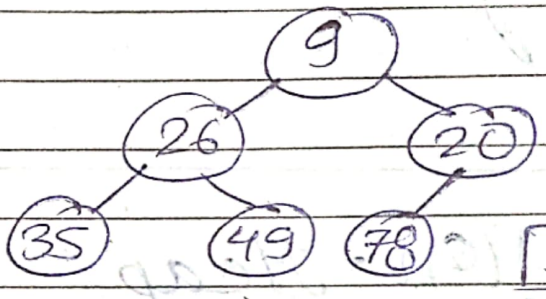
2



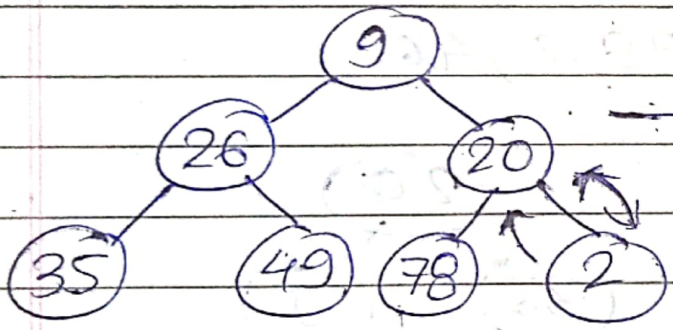
3

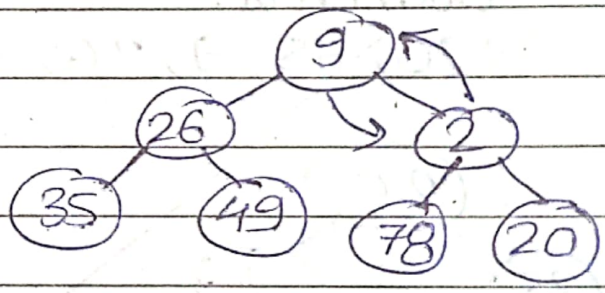


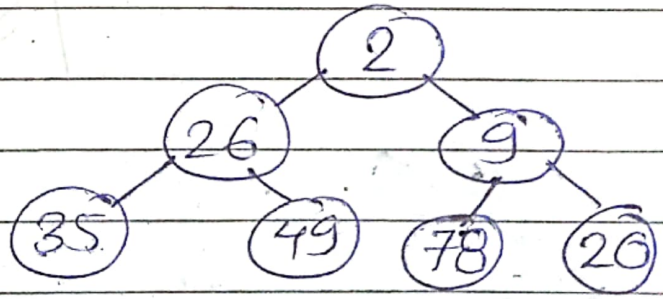
4



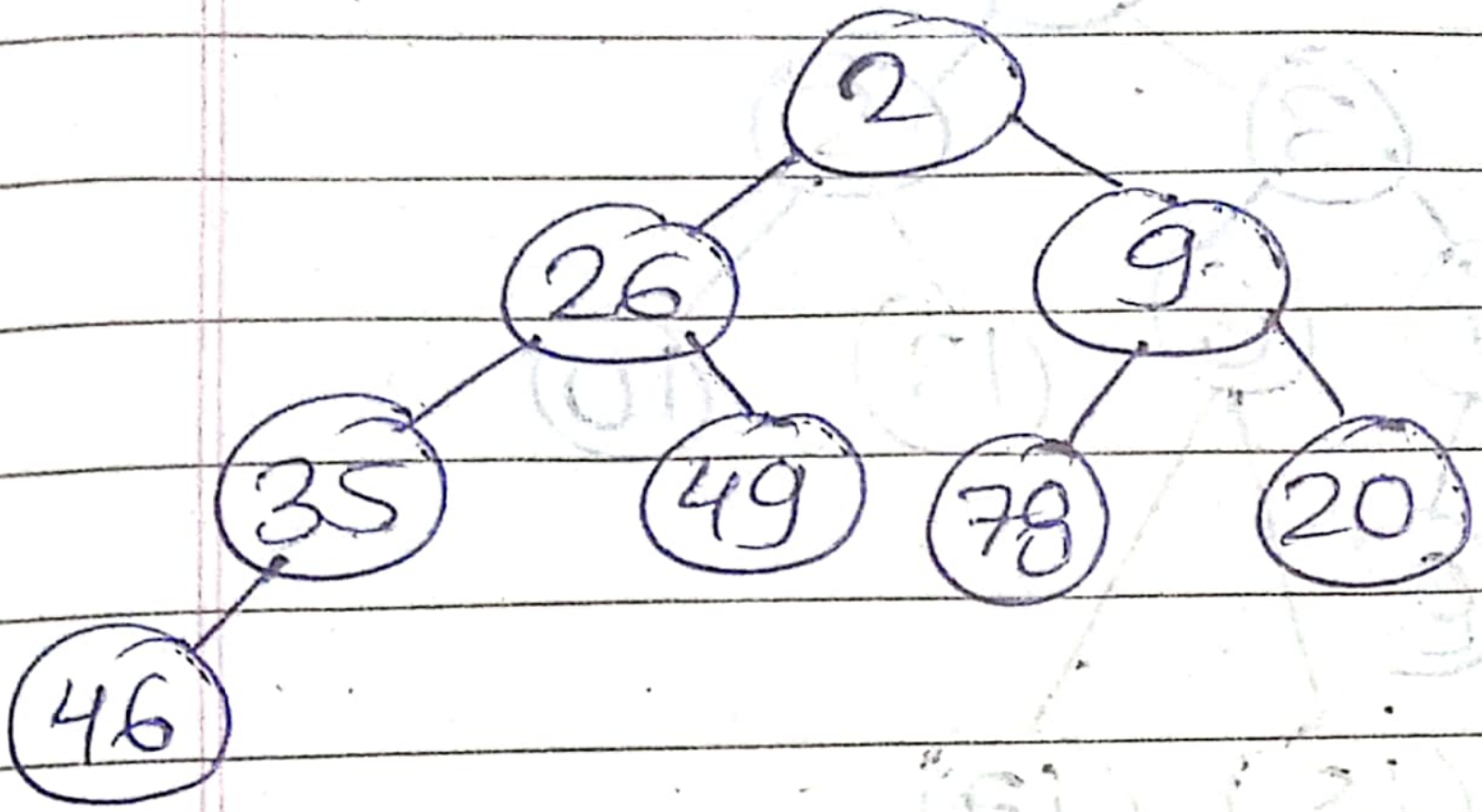
5







6

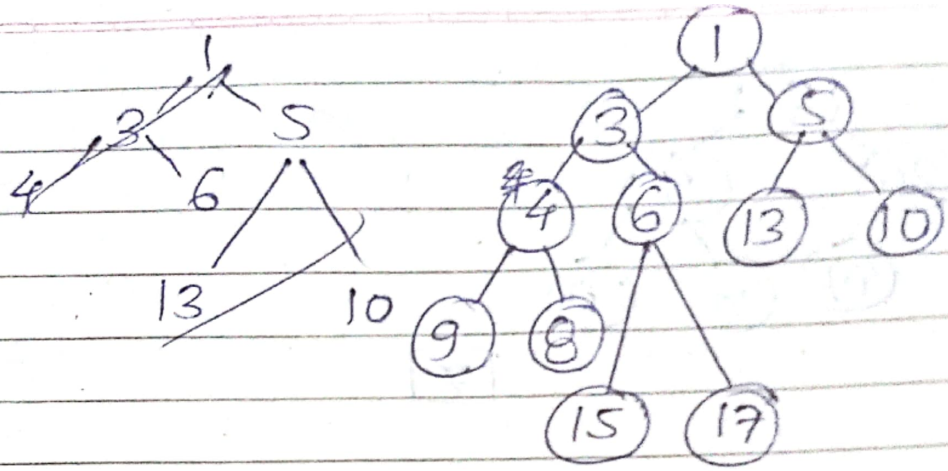


7

Heapify (Bottom-up approach)

Elements: 1, 3, 5, 4, 6, 13, 10, 9, 8, 15, 17

Con. Binary Tree:



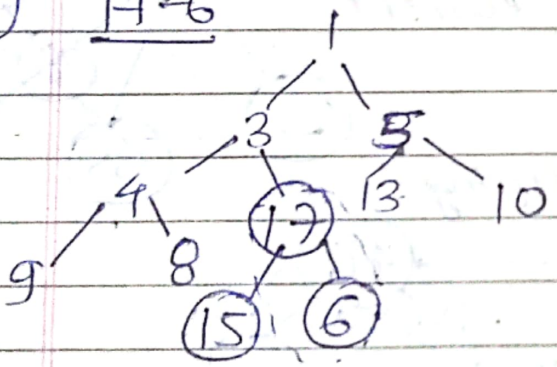
Max-Heap:

Total nodes = 11 ∴ | Index of last non-leaf node $\Rightarrow n/2 - 1$ = 4

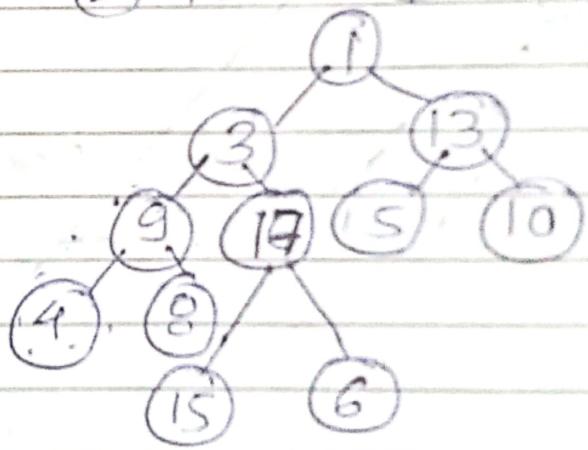
∴ Last non-leaf node = 6

∴ Heapify only: [1, 3, 5, 4, 6] in rev. ord.

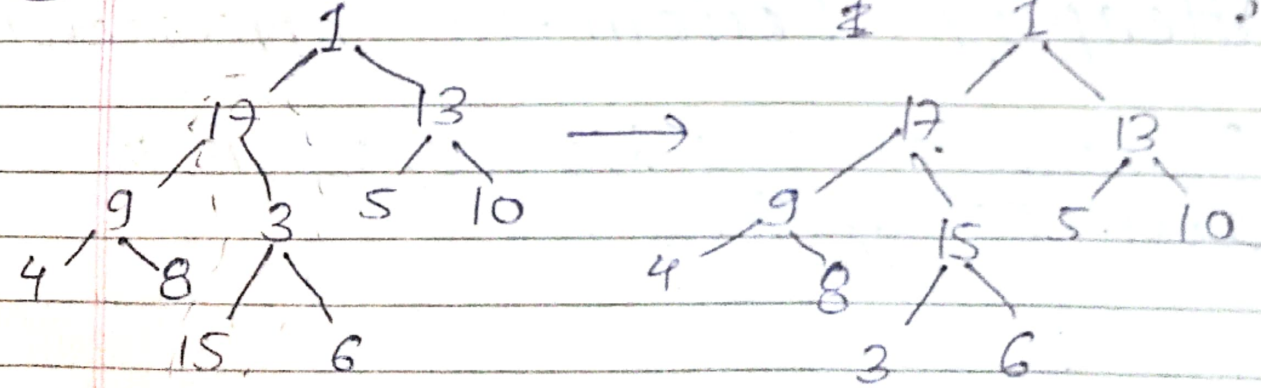
① H-6



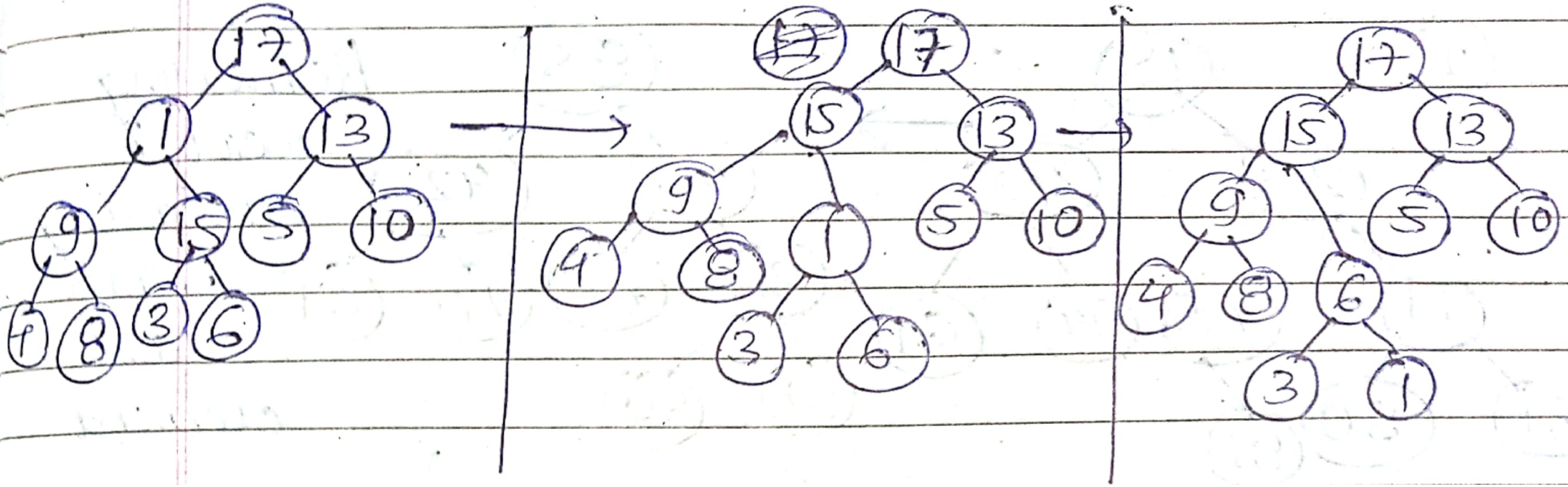
② H-4 & H-5



③ H-3 (1st ~~Heap~~ Swap 3 & 17 then 3 & 15)



(4) H-1 (Swap 1 & 17 and then 1 & 15 then 1 & 6



Algorithm Heapsort

(1) Build a min/max heap from data

(2) Max./Min. element is stored at the root of heap.
   a. Replace it with last item of heap
   b. Reduce size of array by one
   c. Heapify the root

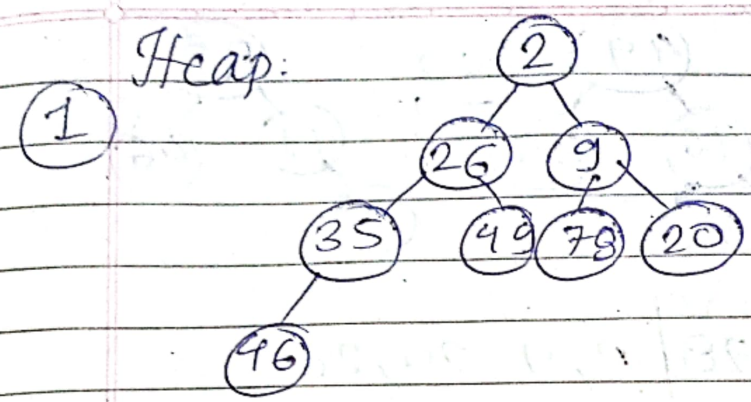
(3) Repeat 2 until size of heap is greater than 1

~~Elements~~: Heap: array:

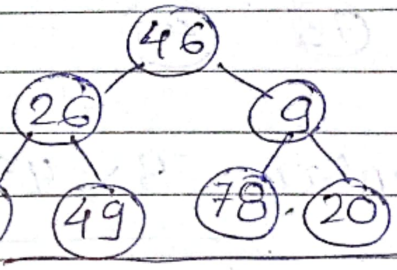
2, 26, 9, 35, 49, 78, 20, 46

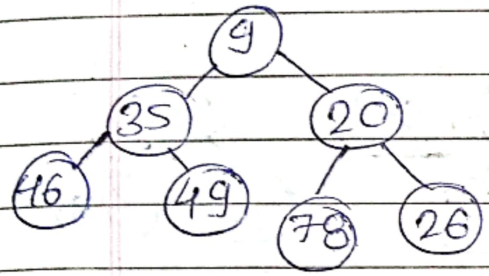
Adwait Purao
2021300101

Heap:



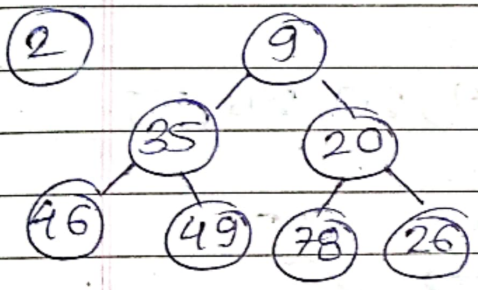
S-1 (Replace root with last element & dec. size by 1)



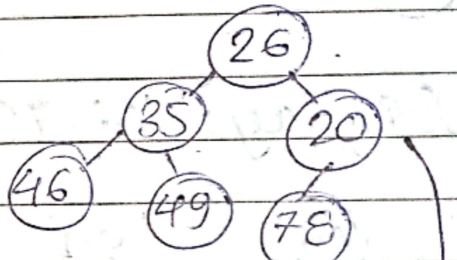


S-2 Heapify

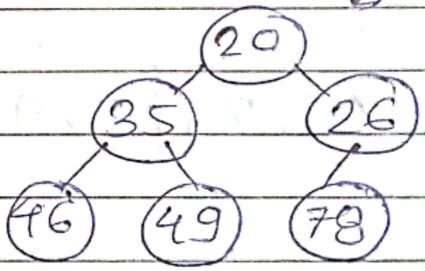
Display:
9, 35, 20, 46, 49, 78, 26 | 2



S-1

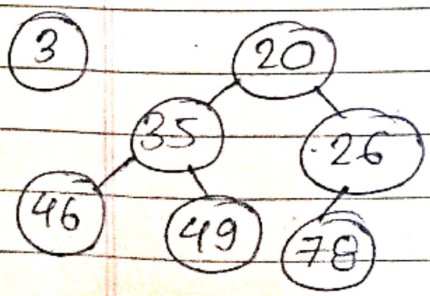


S-2

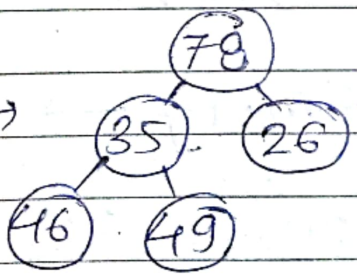


Display:

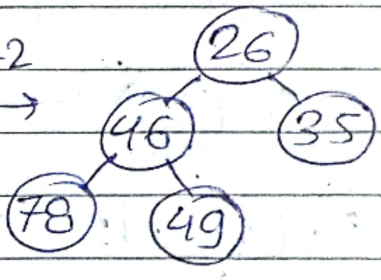
20, 35, 26, 46, 49, 78 | 2, 9



S-1

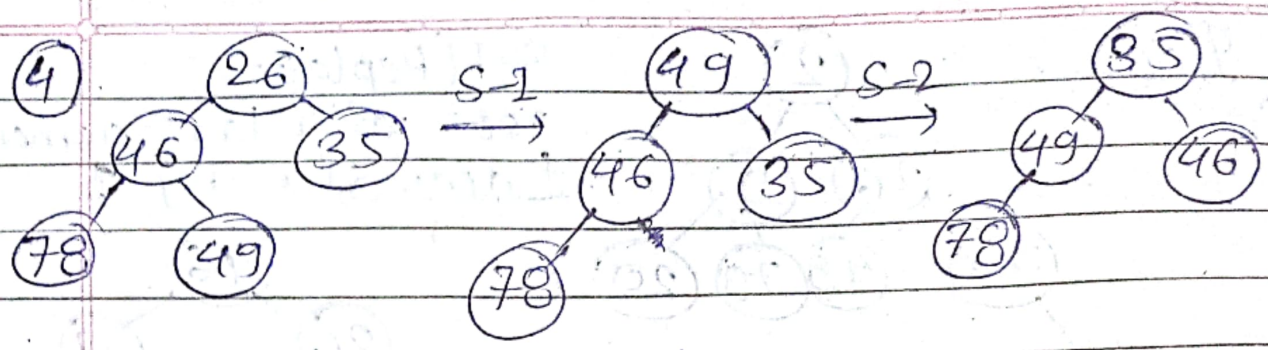


S-2

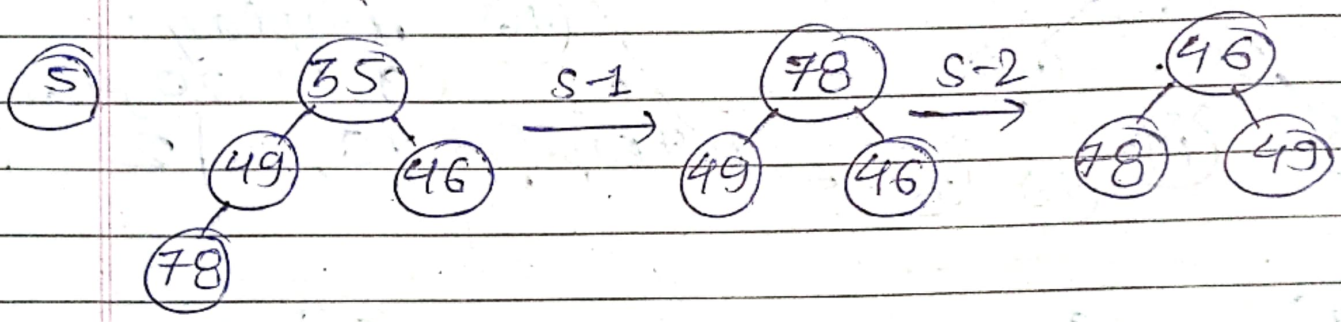


Display: 26, 46, 35, 78, 49 | 2, 9, 20

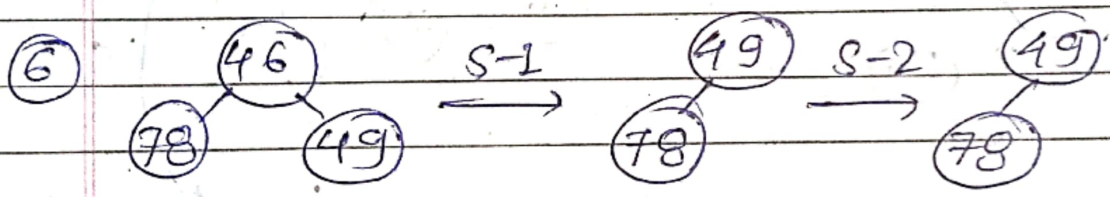
Adwait Purao
2021300101



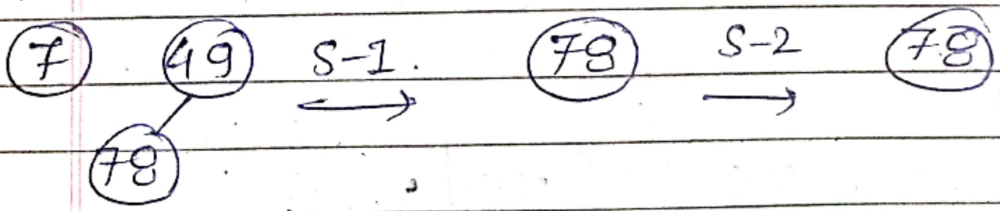
Display: 35, 49, 46, 78 | 2, 9, 20, 26



Display: 46, 78, 49 | 2, 9, 20, 26, 35



Display: 49, 78 | 2, 9, 20, 26, 35, 46

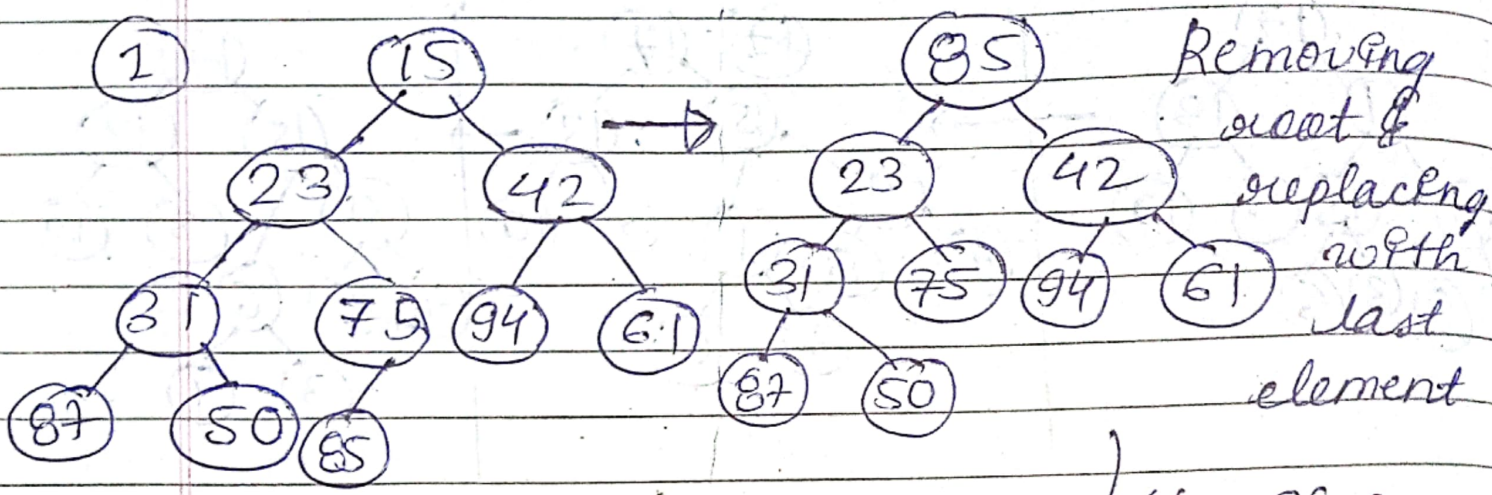


Display: 78 | 2, 9, 20, 26, 35, 46, 49

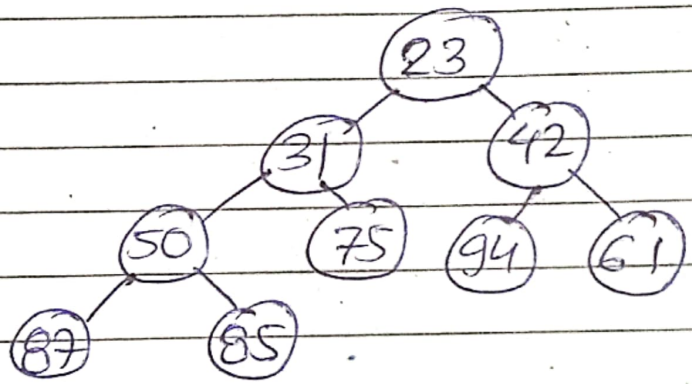
8 78 → Empty → Empty

Display: | 2, 9, 20, 26, 35, 46, 49, 78

Heap sort

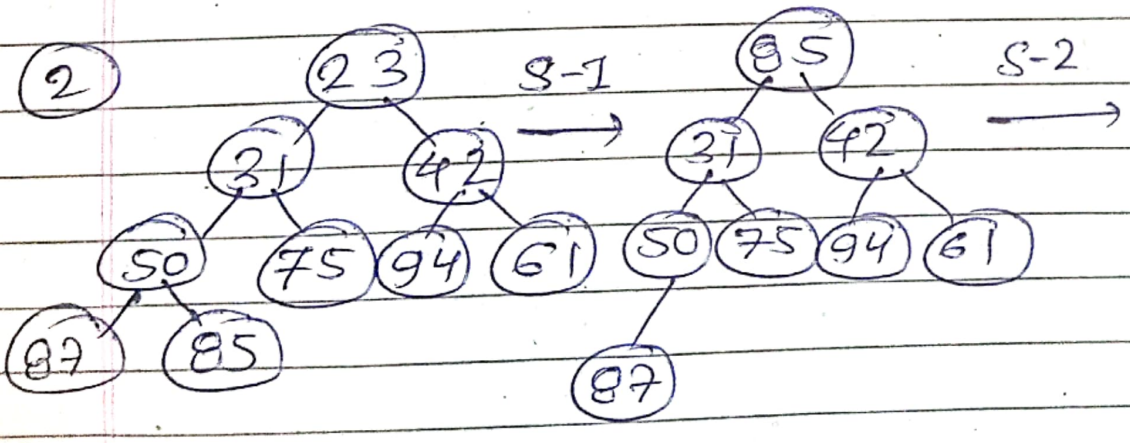


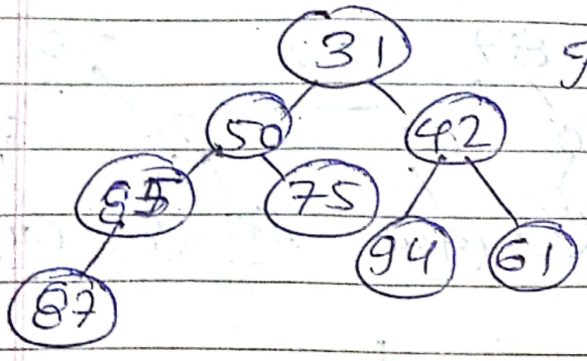
Heapifying rem. tree



~~15~~, 23, 42, 31, 75, 94, 61, 87, 50, 85

23, 31, 42, 50, 75, 94, 61, 87, 85 | 15



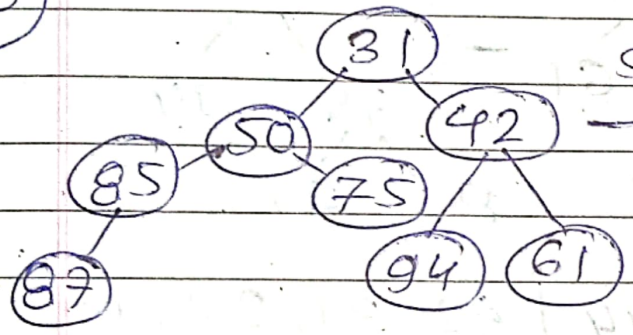


Heapifying

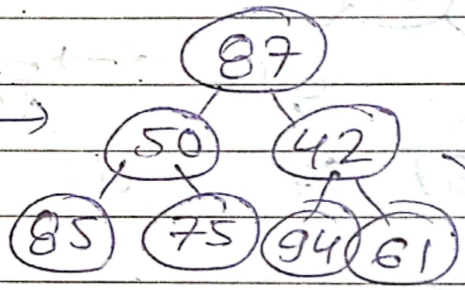
23, 31, 42, 50, 75, 94, 61, 87, 85 | 15

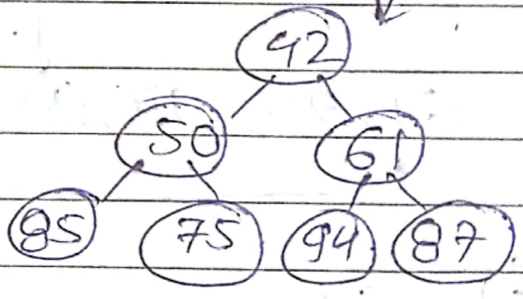
31, 50, 42, 85, 75, 94, 61, 87 | 23

5



S-1

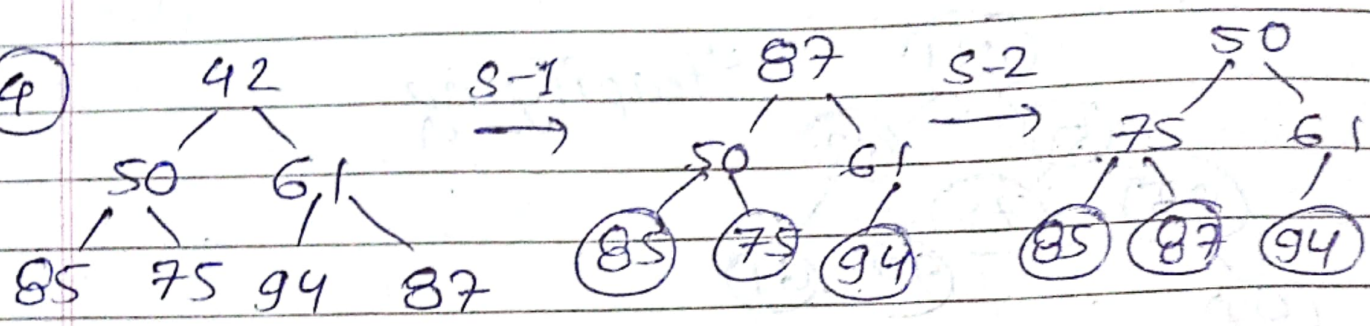




~~31~~, 50, 42, 85, 75, 94, 61, 87 | 23, 15

42, 50, 61, 85, 75, 94, 61, 87 | 31, 23, 15

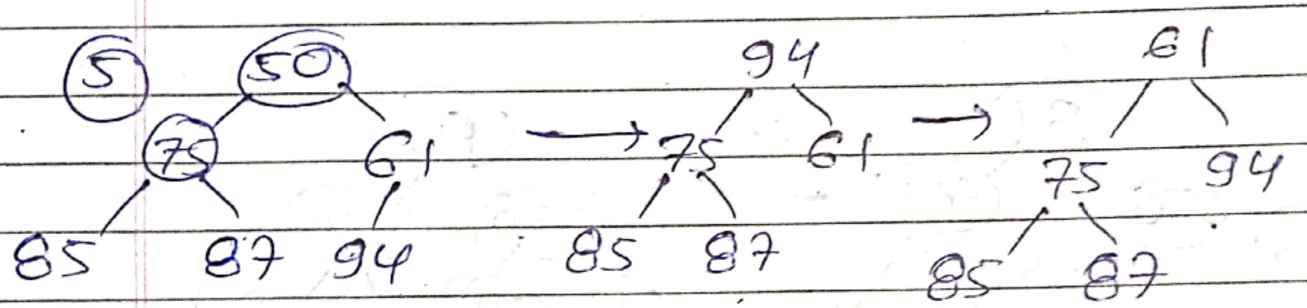
(4)



42, 50, 61, 85, 75, 94, 87 | 31, 23, 15

50, 75, 61, 85, 87, 94 | 42, 31, 23, 15

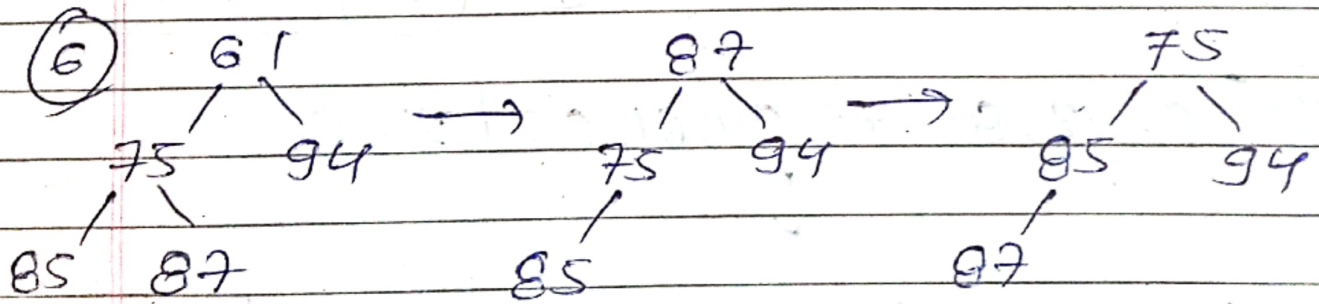
(5)



50, 75, 61, 85, 87, 94 | 42, 31, 23, 15

61, 75, 94, 85, 87 | 50, 42, 31, 23, 15

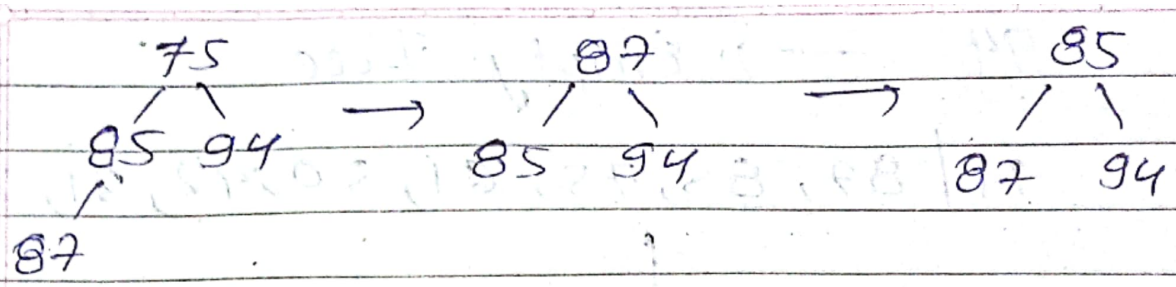
(6)



61, 75, 94, 85, 87 | 50, 42, 31, 23, 15

75, 85, 94, 87 | 61, 50, 42, 31, 23, 15

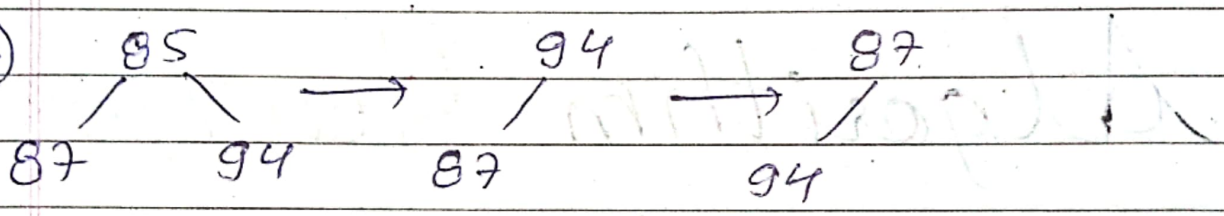
(7)



~~75~~, 85, 94, 87 | 61, 50, 42, 31, 23, 15

85, 87, 94 | 75, 61, 50, 42, 31, 23, 15

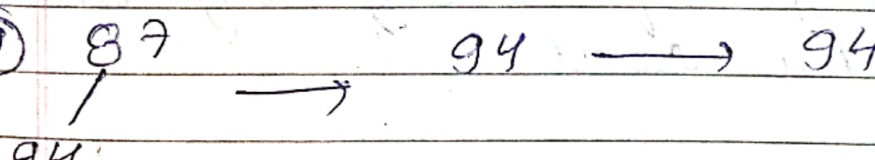
(8)



85 87 94 | 75, 61, 50, 42, 31, 23, 15

87, 94 | 85, 75, 61, 50, 42, 31, 23, 15

(9)

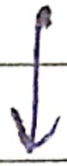


87 94 | 85, 75, 61, 50, 42, 31, 23, 15,

94 | 87, 85, 75, 61, 50, 42, 31, 23, 15

(10) 94 ⟶ Empty Tree

94 | 87, 85, 75, 61, 50, 42, 31, 23, 15



94, 87, 85, 75, 61, 50, 42, 31, 23, 15

∴ Final Heap sorted array

94, 87, 85, 75, 61, 50, 42, 31, 23, 15

Heapify (Top-Down approach)

```
struct Heap{
  int * ar;
  int size; // Maintaing the total size.
  int usize; // Maintaing the index
    };

void swap(int * a, int * b){
 int temp = *a;
  *a = *b;
  *b = temp;
   }

int isFull(struct Heap *h)
{
  if(h->usize==h->size-1)
  {
     return 1;
   }
 return 0;
};

void Display(struct Heap * h){
    pf("\n Display of Heap:\n");

  for(int i=1;i<=h->usize;i++)
  {
    pf("%d ", h->ar[i]);
  }

};
```

```
void TDHeapify(struct Heap * h)
{
    if(h->usize == 1)
    {
        return;
    }
// If size of heap is already 1, there
// is no need to Heapify the tree as
// it's already Heapified

    else {
        for(int i=2; i<=h->usize; i++)
        {
        int curr_ind=i;
// If parent is greater than current
// index then min heap prop. gets
// violated

        while((curr_ind>1) &&
(h->arr[curr_ind/2] > h->arr[curr_ind])){

swap(&(h->arr[curr_ind]), &(h->arr[curr_ind/2]));

curr_ind /= 2;

        }

    }

}
```

```
void Insertion(struct Heap *h, int data
{
    if(isFull(h)==1)
    {
    pf("Heap is full");
    }
    else {
    h->usize++;
    h->arr[h->usize] = data;
    TDHeapify(h);
    }
    Display(h);

}

int HeapSort(struct Heap *h, int sorted[]
{
    int count=0;
    while(h->usize!=1)
    {
    count++;
    sorted[count]=h->arr[1];
    swap(&(h->arr[h->usize]),&(h->arr[1]));
// Replace last element with first of
Heap
    h->usize--;
    TDHeapify(h);
    Display(h);
    }
count++;
sorted[count]=h->arr[1];
return count;
}
```

```
void heapify(struct Heap *h, int i){
    int n = h->usize;
    int largest = i;
    int left = 2*i+1;
    int right = 2*i+2;
    if(left<n && h->arr[left]>h->arr[largest])
        largest = left;
    if(right<n && h->arr[right]>h->arr[largest])
        largest = right;
    if(largest!=i){
        swap(&(h->arr[i]), &(h->arr[largest]));
        heapify(h, largest);
    }
}
```