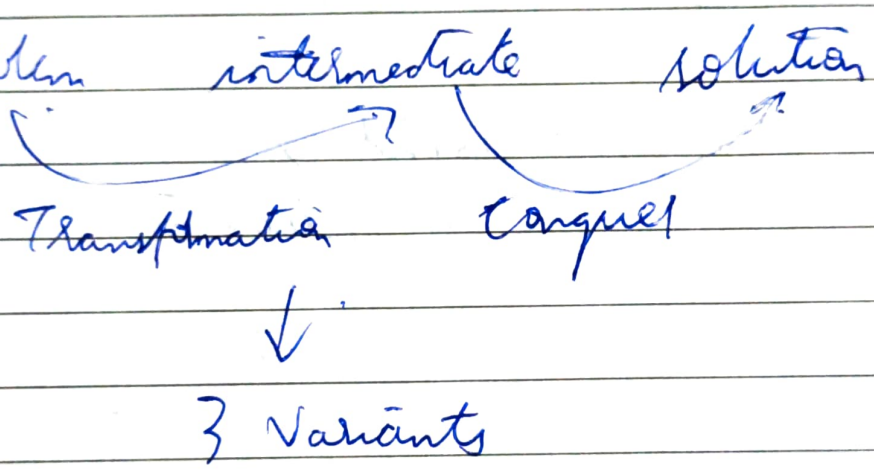
3D) Transform & conquer Instance Simplification

A problem that can be solved by transforming a given problem instance into an intermediate instance & then determining the solution.



↓

3 Variants

1) Instance Simplification → Transforming the problem into a simpler or more convenient way. Eg:- Presorting, AVL

2) Representation change → Transforming the problem instance into a different representation of the same problem
Eg:- 2-3 tree, heapsort

3) Problem reduction → Transformation to an instance of a different problem for which the solution is already available

Eg:- $LCM(a, b) = \frac{a \cdot b}{gcd}$

Presorting (Instance Simplification)

Presorting is not a direct problem solving technique or method, it is just an intermediate stage to solve the problem

Problem:- To find element uniqueness in a set of elements

Initial: 2 4 7 5 6 4 10
Final: 2 4 4 5 6 7 10

Brute Force method
↓
n^2

Assume that ~~sort~~ mergesort is $n \log n$
+
(compare the adjacent element $(n-1)$)

Total Time $= n \log n + n \rightarrow n \log n$

__/__/____

Algorithm :- Presort Element uniqueness (A[0..n-1])

// Checks Element uniqueness based on presorting
// Input : An array A[0 ... n-1]
// Output : Returns TRUE if all elements are distinct else FALSE

Sort A
for i ← 0 to n-2 do
    if A[i] == A[i+1]
        return FALSE
return TRUE

n-1

To find 'mode' among n elements using presorting

Mode is a statistical quantity which is nothing but an element in the set which occurs most no. of times

Eg:- 2 5 4 7 5 4 1 10 5 4 4 12
Mode = 4

Brute force is used to find the soln (n^2)

1 2 4 4 4 4 5 5 5 7 10 12
Mode is 4

Sort all elements - $n \log n$

Compare adjacent element - n

Algorithm mode ($A[0, \ldots n-1]$)

// Computes mode of an array using presorting

// Input: An array A of n elements

// Output: The array's mode

```
Sort A
i ← 0
modefrequency ← 0
while i ≤ n-1 do
    runlength ← 1
    runvalue ← A[i]
    while (i + runlength ≤ n-1 & A[i + runlength]
    = runvalue)
        runlength ← runlength + 1
    if runlength > modefrequency
        modefrequency ← runlength
        modevalue ← runvalue
    i ← i + runlength
return modevalue
```

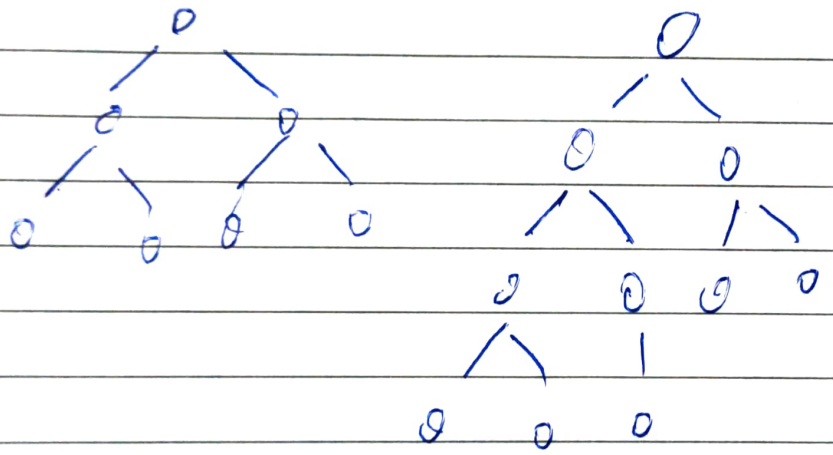
32) Heap Sort

__/__/____

Heap

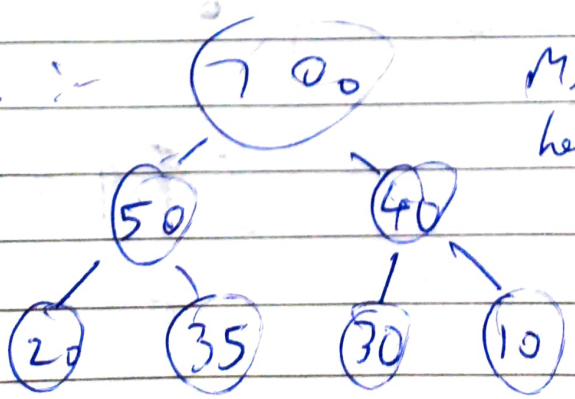
→ Heap is a binary tree with 2 important properties

i) Structural property - it should be an almost complete binary tree i.e all from last level, all other levels should have 2^i node, and last level should be left filled

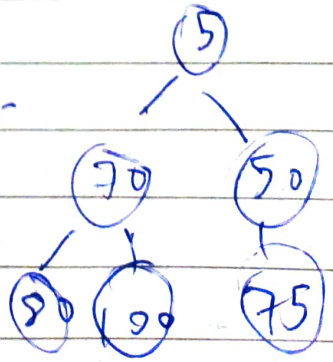


ii) Parent - dominant property → Parent should be more dominant when compared to its children (Max heap / Min heap)

Max heap :-



Min heap :-



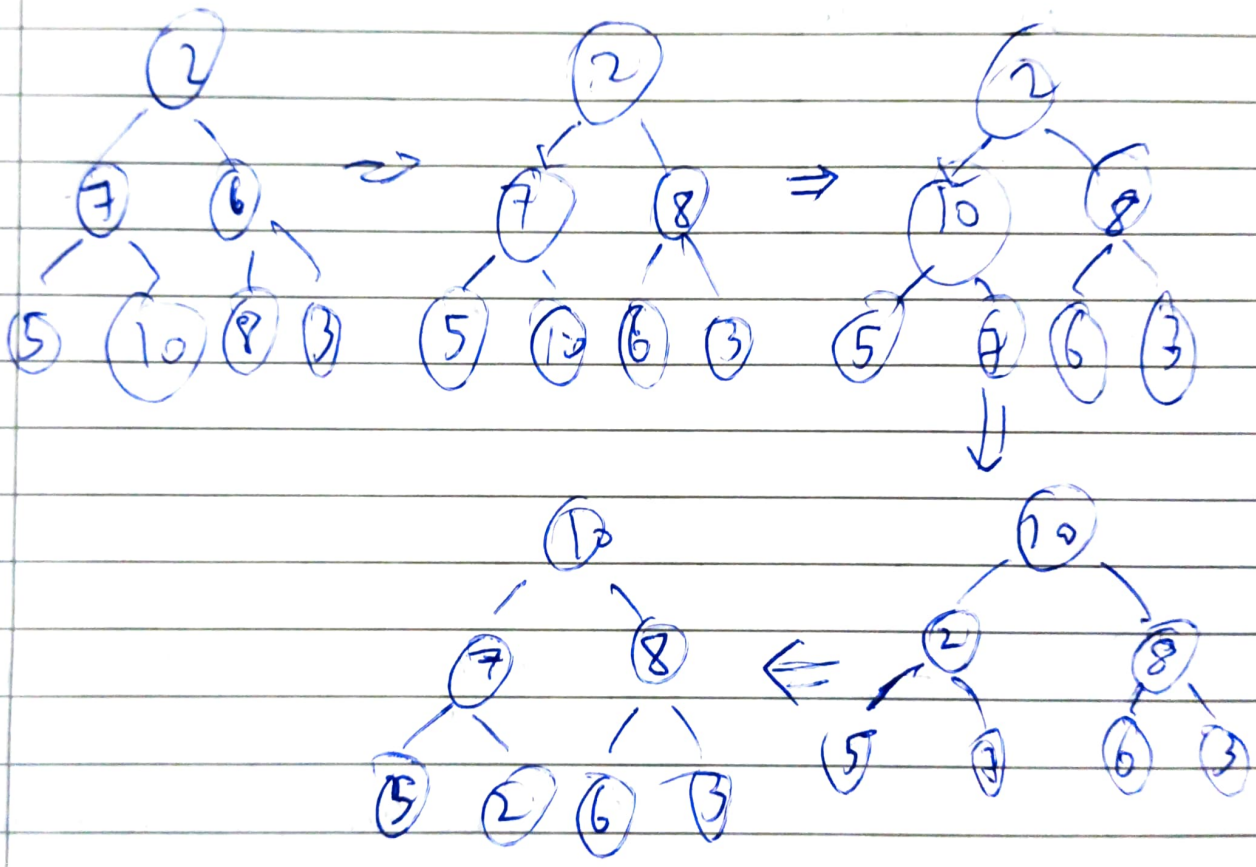
Methods of constructing heap

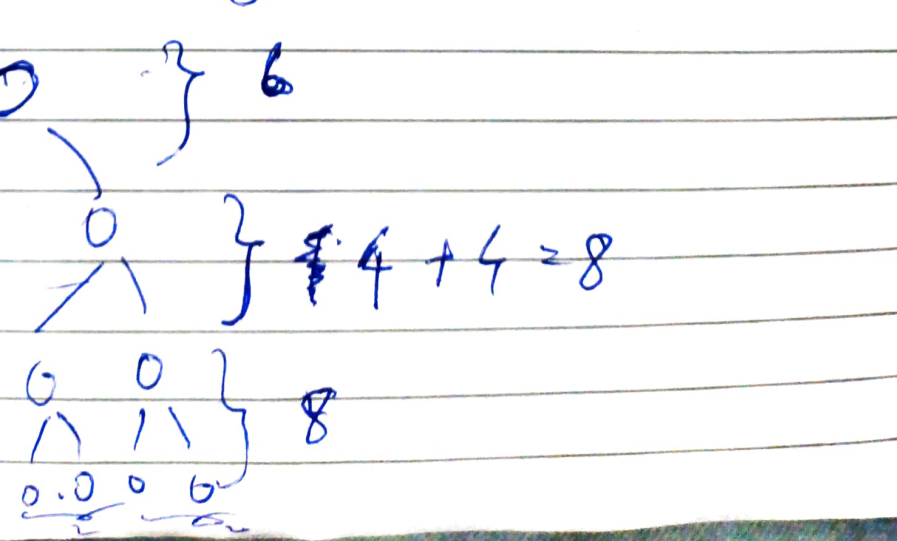
1) Bottom-up approach
2) Top-down approach

objective is to construct maxheap

Bottom-up approach

2 7 6 5 10 8 3





No. of

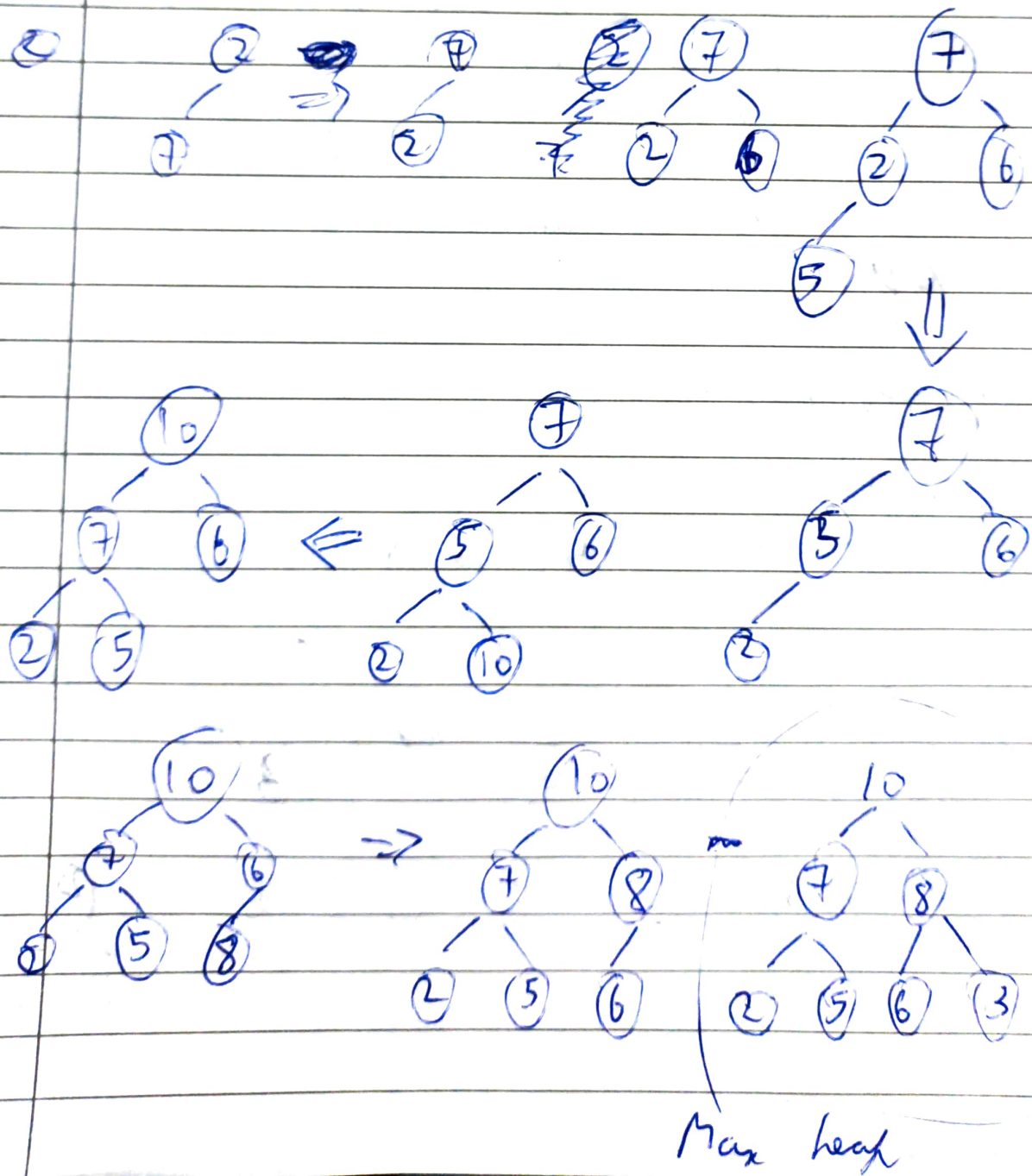
i Total nodes eq = 15

No. of comparisons = 2n - 2 - log(n) ...

∴ O(n)

Top-down approach

2 7 6 5 10 8 3



__/__/____

Time complexity → (log n)

Heapsort

Procedure to perform heapsort :

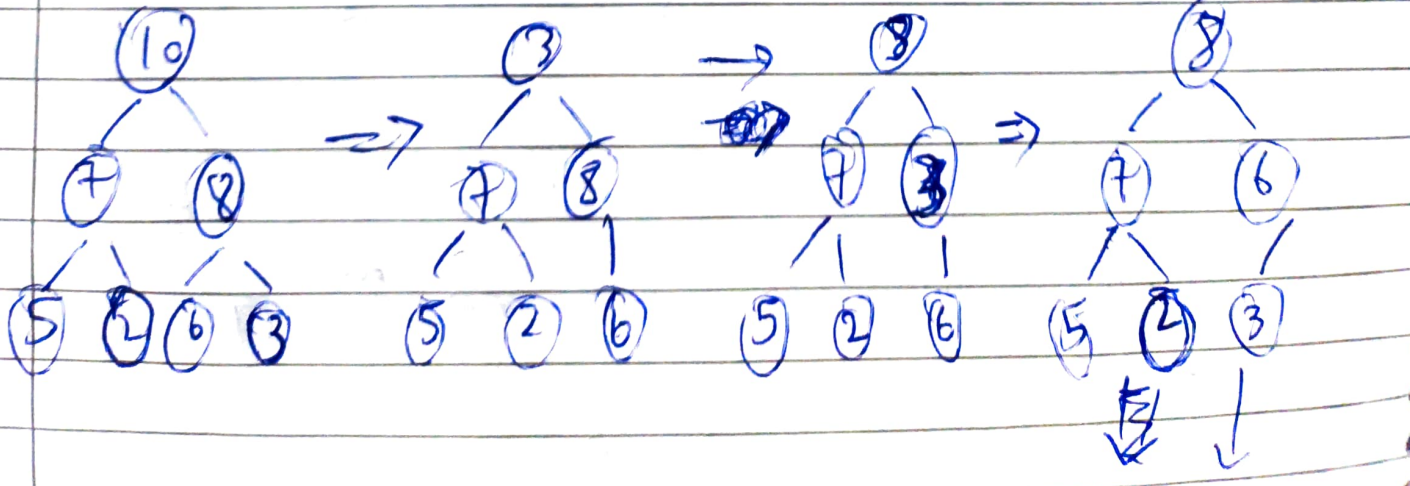
1) Construct a max-heap for the given set of elements (n)

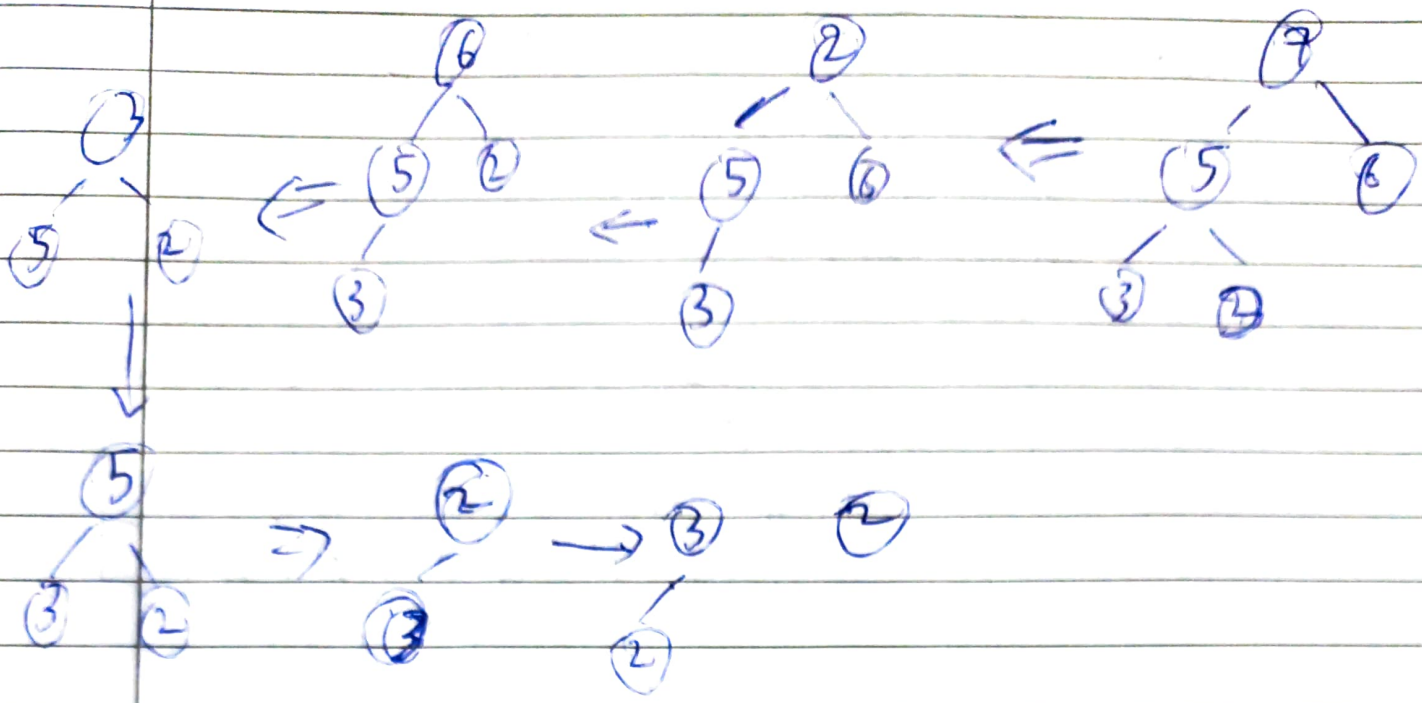
2) Exchange the first and the last elements, reduce the size of the heap by 1 & construct heap for remaining (n-1) elements

3) Repeat step 2 until only one element is leftout

Apply heapsort for the elements

index →	1	2	3	4	5	6	7	Result;
	2	7	6	5	10	8	3	2 3 5 6 7 8 10





Efficiency of heapsort = Construction + Exchanging
of heap 1st &
O(n) last
element
(n-1) log n

n log n

Heapsort is inplace sorting algorithm

Algorithm Heapify (A[1 ... n])

// Constructs a maxheap using bottom-up approach
// Input: An array A of n elements
// Output: Heap, represented using an array

for $i \leftarrow \left[\frac{n}{2}\right]$ down to 1, do

```
k → i
v ← H[k]
heap ← FALSE
while not heap and 2k ≤ n do
    j ← 2k
    if j < n
        if H[j] < A[j+1]
            j ← j+1
    if v ≥ H[j]
        heap ← TRUE
    else
        H[k] ← H[j]
        k ← j
H[k] ← v
```

33) C Program to implement heap sort

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 1000
int main()
{
    int i, n, a[MAX];
    printf("\n Read value for n: ");
    scanf("%d", &n);
```

```
printf("\n Read array elements \n");
for (i=1; i<=n; i++)
    scanf("%d", &a[i]);
heapify(a,n);
heapsort(a,n);
printf("\n sorted elements are \n");
for (i=1; i<=n; i++)
    printf("%d \n", a[i]);
return 0;
}

void heapify(int a[max], int n)
{
    int i, j, k, v, flag;
    for (i=n/2; i>=1; i--)
    {
        k=i;
        v=a[k];
        flag=0;
        while(!flag && 2*k<=n)
        {
            j=2*k;
            if(j<n)
            {
                if(a[j]<a[j+1])
                    j=j+1;
            }
            if(v>=a[j])
                flag=1;
```

```
        else
        {
            a[k] = a[j]
                k = j;
        }
    }
    a[k] = v;
  }
}

void heapsort (int a[MAX], int n)
{
    int i, temp;
    for (i = n; i > 1; i--)
    {
        temp = a[1];
        a[1] = a[i];
        a[i] = temp;
        heapify(a, i-1);
    }
}
```