GRADING:

- → No Quiz 1
- → Quiz 2 : 10
- → Mid : 15
- → End : 20
- → Assignment : 10
  (5 Assignments)
- → Labs: Grading starts in 2nd Half (20)
  (Best 5 of 10)
- → Mid Lab : 10
- → End Lab : 15
- → Class Tests

⁂ Sorting Methods :

(1) Insertion Sort :

Pseudo Code :

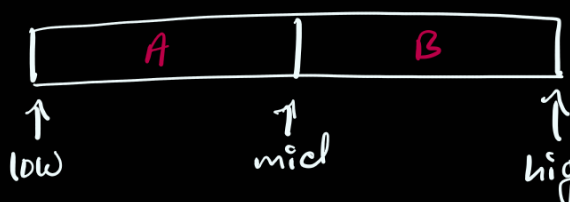
```
for (i : 1 -> n)
    int j = i-1
    int key = arr[i]
    while ( j>=0 && arr[j] > key )
        arr[j+1] = arr[j]
        j--
    arr[j+1] = key
```

Time Complexity : $O(n^2)$

NOTE: Binary Search : search in a sorted array.

For eg, if we want to search for a "target" in a sorted array;
take first element as low, last element as high

Take mid = low + (high - low)/2 ;



low to high is called a search space.

If target = mid then we found the target.

else if, target > mid, we know that target won't be in search space B, thus we cut down the seach space to A by, high = mid - 1

else, [target < mid]

we cut down the search space to B by, low = mid + 1

Time Complexity: $O(\log_2^n)$

Note: Inversions => (a_i, a_j) such that $i < j$ but $a_i > a_j$

> Number of Inversion pairs in an array = Number of shifts happening in Inversion Sort.

think about this!!!

---

(2) BUBBLE SORT:

→ Traversing the array, & swapping continuously two consecutive elements as long as the former greater than latter.

→ Once we have traversed once, we start again from 1st element until the 2nd last element.

Time Complexity: $O(n^2)$

Best case $O(n)$ (already sorted).

Pseud Code:

```
for(i: 0 -> i<n-1)
    flag = 0
    for(j: 0 -> j<n-i-1)
        if(arr[j+1] < arr[j])
            swap(arr[j+1], arr[j])
            flag = 1

    if(flag == 0)
        break
```

(3) SELECTION SORT:

→ Finding the maximum in the collection & swapping it to the end of the collection.

→ Now, repeat leaving out the last element.

Time Complexity: $\Theta(n^2)$

OR we can find min. in the array swapping it to the start of the collection.

Pseudo Code:

```
for (i: 0 → i<n-1)
    min_index = i
    min = a(i)
    for (j: i+1 → j<n)
        if (a(j) < min)
            min = a(j)
            min_index = j
    swap(a(i), a(min_index))
```

(4) MERGE SORT:

→ The Idea is to merge two sorted arrays in linear time complexity.

→ It implements Divide & Conquer Algorithm. It that, we divide the array into equal sorted arrays and then merge them.

→ Here, we will use two functions:

merge(): to merge the two sorted arrays.

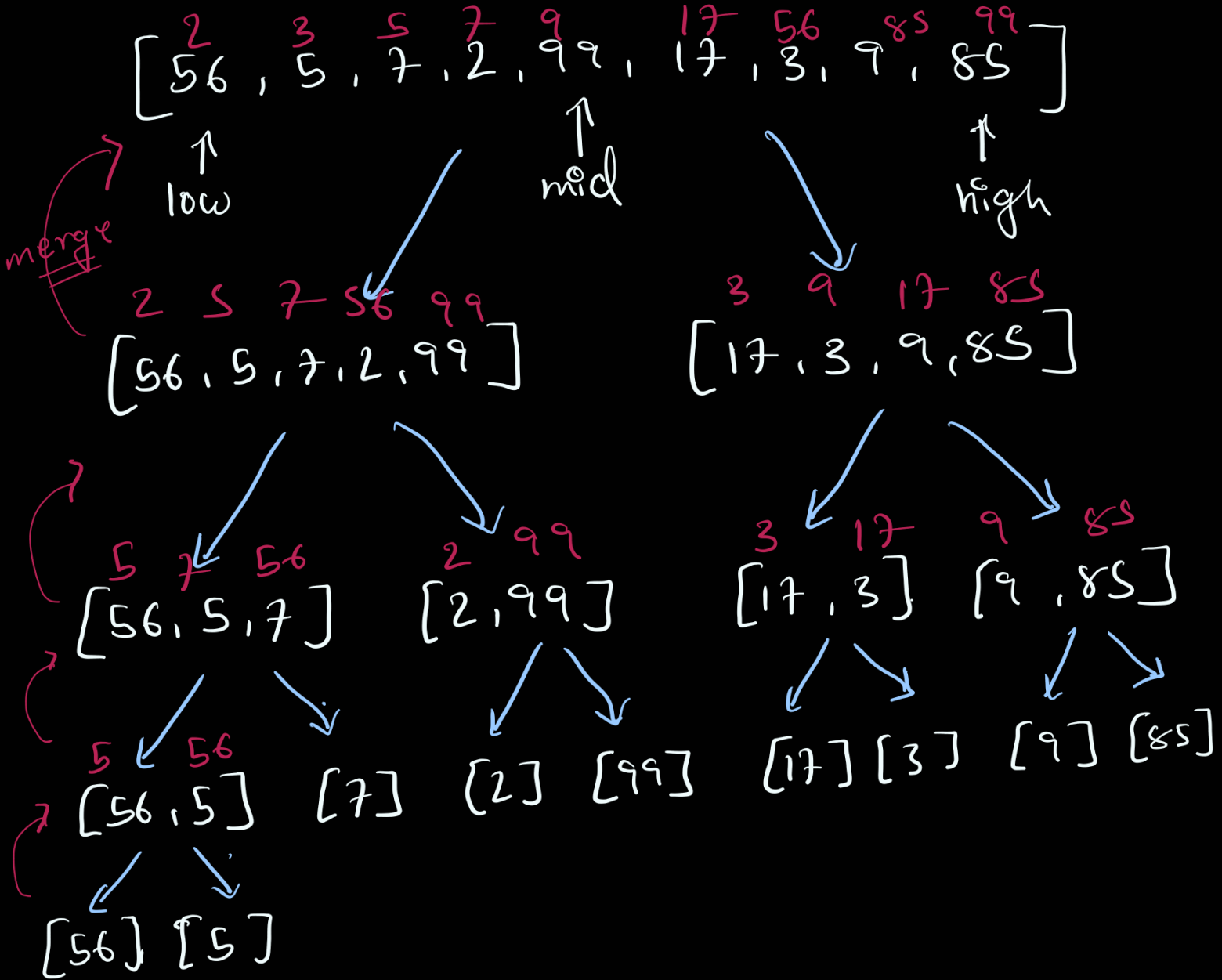
mergesort(): to divide the array into equal parts.

→ Let low = first element
high = last element
mid = low + (high - low)// 2 (// denotes int division)

→ Dry Run:

Let Array = [56, 5, 7, 2, 99, 17, 3, 9, 85]

Note: A single element array is sorted.



Pseudo Code:

```
void merge(arr, low, mid, high):

    vector<int> temp
    int i = low
    int j = mid+1
    while(i<=mid && j<=high)
        if(arr[i] > arr[j])
            temp.append(arr[j])
            j++
```

```
        else
            temp.append(arr[i])
            i++
    while(i<=mid)
        temp.append(arr[i])
        i++
    while(j<=high)
        temp.append(arr[j])
        j++
    arr = temp  // copy temp to array from [low to high]

void mergeSort(arr, low, high)
    if(low>=high)
        return;
    int mid = low+(high-low)//2
    mergeSort(arr, low, mid)
    mergeSort(arr, mid+1, high)

    arr = merge(arr, low, mid, high)
```

Time Complexity : There are $\log_2^n$ levels and at each level we traverse the array n times. $O(n \log n)$

Space Complexity : $O(n)$

↳ temp array

NOTE: Time Complexity Analysis :

Merge Sort :

Let $T(n)$ be the time taken to merge an array of length n.

$$\therefore T(n) = T(n/2) + T(n/2) + Cn$$

(C = const.)

$$\therefore T(n) = 2T(n/2) + Cn$$

$$= 4T(n/4) + 2Cn$$

$$= 8T(n/8) + 3Cn$$

$$\therefore T(n) = 2^k T(n/2^k) + kCn$$

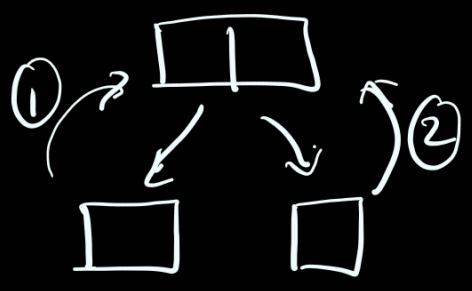
$$\therefore T(n) = (2)^{\log n}\left(T\left(n/2^{\log n}\right)\right) + c\,n\log n$$

$$n/2^{\log n} = n/n = 1$$

$$\boxed{T(1) = 0}$$

$$\therefore T(n) = Cn\log n$$

Take $n = 2$



$$T(2) = C(2)\log_2^2$$

$$= 2C = 2$$

$$\boxed{\therefore C = 1}$$

$$\therefore \boxed{T(n) = n\log n}$$

---

⑤ QUICK SORT :

→ Divide & Conquer algorithm

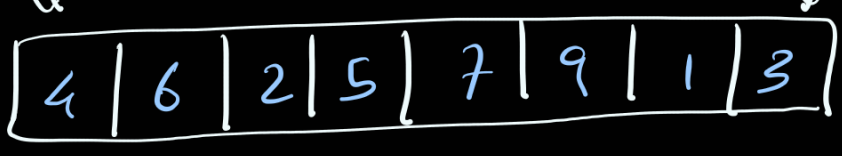
→ Unlike Merge Sort,
It uses an auxiliary stack space,
but does not use any extra array.

→ Steps :

① Pick a Pivot and place it in its correct place in the sorted array.

② Shift smaller elements on the left of the pivot and larger ones to the right.

→ Dry Run :

Let arr = 

[Here, we will take first element as pivot]

∴ pivot = 4

Now assign i = 1st index

j = last index

the i pointer will move forward & find the first element that is greater than pivot & the j pointer moves towards the pivot & find the first element that is smaller then the pivot.

Also, we need to ensure that, $i <= high - 1$

$j >= low + 1$

Once we find such elements,
i.e. arr(i) > pivot & arr(j) < pivot, we will swap arr(i) & arr(j)
Continue these steps until $j < i$.
At last we will swap the pivot with arr(j)

⇒ Pseudo code:
2 functions : ① Partition
② QuickSort (Recursive Part)

```
int partition (arr, low, high)
    pivot = arr(low)
    i = low
    j = high
    while (i <= j)
        while (arr(i) <= pivot && i < high)
            i++;
        while (arr(j) >= pivot && j > low)
            j--
        if (i < j) swap(arr(i), arr(j))
    swap(arr(low), arr(j))
```

```
    return j;

void QuickSort(arr, low, high)
    if(high <= low)
        return
    partIndex = partition(arr, low, high)
    QuickSort(arr, low, partIndex-1)
    QuickSort(arr, partIndex+1, high)
```

Time Complexity:

Best Case: $O(n \log n)$

Worst Case: $O(n_2)$

Avg. T.C.: $O(n \log n)$

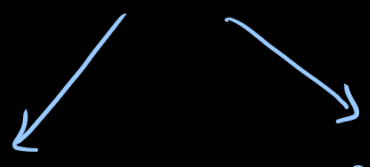
Space Complexity: $O(1)$

※ STACK :

→ An Abstract Data Structure.

" Last Come, First Out "

→ Operations :



Push

Inserting an element at the end of an array.

Pop

Removing last element of the array..

→ Stack in C can be designed using either an array or a LinkedList.

NOTE: Infix : $operand_1$, operator, $operand_2$

Postfix : $operand_1$, $operand_2$, operator

Prefix : operator, $operand_1$, $operand_2$

Eg: Convert the following to Postfix:

1) $(a+b-c)*d-(e+f)$

↳ $((a+b)-c)*d)-(e+f)$

Postfix : $ab+c-d*ef+-$

2) $4\hat{}2*3-3+8/4/(1+1)$

Postfix : $42\hat{}3*3-84/11+/+$

3) $a+b/c*(d+e)-f$

↳ $a+(b/c)*(d+e)-f$

↳ $(a+((b/c)*(d+e)))-f$

Postfix : $a\ bc/de+*+f-$

Rules : (Using Stack) → stack, → Output List Infix → Postfix

Current Element	Operation
(1) Operand	Append at the end of the output list.
(2) Opening Bracket	Push onto the stack
(3) Closing Bracket	Pop the stack, until the corresponding opening bracket is popped out.

Current Element	Operation
(4) Operator	While top of the stack has a operator of higher or equal precedence, pop the stack. Then, push the operator onto the stack.

NOTE:

⇒ Whenever an operator is popped out of the stack, the operator must be added at the end of the Output list.

For Infix to Prefix: 1. Reverse the infix
2. Infix → Postfix
3. Reverse that answer.

QUEUES:

"First In First Out"

Functions:

- Enqueue
- Dequeue
- Show

Implementations:

Global variables:

- size
- rear
- front

=> Implementation Using Arrays:

Initialize:

size = 0

rear = -1

front = -1

✳ BINARY SEARCH TREES:

→ Depth: Depth of a node is the number of edges from the root to that node.

→ Height:

length of the longest path from node to a leaf.

→ Depth(Tree) = Height(Tree)

↓ depth of the deepest leaf

↓ height of the root

(But As of now, depth of a tree = 0)

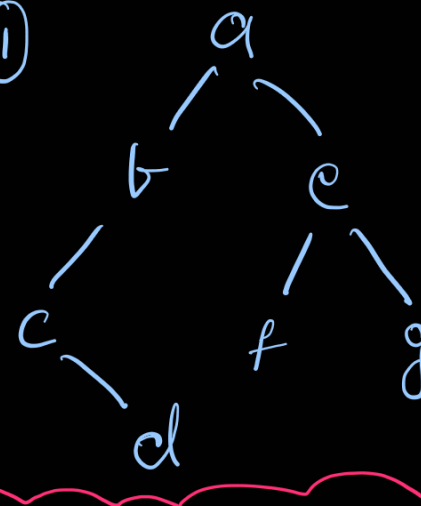
Implementation :

```
struct Tree {
    int data;
    struct Tree* FirstNode;
    struct Tree* SecondNode;
}
```

Tree Traversals :

Inorder : Left Root Right
Preorder : Root Left Right
Postorder : Left Right Root

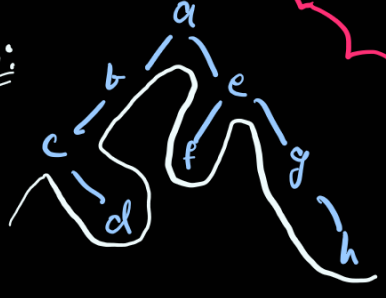
Eg: (1)



Inorder : c d b a f e g h
Preorder : a b c d e f g h
Postorder : d c b f h g e a

Trick : **Inorder** : Left → Root → Right
start traversing from the leftmost root.
End at the rightmost leaf.

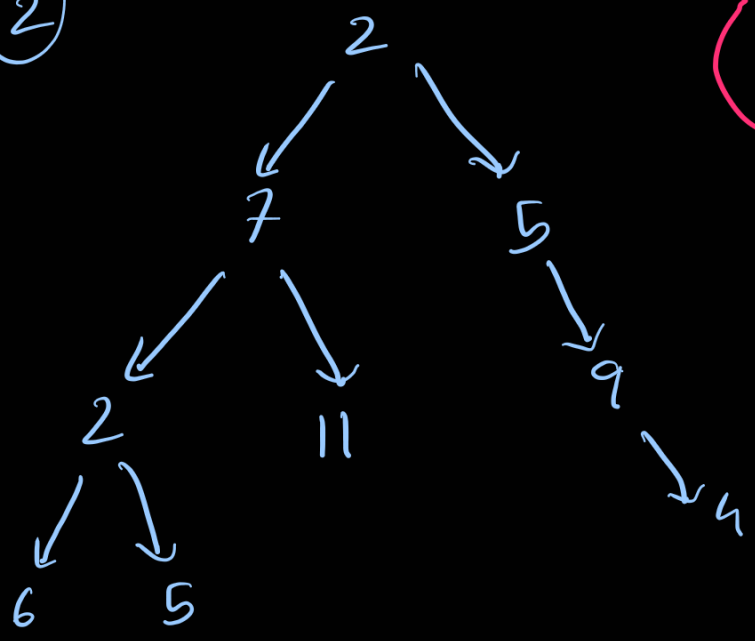
Eg:



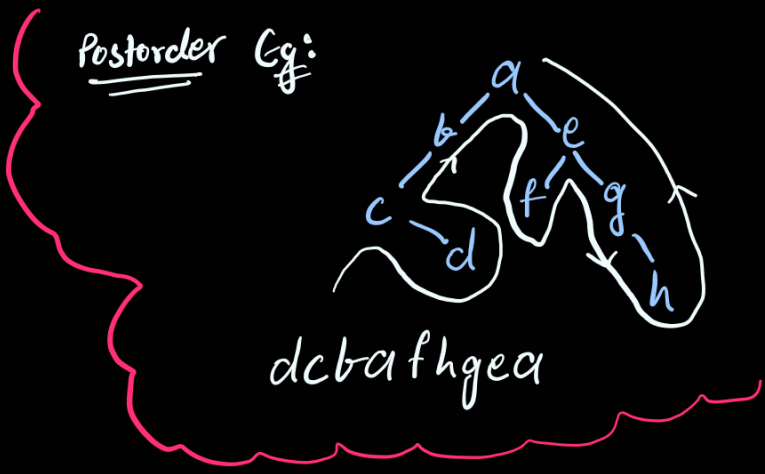
Preorder : Root → Left → Right
Start traversing from the Root then left and then to the right.

Postorder : Left → Right → Root
Same as inorder but pick the root during return traversal.

(2)



Postorder Eg:



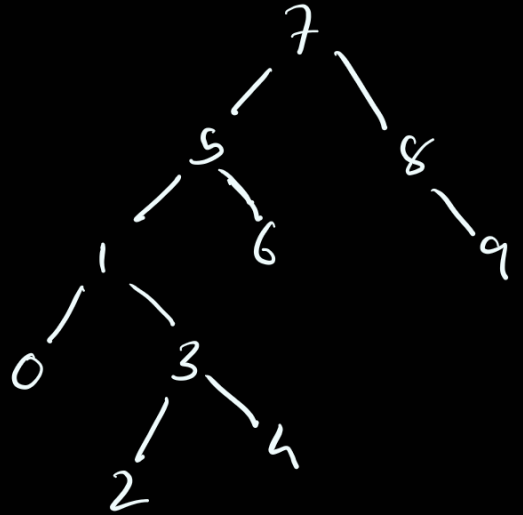
Inorder: 6 2 5 7 11 2 5 9 4

Preorder: 2 7 2 6 5 11 5 9 4

Postorder: 6 5 2 11 7 4 9 5 2

Binary Search Tree: smaller on the left side, greater on the right side.

Eg: 7 5 1 8 3 6 0 9 4 2

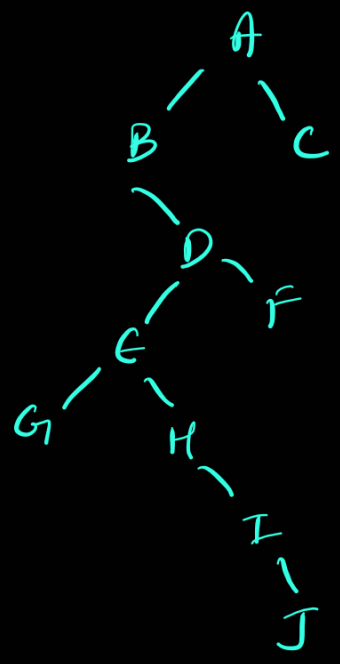


Inorder: 0 1 2 3 4 5 6 7 8 9

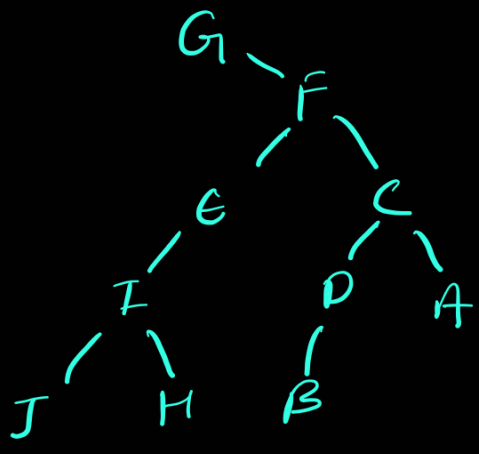
Preorder: A B E C D
Inorder: B E A D C

(2)

Post: G J I H E F D B C A
In: B G E H I J D F A C
Pre: A B D E G H I J F C



Post: J H I E B D A C F G
Pre: G F E I J H C D B A
In: G J H I E F B D C A

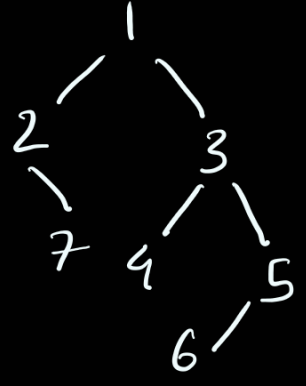


(3) Given: Pre: 1, 2, 7, 3, 4, 5, 6
In: 2, 7, 1, 4, 3, 6, 5
Construct the tree.

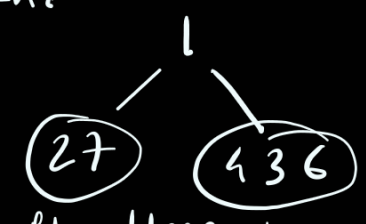
∴ Root of T = 1

To check for the left side of the tree, check Inorder

Pre:



In:



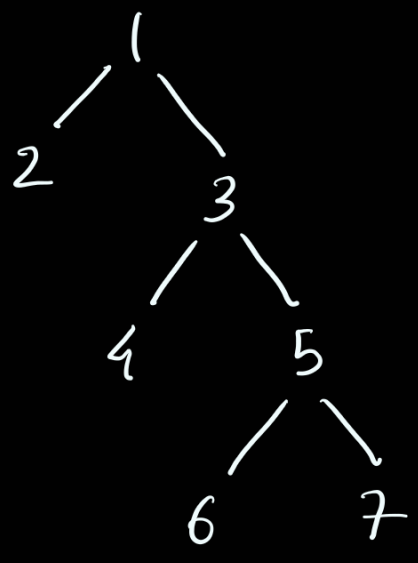
∴ Left subtree is not empty. } Root of the left subtree = 2

NOTE: A Full Binary Tree, is a binary tree where every node except the leaf have two children.

(4) Full Binary Tree:

Pre: 1, 2, 3, 4, 5, 6, 7 — Root → L → R

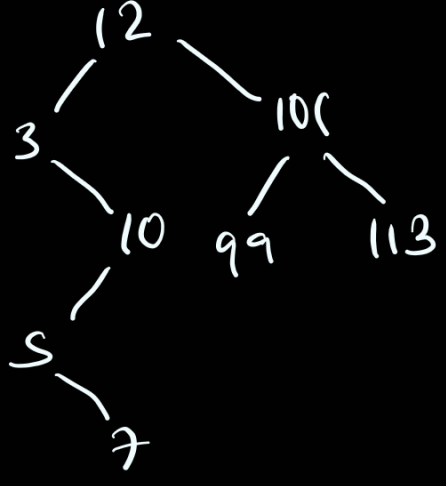
Post: 2, 4, 6, 7, 5, 3, 1 — L → R → Root



1 2 3 4 5 6 7

2 4 6 7 5 3 1

(5) Create a BST using numbers 12, 3, 10, 5, 7, 101, 99, 113



Successor of 10 = 12

Predecessor of 10 = 7

Smallest element = first leftmost node without a left child.

largest element = rightmost node (1st rightmost node without a right child).

Successor of an element = either the leftmost node of the right subtree, or the first 'right' ancestor.

If you reach the root in this manner, then that number cannot have a successor.
[If you are on the rightsubtree of the root].

Predecessor of an element = either the rightmost node of the leftsubtree or the first 'left' ancestor.

↳ Meaning:
start from the current element,
if the parent is on the right side, continue traversing,
if the parent of the curr node is on the left side, then that will be our desired answer.

If you reach the root in this manner, then that number cannot have a predecessor.
[If you are on the leftsubtree of the root.]

Eg: Let ele = 10
to find the predecessor:

① Check the right subtree:
no right subtree

② Then, check the ancestors:
3 → left ancestor
12 → right ancestor
STOP

Ans = 12

Inorder - Traversal of a BST:

```
Inorder (x)
if n = NULL
   return
Inorder (left[x])
print key(x)
Inorder (right[x])
```

Time complexity = O(n)

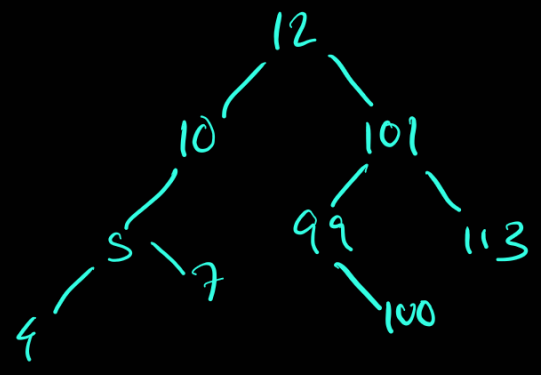
Tree Search

root → x, target → k

```
Search (x, k)
    if x = NULL
        return
    if key(x) = = k
        return.
    if k < key(x)
        Search (left[x], k)
    else
        Search (right[x], k)
```

Time Complexity = O(h)

Write pseudocode for Insertion & deletion of a node.



Fenwick Trees (Binary Indexed Trees)

Let A : | 4 | 3 | 8 | 11 | 20 | 6 |

Task: Given two indices

$0 \le i \le j < n$

Compute the sum of the elements from A[i] to A[j].

For example,

if i=2 and j=4,

then sum = A[2] + A[3] + A[4]

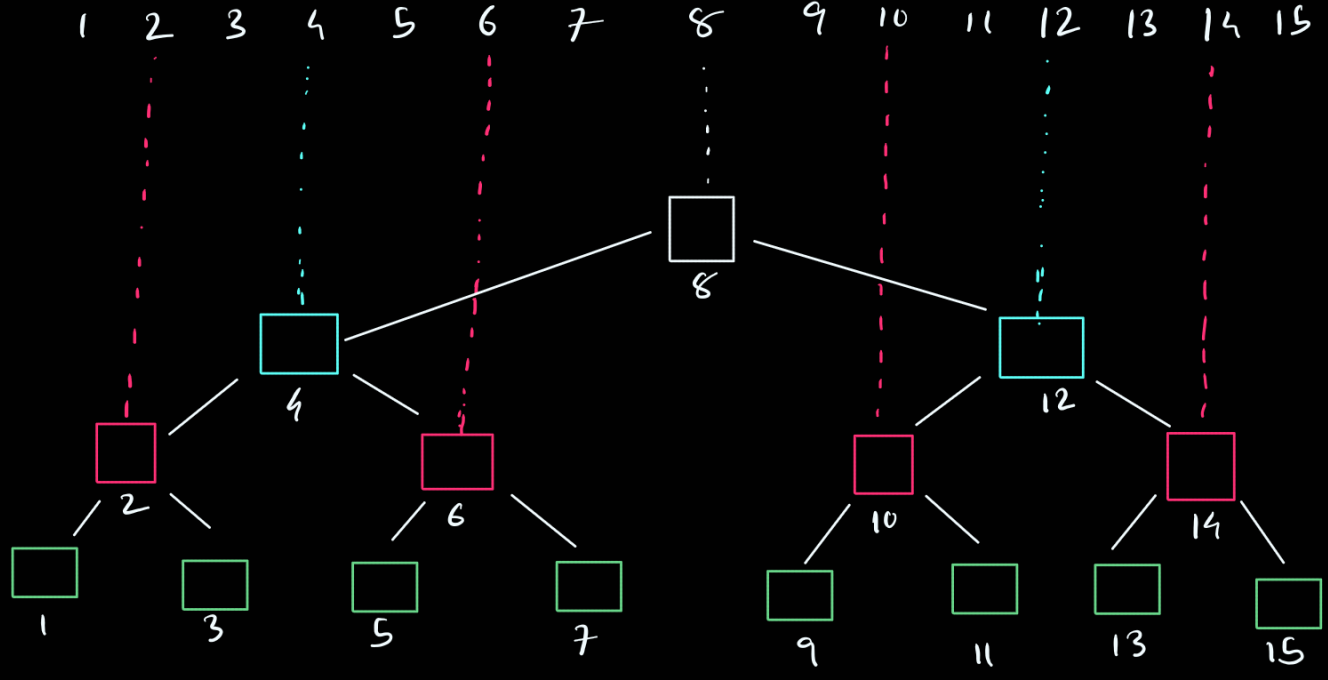
= 39

For multiple queries, this task can be easily done in O(1) time complexity using prefix sum.

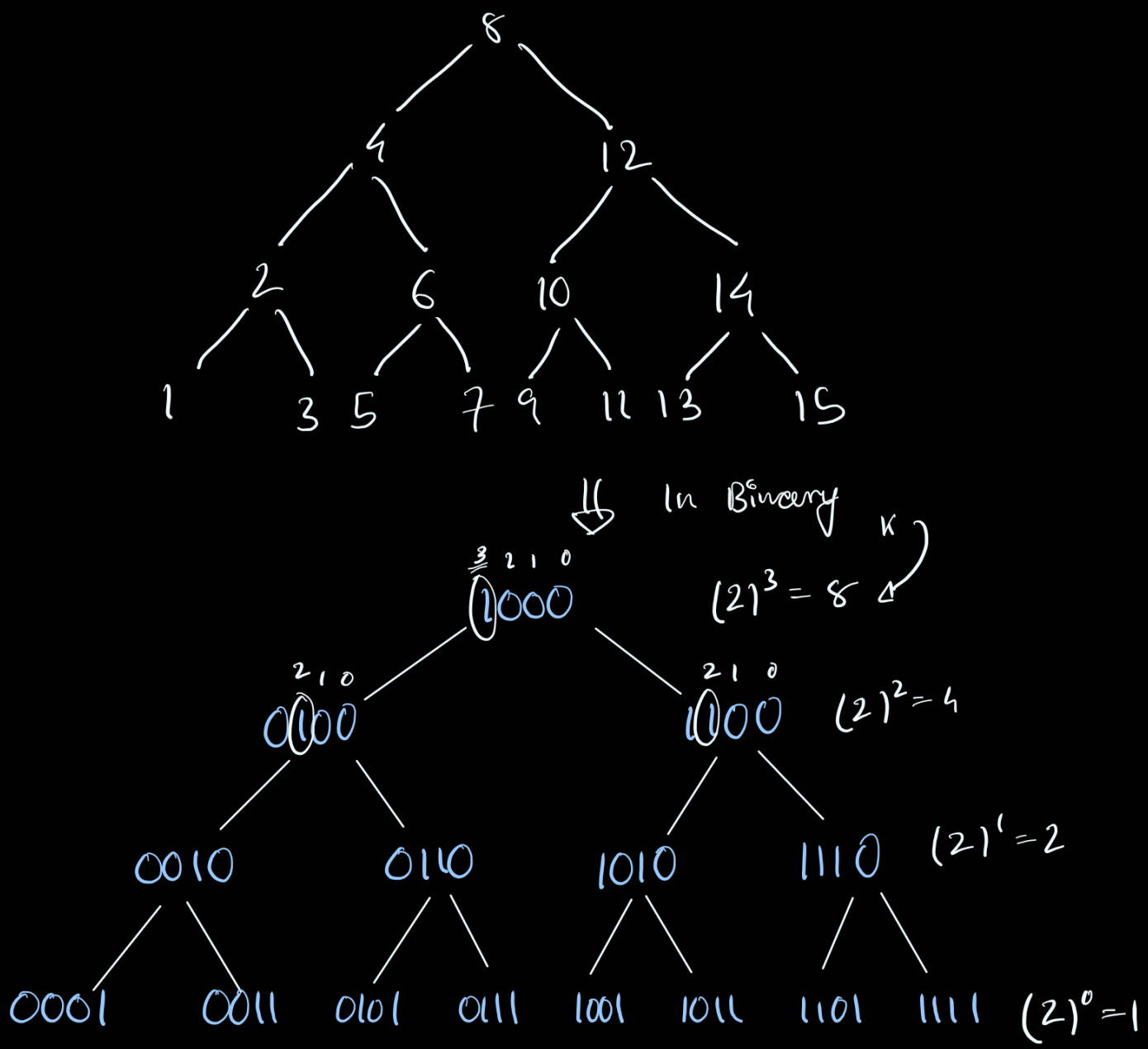
Problem: If any ele. of the original array has changed, then the prefix sum has to be recalculated.

Solution: Fenwick Trees

Eg: A : | 8 | 3 | 2 | 7 | 14 | 6 | 18 | 23 | 19 | 12 | 9 | 1 | 15 | 27 | 4 |

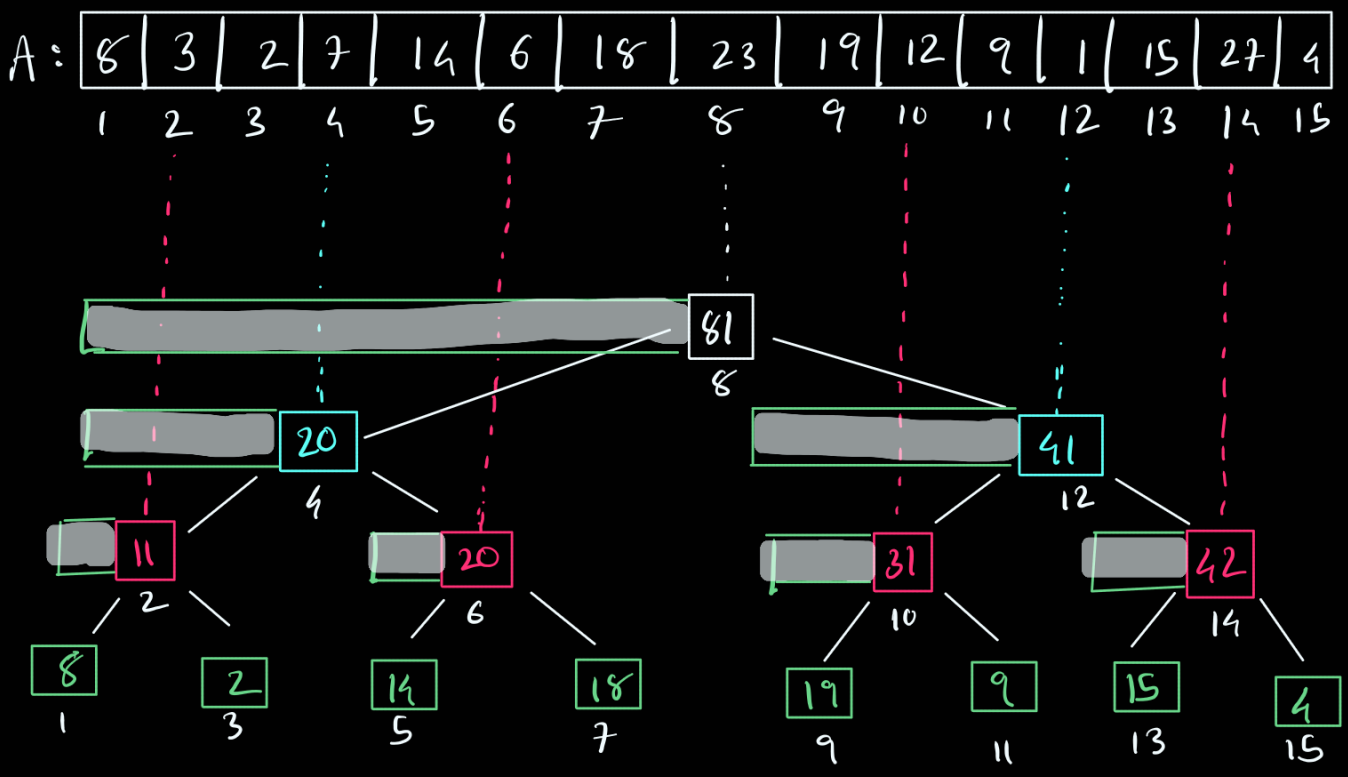


Write indexes in binary:



Let the value of that node be the sum of 'K' elements before that index (including that index).

Now, calc. the values at each node



8	11	2	20	14	20	18	81	19	31	9	41	15	42	4

→ interrogation tree

→ update tree

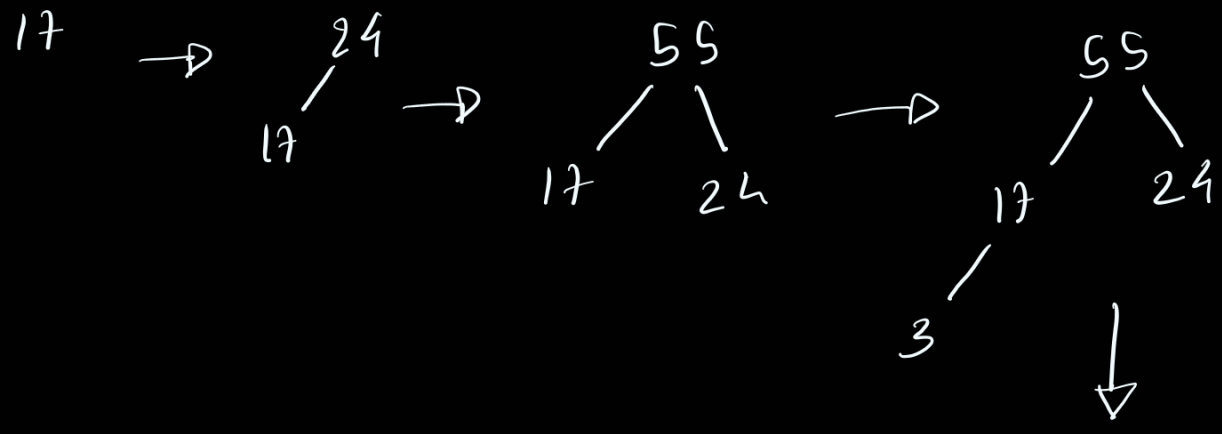
mirror of $i = 16 - i$

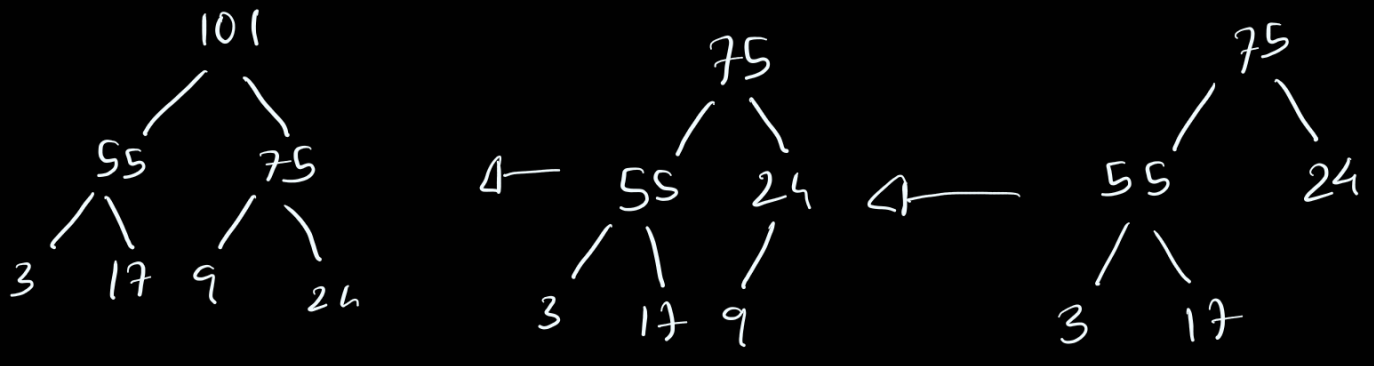
Heap:

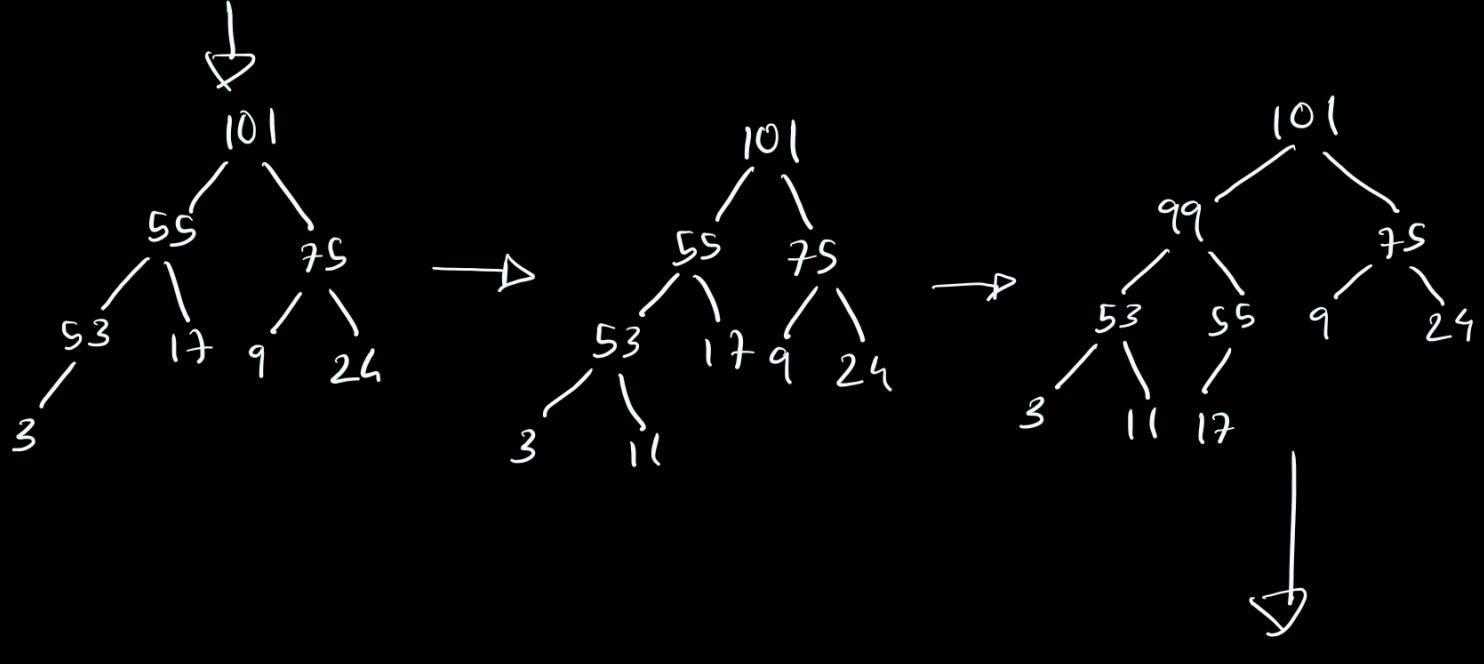
Arrange in descending order:

17, 24, 55, 3, 75, 9, 101, 53, 11, 99, 40, 6

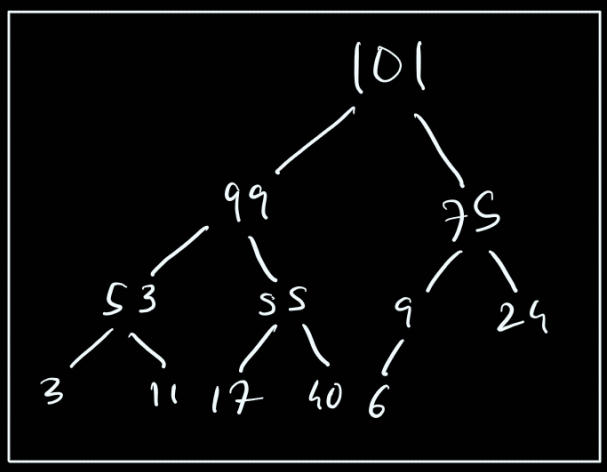
Step1: Inserting every element to form a max heap:





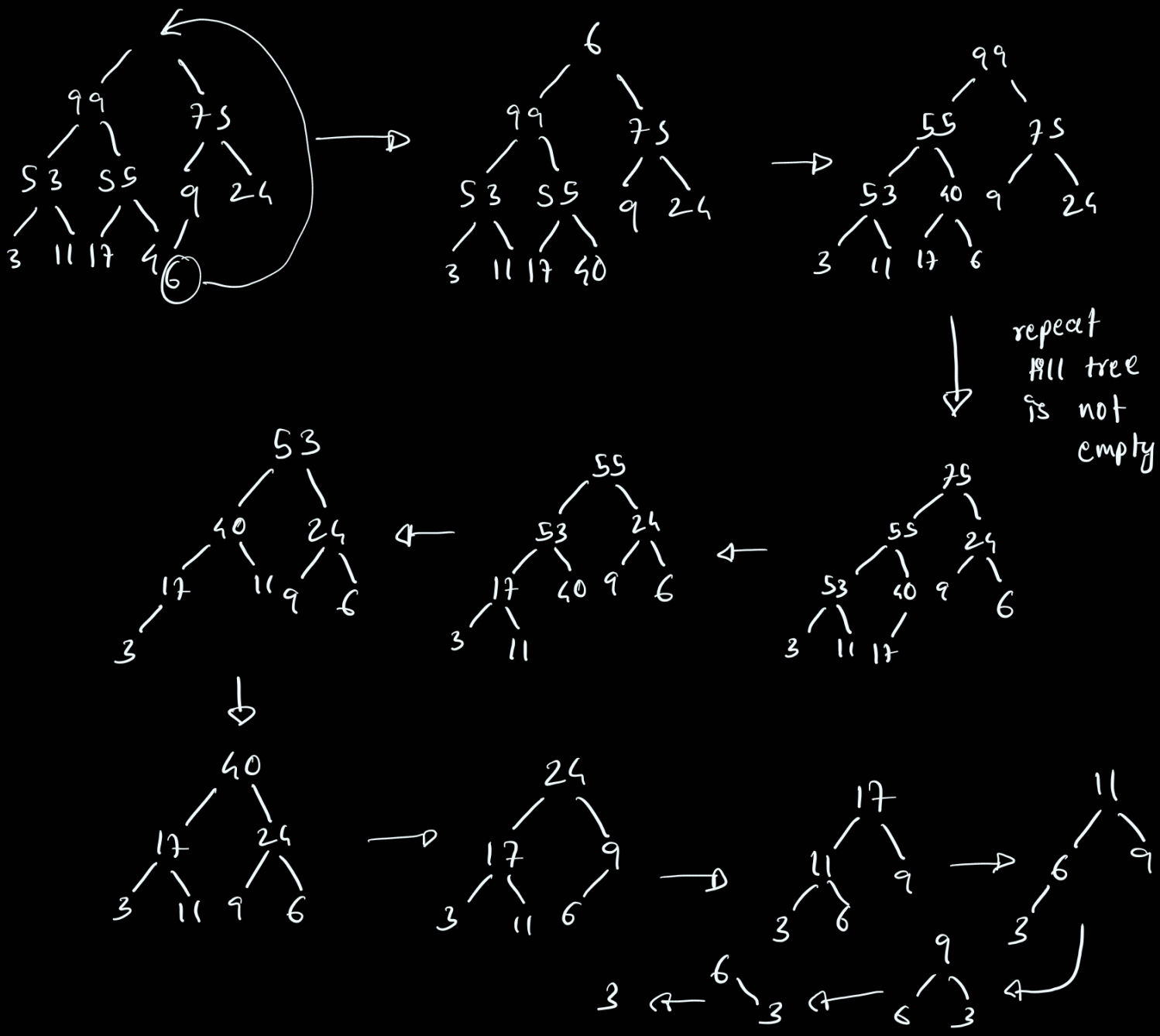


Final max heap



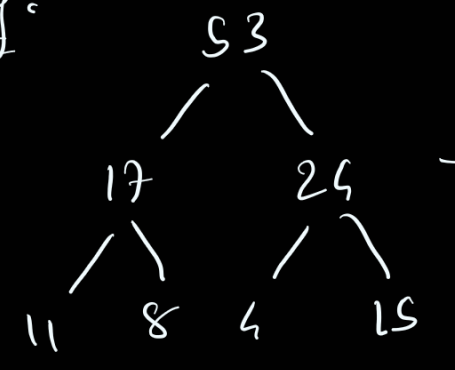
Step 2: Deleting root and rearrange into max heap.

Order: [101, 99, 75, 55, 53, 40, 24, 17, 11, 9, 6, 3]



Implementing Max Heap Using an Array:

Eg:



can be represented as

53	17	24	11	8	4	15
0	1	2	3	4	5	6

i =>

the parent of i^{th} node = $\left[\frac{i-1}{2}\right]$

the children of i^{th} node = $2i+1$, $2i+2$

Task: Convert the following into the max heap

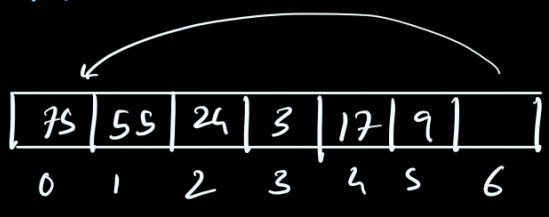
17, 24, 55, 3, 75, 9, 101

101	55	75	3	17	9	24
0	1	2	3	4	5	6

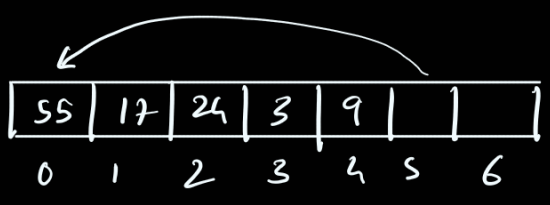
Step 1: Insert at the end of the arr

Step 2: Compare it with its parent.

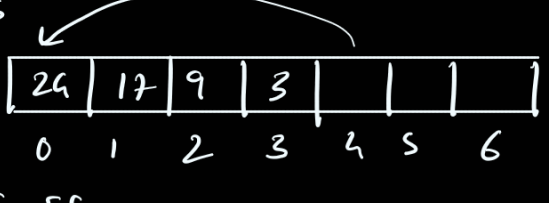
Deletion:



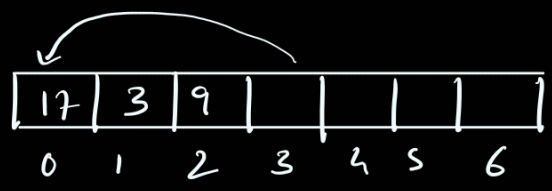
101



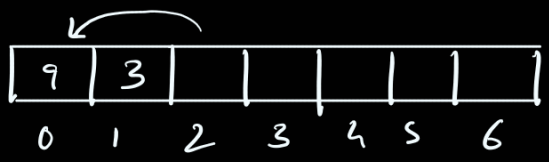
101 75



101 75 55



101 75 55 24



101 75 55 24 17

↳ [101 75 55 24 17 9 3] Final desc. order

$$\frac{n}{4} + \frac{n}{8} \times 2 + \frac{n}{16} \times 3 + \cdots \cdots \infty$$

$$= \left(\frac{n}{4} + \frac{n}{8} + \cdots \infty \right)$$

$$+ \left(\frac{n}{8} + \frac{n}{16} + \cdots \right)$$

$$+ \left(\frac{n}{16} + \frac{n}{32} + \cdots \right)$$

$$\vdots$$

$$\infty$$

$$= n/2 + n/4 + \cdots \infty$$

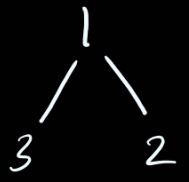
$$= \boxed{n}$$

Time Complexity of Building Heap from last level.

Practice Questions on Heaps:

1] Input: 2 3 1 5 4 8 7 6

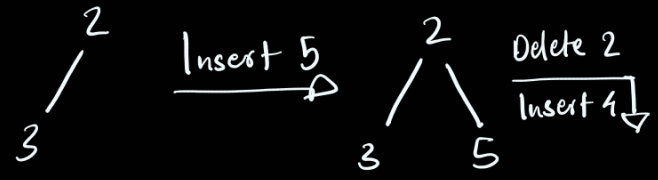
2- Sorted: 1 2 3 4 5 6 7 8



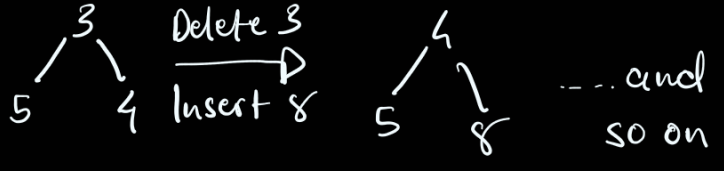
→ If k-sorted then make a min heap of k+1 elements.

↳ Now delete and Insert turn by turn

Ans: 1



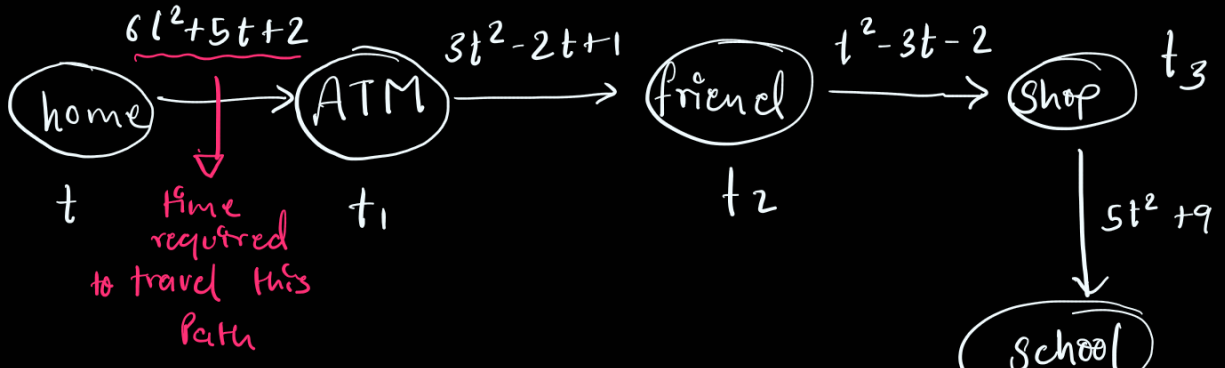
Ans: 1 2 3 4



Time complexity = $(n-k)\log k + n\log k$

$\therefore O(n \log k)$

Task	※ Comparisons	Optimal?
① Finding max and second-max	$n + \lceil \log n \rceil - 2$	✓
② Finding max and n^{th} max	$\left\lceil \frac{3n}{2} \right\rceil - 2$	✓
③ Sorting	$O(n \log n)$	✓
④ Sort 4 numbers	5	✓
⑤ Finding $(n/2)^{th}$ max (median)	$O(n)$	✓



Que: What is the time to reach from home to school?

(1) Does the function have a closed-form expression? YES

(2) Is the function a polynomial in t? YES (Degree = 16)

(3) Can it be computed from the given information? (YES)

$$t_1 = t + 6t^2+5t+2$$
$$= \boxed{6t^2+6t+2}$$

$$t_2 = t_1 + 3t_1^2 - 2t_1 + 1$$
$$\boxed{t_2 = 3t_1^2 - t_1 + 1}$$

$$t_3 = t_2 + t_2^2 - 3t_2 - 2$$
$$= \boxed{t_2^2 - 2t_2 - 2}$$

$$t_4 = t_3 + 5t_3^2 + 9$$
$$= \boxed{5t_3^2 + t_3 + 9}$$

Suppose you have an array of n+1 integers (where n = even)

How do you find the correct position of a number?

→ lcnt = 0, rcnt = 0 } if arr(i) < number
lcnt++
else
rcnt++

then correct position = **lcnt**

Now for a median,

lcnt = rcnt (if array has odd no. of elements)

Hence, to find the median, pick a random number and check if **lcnt == rcnt** is true or not.

Now, suppose that you eliminate 2% of an array each time you traverse through it, s.t.

} In a sorted array

Suppose you take n comparisons to eliminate this 2% of the array.

then, $T(n) = T\left(\frac{98n}{100}\right) + n$

$= T\left(\frac{98n}{100} \times \frac{98}{100}\right) + n + \frac{98}{100}n$

$\vdots$

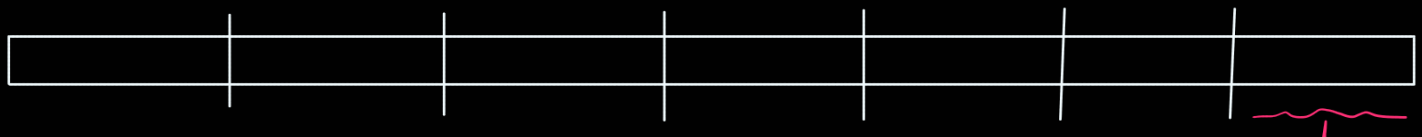
$= T\left(\left(\frac{98}{100}\right)^{50} n\right) + \frac{n}{50}(1 + 2 + \cdots + 50)$

$\therefore T(n) \approx 50n$

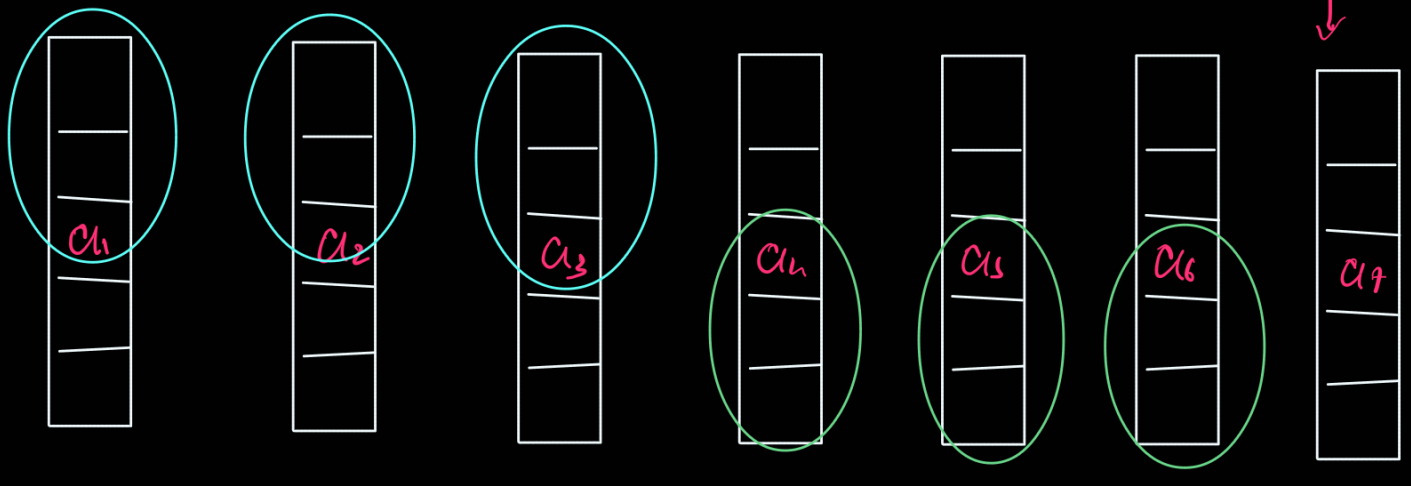
Since, the median is exactly $n/2$ numbers lesser than it, and $n/2$ numbers greater than it.

Thus, a number x is not a median if, it has more than $n/2$ numbers greater than it / lesser that it.

Again consider an array of 35 elements, divide it into 7 equal parts

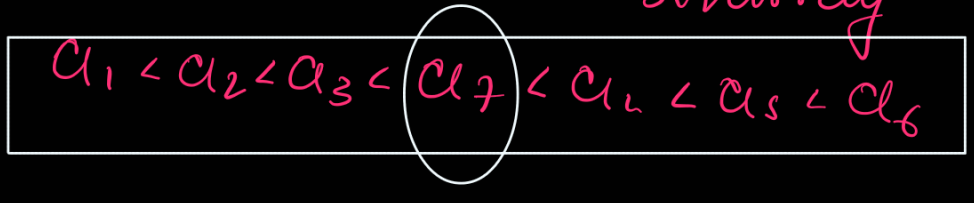


Now, sort each subarray.



$\{a_1, \ldots, a_7\}$ → median of their respective subarray

Suppose



Hence,

Blue elements < Cl7 < green elements

$\therefore \frac{3}{5} \times \frac{n}{2} = \frac{3n}{10}$ of the elements are greater than every blue numbers.

And, $\frac{3n}{10}$ of the elements are less than every green elements

---

MOM → Median of Median
MOF → Median of Fives
MOT → Median of Threes

```
MOM(A, l, n, k)
{
    m ← Ceil(n/5)
    for( i ← 1 to m)
        M(i) ← MOF(A, 5*i-4, min(5*i, n));
    median ← MOM(M, 1, m, Ceil(m/2));
    r ← PARTITION(A, l, n, median);
    if(k<r)
        return MOM(A, l, r-1, k);
    else if(k>r)
        return MOM(A, l, r+1, k-r);
    return median;
}
```

MOM (A, 1, n, K) finds the Kth smallest element in A, b/w the indices 1 & n.

MOK (A, 5i-4, 5i) sorts the element in an array A, between the index 5i-4 and 5i.

PARTITION (A, 1, n, index) places the median in it's correct place, and place all elements of A less than median and then sort all elements of A greater than median to it's right.

```
MOF (A, start, end)
{
    subarr ← extract A[start : end];
    sort (subarr)
    return subarr[ (end-start)/2 ]
}
```

```
PARTITION (A, 1, n, pivot)
{
    pivotidx = index (pivot)
    swap (A[pivotidx], A[n]);
    i ← 1
    for j ← 1 to n-1
        if A(j) < pivot
            swap (A[i], A[j]) // moves elements less than
            i++                  the pivot to it's left
    swap(A[i], A[n])
    return i // returns the correct (sorted) place of pivot
}
```

Explanation:

(1) Divide the array A into 5 parts.
(2) Find median of each of them.
(3) Find the median of medians.
(4) Find the correct place of the median of medians in a sorted array.
(5) let r = correct place(median)
if (k < r)
find in the left part of the array
if (k > r)
find in the right part of the array
if k == r
return median.

※ Analysis of Time Complexity:

→ Sorting each group of 5 take $O(5 \log 5)$
∴ Total time for sorting $ceil(n/5)$ groups
$\simeq O(n/5 \times 5 \log 5)$
$\simeq \boxed{O(n)}$

→ recursive call to find median of medians = $T(n/5)$

→ The median is chosen such that atleast $3n/10$ are less than the median and atleast $3n/10$ are greater than the median.

→ This ensures that we discard $3n/10$ elements in each recursive call.

→ PARTITION takes $O(n)$

→ Since we discard atleast $3n/10$, the worstcase remaining subarray will $7n/10$

→ Recurrence Relation:

$$T(n) = T(n/5) + T(7n/5) + O(n)$$

$$\therefore T(n/5) = T(n/25) + T(7n/25) + O(n/5)$$

$$T(7n/5) = T(7n/25) + T$$