DATA STRUCTURES

AND

ALGORITHMS

VOLUME - 1

Binary

JAVA Bit wise Operators

1)

Name	Operator	Description	Example
&	And		A & B

A	B	Result
0	0	0
1	0	0
0	1	0
1	1	1

25: 2 | 25, 2 | 12 - 1, 2 | 6 - 0, 2 | 3 - 0, 2 | 1 - 1 → 11001

15: 2 | 15, 2 | 7 - 1, 2 | 3 - 1, 2 | 1 - 1 → 01111

A = 25 → 11001
B = 15 → 01111

A & B → 01001 ⇒ 8 + 0 + 0 + 1 ⇒ 9

2^4 2^3 2^2 2^1 2^0
8 4 2 1

2) OR operator (|)

A = 25 → 11001
B = 15 → 01111

A | B → 11111

16 + 8 + 4 + 2 + 1 = 31

A	B	R
0	0	0
1	0	1
0	1	1
1	1	1

3) XOR (^) ⇒ A ^ B

A = 25 = 11001
B = 15 = 01111

A ^ B = 10110

2^4 2^3 2^2 2^1 2^0

16 + 0 + 4 + 2 + 0

8 10 1 0

A ^ B = 22

A	B	R
0	0	0
0	1	1
1	0	1
1	1	0

a = [4, 1, 2, 2, 1]

only one element will occure once
find it

num = 0
~~for (int i : nums)~~ for (int v : a){
n ^= v
n = n ^ v

4 | 0
2 | 0
1 | 0

100
000
100

100
001
001
21

001
011
010
2

4

→ find sum of squares of all the digits in a number

I/P => 23 => $2^2 + 3^2 = 4 + 9 \Rightarrow 13$

Note

% => gives reminder

/ => gives quotient

$11/4 = 2$

$11 \% 4 = 3$

```
        1
4)11 (2
  8
  3
```

dividend

quotient

Divisor

Reminder

Ans

```
num = 23
int sum = 0 ; int digit

while (num > 0) {
    digit = num % 10;
    sum += digit * digit
    num = num / 10
}
return sum
}
```

① iteration

→ 23 % 10 => 3

→ sum = 3*3 = 9

=> 23/10 => 2

num = 2

② nd iteration

2 % 10 => 2

sum => 9 + (2*2)

= 13

2/10 = 0

Data Structures & Algorithms

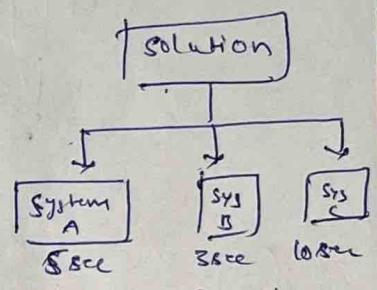
Big - O

↳ Good code
- → Readable
- → Scalable → Time complexity → Algorithm run fast, better time complex
  - → Amt of time it takes to run an Algorithm
- → space complexity → Amt of extra space our Algorithm requires.
  - → Algorithm takes less space, better space complexity

→ Big O → It is a performance parameter which give us different parameter to Rate our code base.

↳ Big-O is used to describe the Execution time required or, the space used.

Why we need Big O Notation?



Ex :- Calculate sum of n - natural numbers.

→ Given problem can run in different time in diff sys, so it purely depends on the Hardware so we dont have the exact runtime for this particular code base? (so we if we compare with time stamp → (Start, End) It gives different results in diff System. → so we require Big-O

→ * Big O doesn't depends on time, it depends on How many Steps we are performing

→ * The Main focus is to calculate the Amt. of work we do or, the no. of steps comparisons we perform.

Ex :-

```
List1 = ("A", "B", "C", "D")
List2 = ('x', 'y', 'z', 'L', 'm', 'A')

def Check value (input, value):
    for element in input
        if element == value:
            Print ('Available')

// fn calling
Check value (List1, "D")
Check value (List2, "A")
```

⇒ If we observe, If no. of element in List ↑, then the no. of iteration also ↑

5 Item → 5 time loop to run

10 items → 10 time loop to run

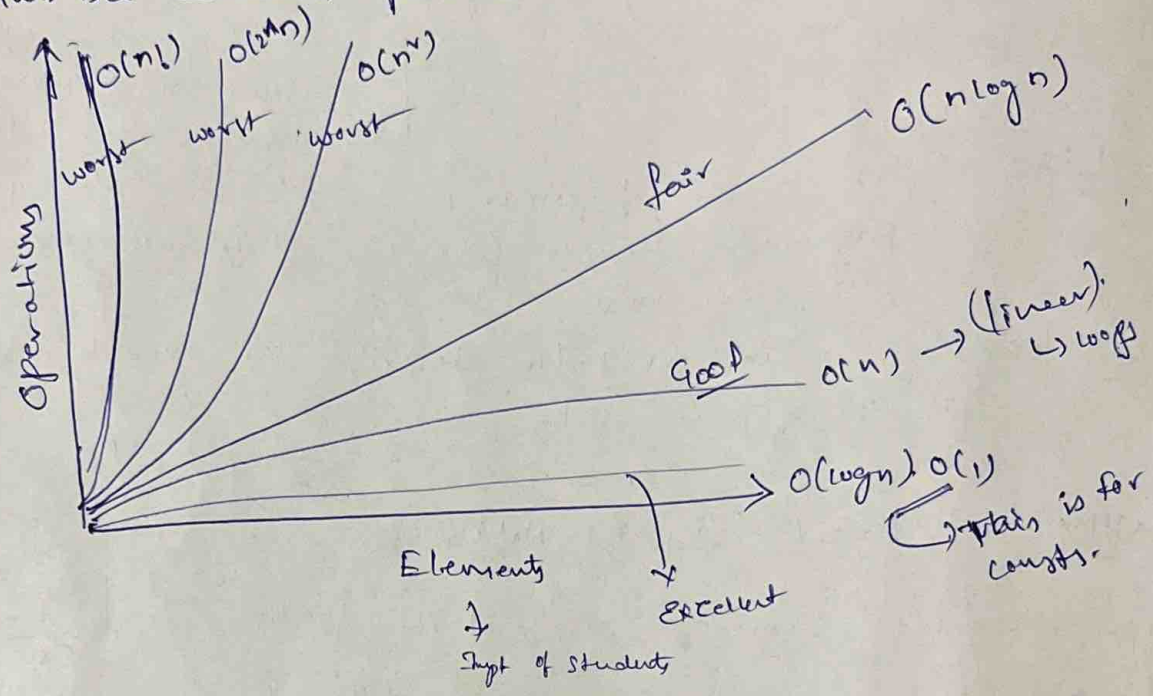
i.e Time complexity Increases linearly → i.e $O(n)$

→ no of element in the Input

→ Now that O(n) is the most common complexity

* If no. of operation is equal to no. of Input we follow a Linear path. That Linear path we call it as O(n)

↳ This is a common patterns with loops, (while, for)



O(1)

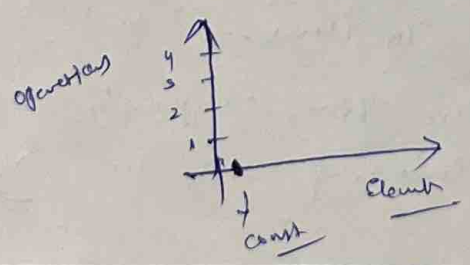
Ex:- Students = ['A', 'B', 'C', 'D', 'E']

```
def display Students (students):
    print (Students[0])  ⟹ O(1)
    print (Student[1])   ⟹ O(1)
display student (students)
```

} O(2) ≃ O(1)

→ The time complexity of this program is O(1), i.e constant i.e the number of operation we are performing & the no. of Input we are depending is "One"

↳ In our Example we five elements but still we are depending on the first Element → i.e irrespective of the Input size we ~~depend~~ dependent of 0th Element.



O(1) = O(10) = O(20) ≃ O(1)

↳ As them only const.

Counting operations

```
def randomFn (Students):
    first = Students [0]        → O(1)
    total = 0                   → O(1)
    newlist = []                → O(1)
    for student in students:
        total += 1              O(n)   → It totally depends on size of students
        newlist.append (student)  - O(n)
    print (newlist)             O(1)
    return total                O(1)
print (randomFun (['A', 'B', 'C'])) O(1)
```

↳ Total complexity :- $O(6+2n) \Rightarrow O(n)$

Simplification of Big O

Rules

1. → Focus on scalability → Take large number i.e $n \to \infty$
2. → Considering worst case scenario
3. → Remove all possible constants ⇒
4. → Consider different variable for different inputs
5. → Remove all non-dominants.

Rules

↳ $O(6+2n^{\to\infty}) \Rightarrow O(n)$

remove 6 → as 6 is very less than ∞

Rule 4 :-

↳ say In previous Example we Running loop twice for same list then $O(n) + O(n) = O(2n) \Rightarrow O(n)$

$O(n^2)$ → Quadratic Time

↳ This usually happens when we Running loop inside a loop.

Ex:-

```
def random_fn (num_list):
    total = 0                    O(1)
    for num1 in num_list:        O(n)
        for num2 in num_list:    O(n)
            print (num1, num2)   O(1)
            total += 1           O(n^2)
    return total
print (random_function([1,2,3,4])
```

Total 10 operations

i.e n^2

$O(1 + n^2 + n^2 + 1) = O(2 + 2n^2)$

$= O(n^2)$

Ex:-2

num list 1 = (1, 2, 3, 4, 5) ⇒ length $n = 5$

num list 2 = [5, 6, 7, 8, 9, 3] ⇒ $m = 6$

$5 \times 6 = 36$

```
def random function (n):
    total = 0
    For n1 in numlist1:
        for n2 in numlist2:
            print (n1, n2)
            total += 1
    return total
print (random fn (n))
```

⇒ $O(n * m)$

Rule-5 (Remove all non-dominants)

```
numlist = [1, 2, 3, 4, 5, 6, 7]

def random_fn():
    total = 0                          O(1)
    all_integer = True                 O(1)
    For num in numlist:
        print(num)                     O(n)

    For num in numlist:
        for n2 in numlist:
            print(num, n2)             O(n^2)
            total += 1                 O(n^2)
    msg = "Rule 5 - Remove all non-dominants"   O(1)
    return total                       O(1)
print(random_fn)
```

$\rightarrow O(\cancel{4}^{0} + \cancel{n} + \cancel{2}n^2) = O(n^2)$ remove all non nominant

O(n!) → very Bad, very Rare

Space Complexity

↳ Space complexity of an Algorithm represents the Amt. of Extra Memory space needed the algorithm in it's life cycle.

↳ We usually have 2 type of space requirements

1) To store our Input Data

* 2) Extra space that we need to execute our program.

1)
```
def all_cubes (items)
    result = 0
    for item in items:
        result = pow(item[0], 3)
    print (result)

items = [2, 3, 4, 5, 6, 7]
print ( all_cubes)
```

Space complexity

$\Rightarrow O(1)$

↳ We need only on space as we operating it on the one element irrespective of Array size.

2)
```
def all_cubes (items)
    results = []
    for item in items:
        results.append ( pow (item , 3))
    print (result)
```

(Space Complexity)

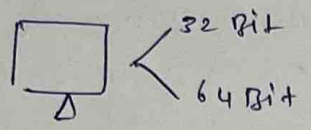
$\Rightarrow O(n)$

3)

Memory

↳ Usually divided into Memory slots.

1 Memory slot = 8 Bit
8 Bit = 1 Byte } i.e 1 Memory slot = 1 Byte



If you consider working on 32 Bit System
↳ If you declare integer it is going to 32 bit = 4 bytes
For 64 Bit computer it needed → 8 bytes for Each element

Ex:- int x = 2 ⇒ It is storing integer so it will take 32 bit = 4 Bytes

x = 2
↳ Its Memory address 601

It means If I want to find the Address Again I just need to visit this Address

1 Byte

601	602	603	604	605	606	607	608
609	610	611	612	613	614	615	616
617	618	619	620	621	622	623	624
625	626	627	628	629	630	631	632
633	634	635	636	637	638	639	640
641	642	643	644	645	646	647	648

} 1 Byte slot Each Box

(Memory canvas)

→ Let's say If we are working with 16GB (or) 32GB (or) 64GB of Memory like the above there will be Millions of Memory slot

→ Each memory address is unique & s/m tech this value with the help of particular Memory address. → Memory Address are Hexadecimal
Say 601 → will be represented as (0x259 Hexadecimal)

→ The Value that is stored inside the Slot is in "Binary Format"

Ex:- x = 2 → 0000 0010.

1 Byte
8 Bits

601	602	602	604
000 0010	0	0	0

↳ Remaining slots will be filled with "Zero"

Memory Example

Let m = [2, 3] => Now both their value are together that means they have to be stored back to back in memory cannot.

i.e value 2 → stores in 4 Byte. & 3 → stored in 4 byte.

601	602	603	604	605	606	607	608
000 0010	0	0	0	0000 0011	0	0	0

2 – 0000 0010

3 – 0000 0011

We are placing zero at right because of Endianness concept

⇓

What is Endianness :-

↳ different languages read their text in diff order. for Example → English (left → Right), Arabic (Right to left)

↳ This is what exactly Endianness to computers.

↳ Endianness means that the bytes in computer are stored & read in a certain order.

→ for strings And special characters follow ASCII values.

A - Z → 65 - 90 | 0 - 9 → 48 - 57

a - z → 97 - 112 | special symbol → 0 - 47, 58 - 64, 91 - 96, 123 - 127

logarithm

$$\boxed{\log_b(x) = y \quad \text{if } b^y = x}$$

→ $\log_b(n) = y$ ⇒ In C.S b is always (2), i.e base is always 2

Ex :- $\log_b(x) = y \rightarrow b^y = x$

↳ $\log_2(x) = y \rightarrow 2^y = x$

!! be always talk with "n" notation

$\log_b(n) = y \rightarrow b^y = n$

$\boxed{\log_2(n) = y \rightarrow 2^y = n}$

→ Remember base always "2" Even if it is not Mentioned.

Ex $\log(1)$ — ? → 0

$\log(1) = y \rightarrow 2^y = 1 \Rightarrow 2^0 = 1 \Rightarrow 1 = 1$

Ex :- $\log(2) = y \rightarrow 2^y = 2 \Rightarrow 2^1 = 2 \Rightarrow 2$

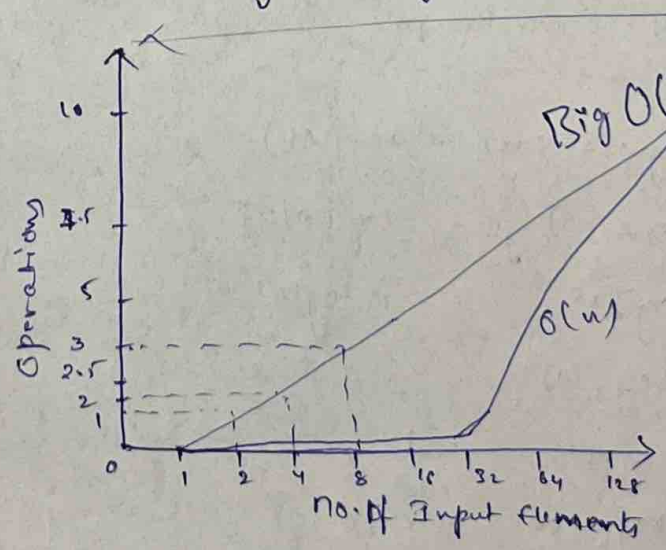
↳ $\log(2) = 1$

Ex :- $\log(16) = y \quad 2^y = 16 \Rightarrow 2^4 = 16 \Rightarrow y = 4$

* $\log(1) \Rightarrow 0$
$\log(2) \Rightarrow 1$
$\log(4) \Rightarrow 2$
$\log(8) \Rightarrow 3$
$\log(16) \Rightarrow 4$
$\log(32) \Rightarrow 5$
$\log(64) \Rightarrow 6$
!!
$\log(512) = 9$
$\log(1024) = 10$
$\log(1048576) = 20$

Observation

* When you double the value of "n", the value of "log" only increases by 1



log is so powerful.

↳ If even Input Increases but the complexity won't, so It is better than O(n)

Ex:- Say we have Array, And at Every step we are Eleminating the half of the Array.

Ex:- [0, 1, 2, 3, 4, 5, 6, 7]

Step 1 [0, 1, 2, 3]

Step 2 [0, 1]

Step 3 0 left

So If we have 8 Element we only performing 3 steps

i.e $2^3 = 8$

$\log(3) \simeq \log(n)$

for n input

Ex2: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]

(1) [0, 1, 2, 3, 4, 5, 6, 7]

(2) [0, 1, 2, 3] => [0, 1] => 0

(3) (4)

4- steps. $\xrightarrow{\text{for}}$ 16 Elements

i.e $(2^4) = 16$

So Even we double the Input the no. of steps increases by only '1'

Laws of Logarithms

1) $\log A + \log B = \log(AB)$

2) $\log A - \log B = \log(A/B)$

3) $\log A^n = n \log(A)$

4) $\log_b(1) = 0$

5) $\log_b(b) = 1$

6) $\log_b(b^k) = k$

7) $b^{\log_b(k)} = k$

Time Complexity Problems

1)

```
int a = 10,  b = ..  — O(1)
int b = 0            — O(1)
for (i=0 ; i<N ; i++){
        a = a + rand()
}                    } O(n)
for (j=0 ; j<m ; j++){
        b = b + rand()
}                    } O(m)
```

$O(1) + O(1) + O(n) + O(m)$

$O(n+m)$

since we don't know which is bigger we say as $O(n+m)$

This can also be written as

$O(\max(N, m))$

There is no Additional space used so the space complexity is constant — $O(1)$

2)

```
int a = 0
for (i=0 ; i<n ; i++){
    for(j=N ; j>i ; j--){
        a = a + i+j
    }                    } O(n^2)
}
```

$n = 2$

$j = 2 \Rightarrow$

$O(n^2)$

3)

```
int i, j, k = 0
for(i = n/2 ; i<n ; i++){
    for(j=2 ; j<=n ; j=j*2){
        k = k+n/2
    }
}
```

$O(n/2)$

let $n = 8$

outer loop $i=0 \rightarrow 4, 5, 6, 7$

inner loop

$\hookrightarrow j=2 ;$ 4

2, 4, 8

$j=2 ;$ 5

2, 4, 8

$n = 16$

2, 4, 8, 16

$\hookrightarrow$ Inner loop $n = 2^k$ non linear

i.e $n = 4$ — 2

$n = 8$ — 3 steps

$n = 16 \rightarrow 4$ steps

$n = 32 \rightarrow 5$ steps

i.e $\log(n)$

$O(n/2) * O(\log(n))$

$\Rightarrow O(n * \log(n))$

4) int a = 0 ; i = N

while (i > 0) {

a += i ;

i = i/2

}

n = 8

↳ i = 8

loop i/2 . 8, 4, 2

n = 16 16, 18 ; 4, 2

$O(\log(n))$

5) for (var i = 0 ; i < n ; i++) {

i = i * k

}

let = k = 4 ; n = 8

i = 0

↳ i = 0 * 4 = 0

let K = 2, n = 8

i = 0

↳ i = i * k = 0

↳ i++ => 1

i = 1

↳ i = i * k = 2

↳ i++ => 2+1 = 3

i = 3

↳ i = i * k = 3 * 2 = 6

i++ = 6+1 = 7

i = 7

↳ i = i * k = 7 * 2 = 14

i++ = 14+1 = 15

n = 8, k = 2 → 4 steps

n = 16, k = 2 → 5

n = 32, k = 2 → 6

Approximation

$\log_k(n)$

i < n

=> 14 < 8 } Rate } Reverse log

Data Structure

→ A data structure is a particular way of organizing data in a computer so that it can be used effectively.

- Access (Retrieve)
- Modification (Update)

Data Structure

- **Primitive** (Predefined way of storing Data)
  - Integers
  - Float
  - Characters
  - Pointers
- **Non-primitive** (User-defined way of storing data)
  - **Linear Sequential**
    - Array
    - Stacks
    - Queue
    - Linked List
  - **Non-Linear Sequential (non-~~Sequential~~)**
    - Graphs
    - Trees

1) Array :-

Imp ★ Items are organized sequentially one after another in memory stack

Array
- → Static
- → Dynamic

Commonly used operations
- → Access
- → Search
- → Insert
- → Delete

} Complexities to remembering
- Avg case
- Worst case.

Access

↳ list = (1, 2, 3)

list (1) = 2 ⇒ O(1) complexity

↳ let first Element starts with 605 in Memory sources 1m

2nd element → 605 + 4 (bytes) ⇒ 609

3rd element → 609 + 4 ⇒ 613.

Pattern :- $\boxed{\text{Starting Address} + (\text{Index} * 4)}$ → 4 for 32 bit computer, 8 for 64 bit computer

→ So s/m can find any Element without wasting time so the time complexity for Accessing is '$O(1)$'

↳ since it has specific pattern → no matter how big the Array is it runs in const time.

2) Overwriting (update)

Say $x = \boxed{1 \mid 2 \mid 3}$ 600 605 604

0 1 2 of update

↳ starting memory Address + index * 4

$601 + (2 \times 4) = 609$ → no matter how big Array it runs in constant time i.e $O(1)$

3) ~~Searching~~ (Traversing)

↳ To visit Each and Every Element in Array we have use For loop / while

↳ And for loop time complexity is $O(n)$

↳ It is purely depends on the no. of Elements Hence it is linear.

4) ↳ Searching is Equivalent to Traversing

↳ Traversing (+) Comparision

if value == 10

↳ ~~continue~~ Break

↳ else

↳ continue

5) Copying

↳ Extensive operation → Because S/m has to traverse all the value, find new memory slot and store complete Array.

↳ If we are duplicating Array means we are taking double space.

↳ Time complexity ≃ O(n)

6) Insertion → O(n) (time complexity)

- → At Beginning
- → At middle
- → At End

Ex :- V = [100, 200, 400] (0, 1, 2)

↳ Let say we want to insert 300 at Index position 1

↳ i.e we have to shift all memory blocks after it And we have to add space for our new value

→ In this case we just needed to add one value that is "300" so we require four memory slot for our second position, Zeroth.

100	200	300	400
0	1	2	3

↳ The problem here we have not reserved Extra Memory Slot and there are chances that OS might have stored something & we can't shift 4 slots

↳ we face this problem for inserting in the middle but also at the begining & at the End.

↳ Behind the scene, when these situations happen, i.e when the available space is not Enough → then we copy complete Array to a new address → i.e it requires traversing, copying so if we generalize Time complexity is O(n)

↳ Most common for JS & python
no

Dynamic Array

↳ It can ~~chang~~ change size in c++ we have array list

→ Dynamic Array Allows us for faster Insertion.

Ex :- If we declare a static Array

V = [100, 200, 300) → 12 slots (4+4+4)

In case if we declare a dynamic Array.

↳ It will take 16 Memory slots.

↳ With dynamic Array it will always take Extra space

↳ This Extra space depends on "2" and it's Power

2^1 - 2
2^2 - 4
2^3 - 8
2^4 - 16
2^5 - 32
2^6 - 64
2^7 - 128

Statement

We depend on the smallest power of "2" that is Just greater than (>) Equal to the length of the Array.

↳ Example :- Suppose we have 3 Elements in our Array, we are going to reserve space for 4 Element

↳ 3 → smallest power of 2 is $2^2 = 4$

Ex :- suppose we have 5 Elements then we are going to reserve space for 8 Elements

i.e
3 - Elements — 4 slots
4 - Elements — 4 slots (As per statement >(or) =)
5 - Element — 8 slots

→ Because of this Extra Memory we can freely append Array to a certain limited size & once we Exhaust the freely available space then only our s/m is going to copy the whole Array but again provide the Extra space

In case of Dynamic Array

- for general case it is O(1)
- for worst case i.e O(n)
  - (If the free space Exhausted)

Deletion :-

- At Beginning O(n)
  At Middle O(n)
  At End O(1)

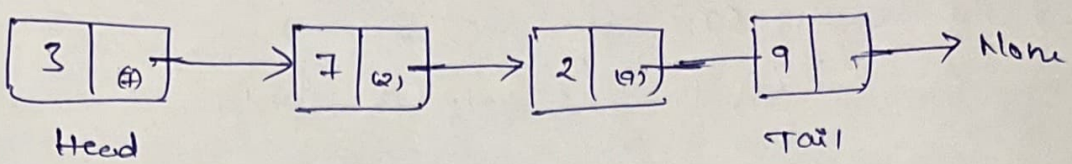
let say if we want to delete first Element then we have to access the Address of the first element delete it and shift all other Elements, ~~since the first~~ As it is Empty now. i.e it depends on no. of Elements we have ~~access~~ i.e directly depend on time.

→ If we generalise.

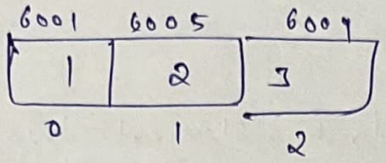
Access	Search	Insertion	Deletion
O(1)	O(n)	O(n)	O(n)

Linked List —

Ex:-



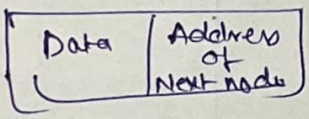
→ As we know condition of Array is store the values in sequence i.e if we know the address of the first value we just need to add 4 (for int type - 32 bit SYS)



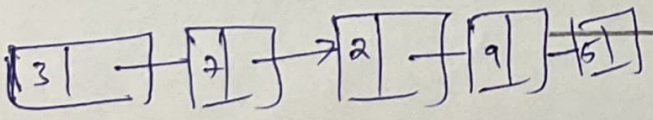
→ But in Link List this not going to be the case may be the first value can store at 6001 add second at 6015 address 3rd could be at 6045 Address, this is the major difference.

Representation → linked list represented in the form of node i.e

Node → Has two parts ~~one is Address~~



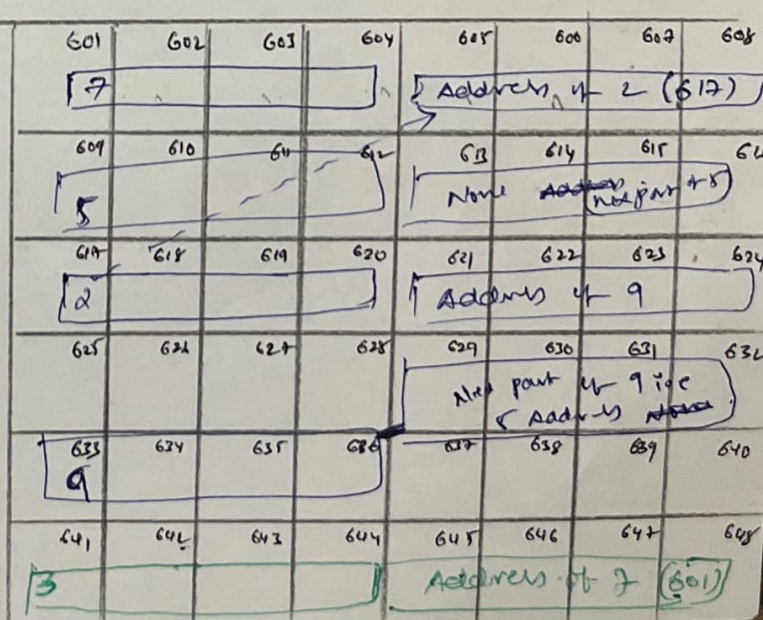
Linked List → there are some Elements that are linked Each other i.e by Pointer. (Address of next node)



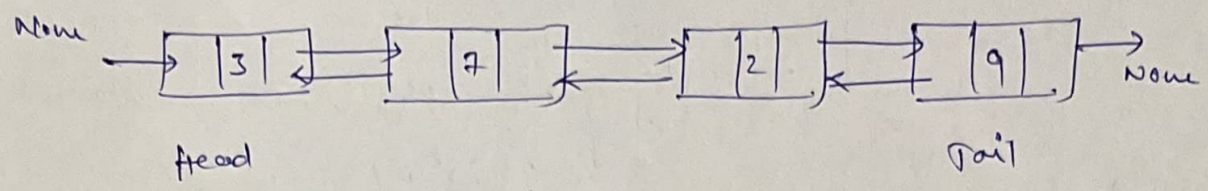
→ So here when we declare array it will take 8 spaces instead 4 (unlike Array)

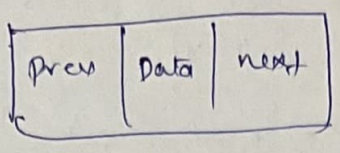
first 4 - spaces store data
Second - 4 spaces stores Address.

→ Since they store in Random block so there is no Index



Double linked list

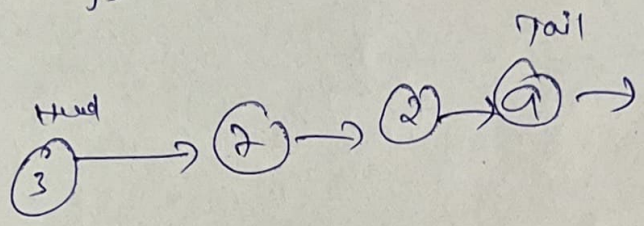




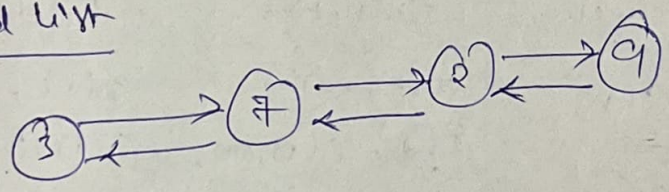
* Main difference b/w singly linked list and double linked list is, since double linked list has "prev" node reference we can go back in the traversal, but where as in single linked list won't possible as it has only next ref

Representation (2nd way)

Single linked list



Double linked list



Operations

↳ Access, Search, Insert, Delete

O(n) O(n)

Access

↳ To have access any element we need to traverse

~~Head to tai~~ Head → Tail

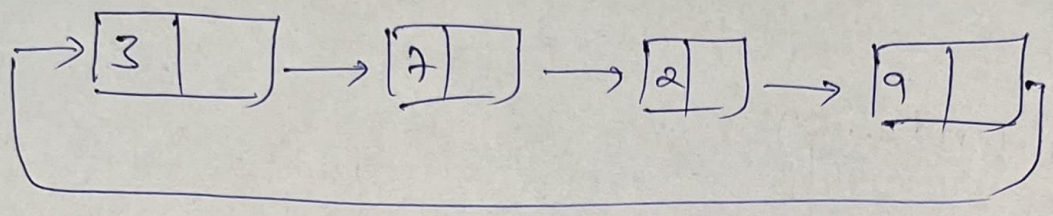
↳ O(n)

Dr. Pramod

→ Insert
- Start → O(1)
- End → O(1) → If we have info of Tail Node
- middle O(n) → else we need to traverse & then update.
  - → s/ly → O(1) → If we have info
  - → O(n)

- Deletion
- → start — O(1)
- End } O(1) & O(n)
- → middle } O(1) & O(n)
  - ↳ If we don't have info. of prev & next node.

Circular linked list



Access	Search	Insertion	Deletion
O(n)	O(n)	Start — O(1) End — O(1), O(n) s/ly middle — O(n) O(1) ↳ If we have info of tail node, else we need to traverse and update	O(1), — Start O(1) O(n) — middle O(1) O(n) — End

Stacks & Queues :-

1) **Stacks** → First in Last out (or) Last In - first out
   ↳ example is Stack of Books.

↳ Stacks is a List of elements that are in some Order, here we have our books in particular order which follow **LIFO** principal.

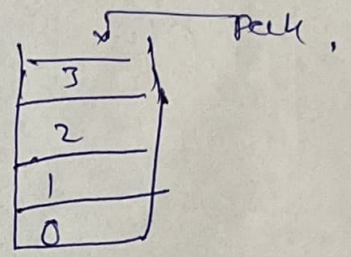
→ we can not touch the remaining items unless untill the above item pop out.

↳ Pop → Remove last time
Delete

Access → PEEK / TOP → returns the last item

Insert → Push → Add any item at the last position.

[0, 1, 2, 3]
↑
peek



→ Implementing Stack we use "Array"

2) **Queue** (FIFO → first in first out)

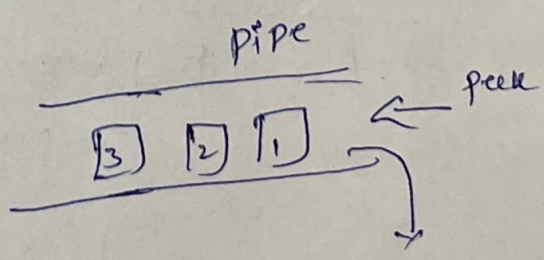
Delete ~~Pop~~ → dequeue → remove first element

Insert → enqueue → Add new element

Access → peek / TOP
↳ Returns first element

[0, 1, 2, 3]
↑ enqueue

Dequeue



peek

↳ for Implementing Queue we use Linked list

Stack & Queue Complexities

1) Stack

Becot Just we need to Free up last Block. ↑ From memory.

- → Pop ⇒ Delete → delete last Element → O(1)
- → ~~Access~~ Peek ⇒ Access → O(1)
- → Push ⇒ Insert → O(1), O(n)
  - → Since it is dynamic Array we have Extra Space Expect edge case (i.e Array Full)
  - → O(1) → For normal case
  - O(n) → For Edge case

Time Complexity

Pop	Peek	Push
O(1)	O(1)	O(1) O(n) → For Edge case

2) Queue

↳ If we Implement Queue with Array, it is going to cost Because → "Dequeue" means we need to remove Element from ~~last~~ first i.e we need to shift all the Memory blocks i.e O(n)

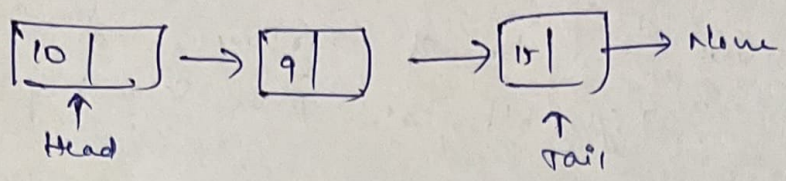
Queue with Array

↳ Dequeue → O(n)

Enqueue → O(1)
O(n) → For worst case

peek → O(1)

Queue with Linked list



Dequeue → Just remove Head → O(1)

Enqueue → Add Element to tail

↳ If we know the Tail → O(1)

Access → Accessing first Element from Linked List

↳ O(1)

HASH Table

Hash Tables :-

↳ The Main advantage of Hash table is → Search complexity → O(1)

↳ where in Array & Linked list is O(n)

→ Hash tables are built in in many programming languages

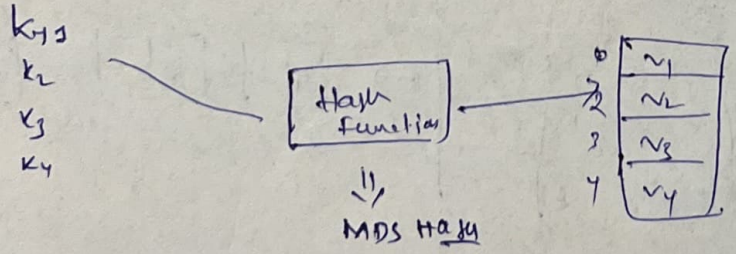
Python - Dictionary | Javascript - Objects

Java - maps | Ruby - Hashes

Ex :-

```
d = { 'key1' : 'value'
      'key2' : 'value2'
    }
```

Accessing ⇒ d[key]



→ Behind scenes

↳ When declare an dict (or) Hash Table, first our s/m is going to select the key & then it is going to get the into the hash function that is a magical function and then we get a Index position

→ ~~This index value can be anything, suppose we got "Zero", then we take key1 "Red"~~

→ This index value can be of anything and count, It count going to change if we give same key. (Any language)

↳ Colors = { "blue" : v_1
"red" : v_2
"Blue" : v_3
"Green" : v_4 }

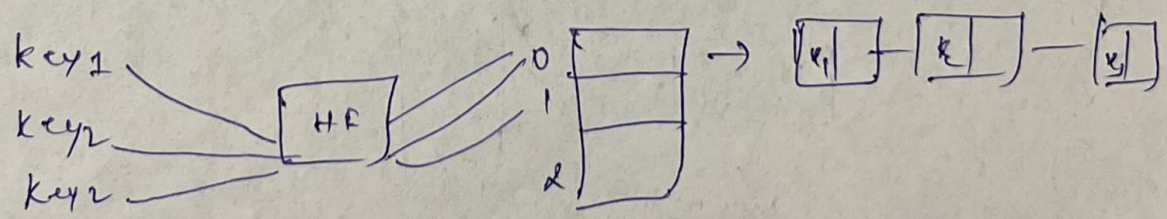
We no need to traverse the whole Hash table to get a value of a key.
⇓
When we access any key like Colors["blue"]
⇓
It is going to get the value directly by using the Hash function.

Own Hash Function

Let say we have 4 Elements in Hashtable The index is lies b/w 0, 1, 2, 3 → for 5 → 0, 1, 2, 3, 4.

Ex: blue → 98 + 108 + 117 + 101 ⇒ 424 % 4
⇒ 0 → Zero index

→ let say we defined a Hash function & Returning the index of the value stored in memory, for suppose say Two values pointing to same Index then this situation called as Hash collision



Then values in that Index will be stored as linked list
↳ then search complexity increases → O(n)
↳ This will be a rare situation ~~and~~ almost impossible If we use predefined s/m Hash functions.

Difference b/w Array and Hash Function

↳ When we work with array, we actually define our own index position i.e 0, 1, 2 etc.

→ When we work with Hash table, it is going to utilize the property of key & then define our index position

Hash table Complexities

1) Search / Access — $\boxed{O(1)}$

2) Insert

↳ Hash table is Exactly going to work as the dynamic Array → it means it reserves some Space. in terms of 2^{length}.

Ex:- Values = { "key1" : v_1, "key2" : v_2, "key3" : v_3 } for this It reserves 2^2 → 4 memory blocks.

↳ So it has 1 Extra slot

→ If we want to add new key more than 2^2 then the whole variable is going to replace it in the memory cannon.

↳ for Insertion

For normal case	$O(1)$
For Edge cases	$O(n)$

3) Deletion

↳ $O(1)$ → for general / Avg case

↳ $O(n)$ → for worst case In case of Hash Table Collision.

↓

As it need to work with linked list.

Example

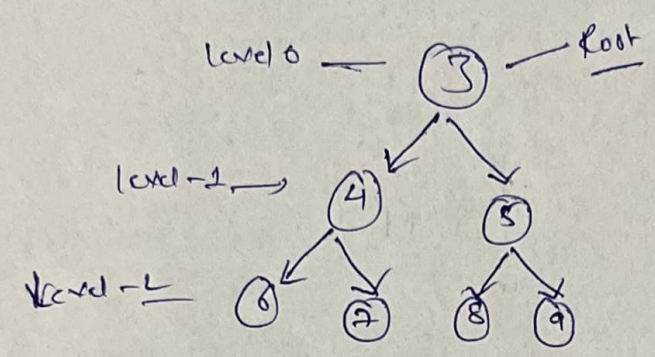
FB-comment

HTML - DOM

Trees → Often used to represent the Hierarchical Data.

Tree Important terminology

↳ Node, Root, Parent, child, leaf node, Siblings, Edge, levels, path.



→ 3 is parent of 4 & 5

→ 4 & 5 are children of Node 3

→ 4 & 5 are siblings.

Edge :- The connection b/w two nodes is edge, and it has direction. In trees we have only one direction i.e from parent to child.

→ Leaf node :- Any node which has no child are known as leaf node. Here 6, 7, 8, 9 are leaf nodes.

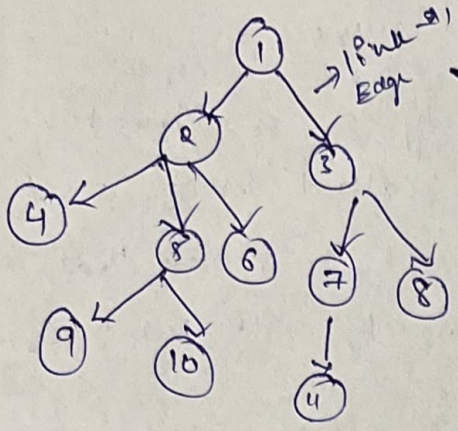
→ ~~Path~~

Definition :- Tree data structure can be defined as a collection of entities called nodes linked together to simulate a hierarchy

↳ Root → The top most node in the tree is called root of the tree

↳ Each node contain data and may contain link (or) reference to some other nodes that can be called its "Children"

Tree Vocabulary



Top most node without a parent

↑

node 1 → called as root node.

→ nodes 2 & 3 will be called "children" of node 1

→ node "1" will be called as "parent" of nodes 2 & 3

→ Children of parent called "Siblings" Here for Example 2 & 3 are siblings, (4, 5, 6). (7, 8) (9, 10)

→ Leaf nodes :- Node which has no children.

↳ Example ~~(9, 10) (11)~~ [4, 6, 8, 9, 10, 11] nodes.

→ Internal nodes :- All other nodes with at least one child can be called as Internal nodes. (or) Inner node

→ Grand parent :- Parent of parent

↳ Ex :- 1 is grand parent of 4 & 4 is grand child of 1

→ Link (or) Edge :- Connects two nodes to show that there is a relationship b/w them.

→ We can traverse Tree in one direction.

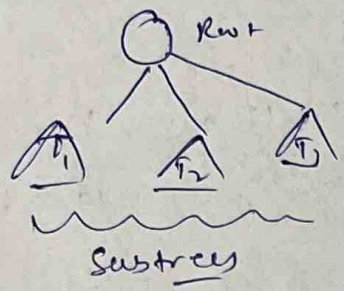
↳ 1, 2, 5 → are all Ancestors of 10 (And)

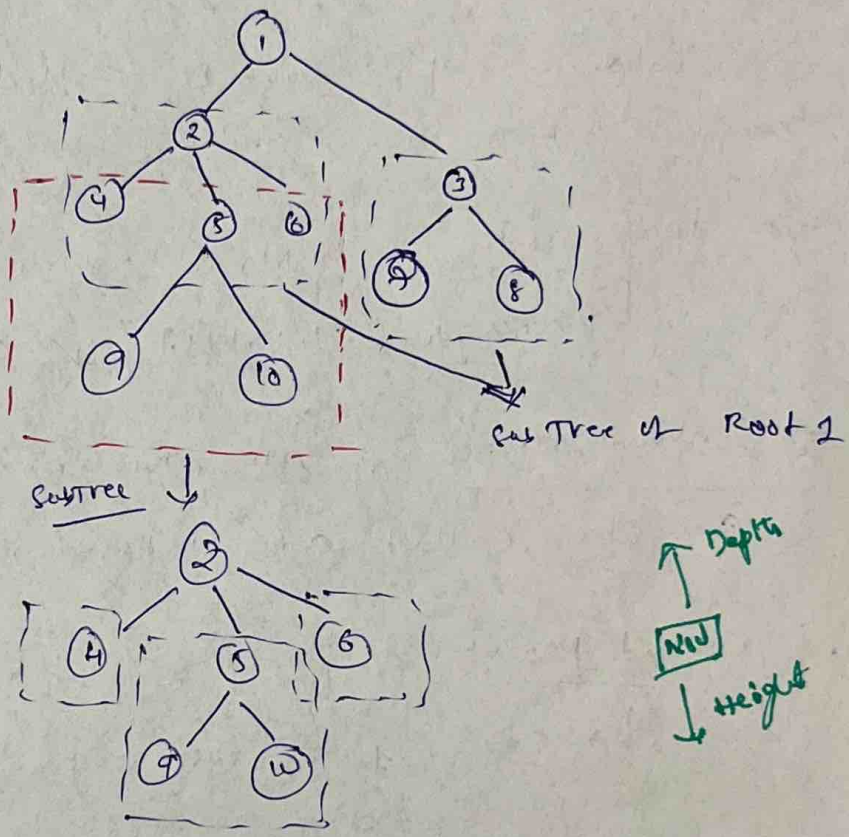
10 → is the descendent of all these nodes (1, 2, 5)

Properties of Trees :-

→ Trees are recursive data structure

Subtree ↳ We can define tree recursively as a structure that consists of a distinguished node called "root" and some "sub-tree" and the arrangement is such that root of the tree contains Link to roots of All sub trees.





* Tree with "N" nodes will has "n-1" Edges/Links

* **Depth** :-

↳ Depth of a node(x) = length of path from root to node(x) (8)

no. of edges in path from root to node(x)

→ Depth of Root node — Zero

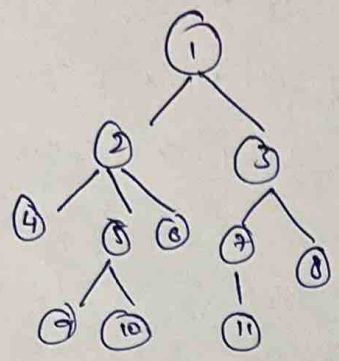
→ Depth of node (5) = 2 (i.e 1 → 2 → 5)

→ Depth of node (10) = 3 (i.e 1 → 2 → 5 → 10)

(Count Arrows)

* Height of Tree Node

Height of node (x) = no. of Edges in ~~larger~~ longest path from x to a leaf node.



⇒ Height of node (3) is → 2

Ex:- possible path to reach a leaf node (11) from 3 is

3 → 7 → 11 (2)
3 → 8 (1)
} max = 2

→ In the Above Example Height of root node is (3)

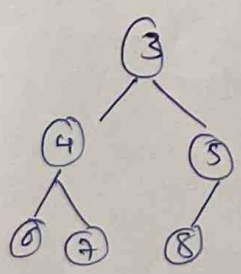
1 → 2 → 4 | 1 → (3) → 3 — 11
1 → 8 → 5 → 9 | etc.

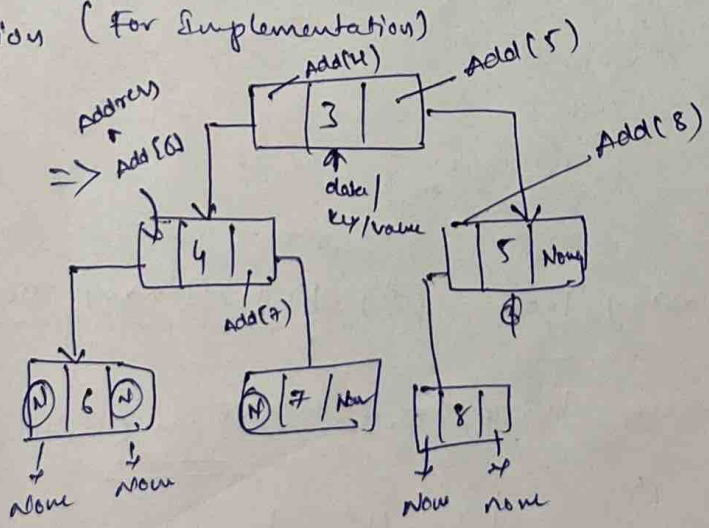
→ Height of Tree = Height of Root Node

↳ For the Above Example Height of Tree = (3)

* → Height & Depth are different → so for a given node Height & depth may (or) may not Equal.

Representation (For Implementation)





In double list we have prev & next
In Tree we will have left & right

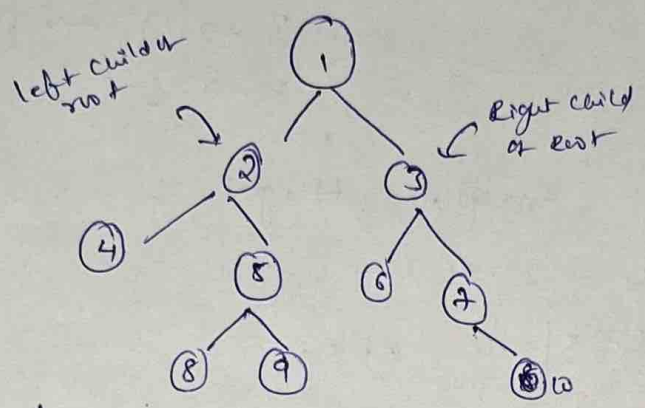
Binary Tree –
- → Full Binary Tree
- → Complete Binary Tree
- → Perfect Binary Tree
- → Balanced Binary Tree
- → Degenerated Binary Tree

* Binary - Tree

↳ Each node in a Tree can have at most 2 Children

↳ i.e A node can have, 0, 1 (or) max 2 Children

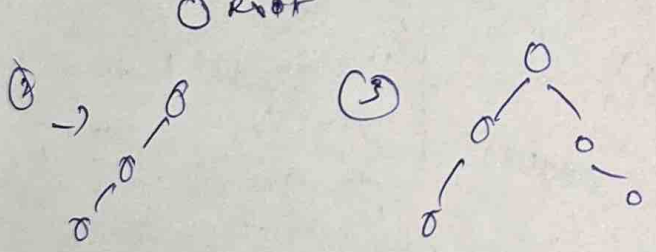
→ We call one of the child as left child & another as right child.



→ In Binary tree A node can have both left & right child (or) only Either right (or) left (or) node (for leaf node)

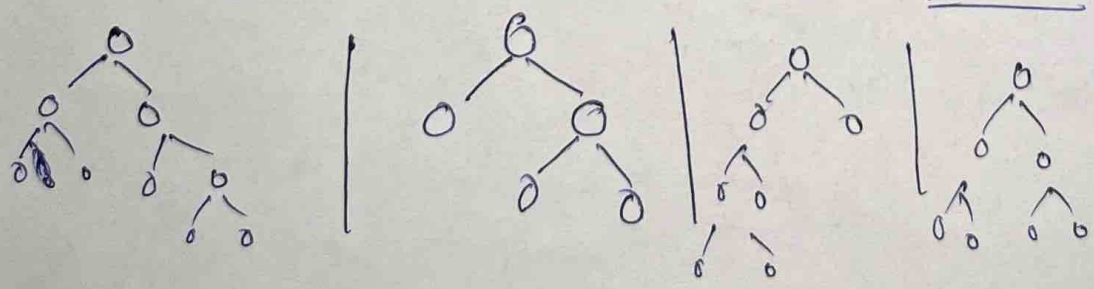
→ Different Binary Tree Structures

① ↳ A Tree with only root node is also a binary tree.



→ Strict / proper Binary Tree (or) Full Binary Tree

↳ Each node can have Either 2 (or) 0 children

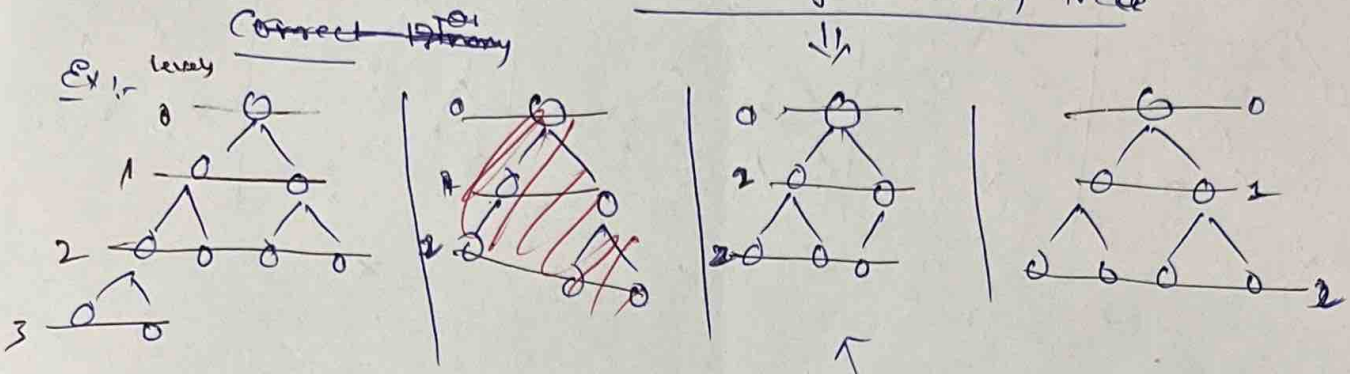


→ Say we have 0, 1, 2 levels then level 0, & level 1 should completely filled and level 2 can have empty ~~[illegible]~~ but all the nodes should be left aligned

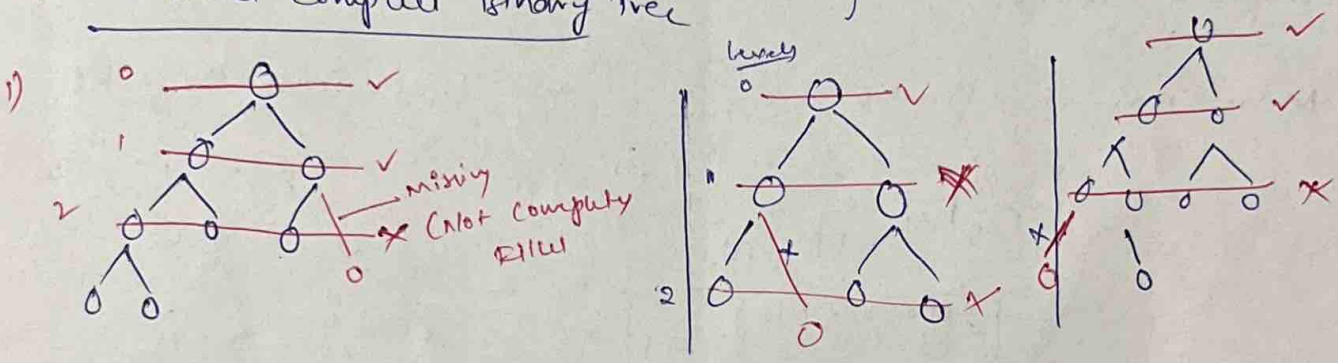
Complete Binary Tree :-

→ All levels completely filled with nodes expect the last level and in the last level, all the nodes as left side as possible.

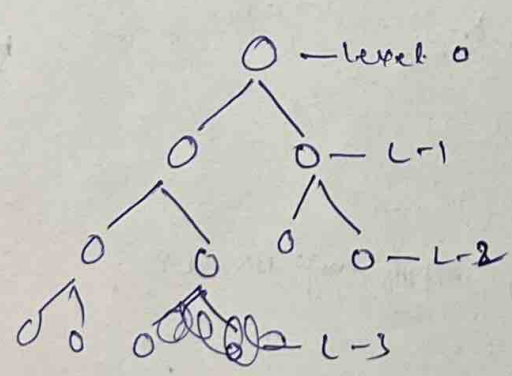
Ex :- levels



→ In valid Complete Binary Tree



* Max no. of nodes at level $i = 2^i$



Max

level-0 ⇒ $2^0 = 1$

level-1 ⇒ $2^1 = 2$

level-2 ⇒ $2^2 = 4$

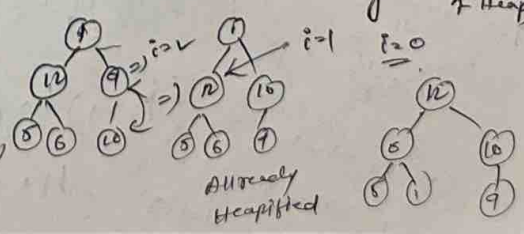
level-3 ⇒ $2^3 = 8$

* In case of a complete tree, the first index of a non-leaf node is given by $(n/2) - 1$; All other nodes after that are leaf-nodes → and thus no need to Heapify (In case of Heap)

Ex :- a = | 1 | 12 | 9 | 5 | 6 | 10 | (indexes 0, 1, 2, 3, 4, 5)

n = 6

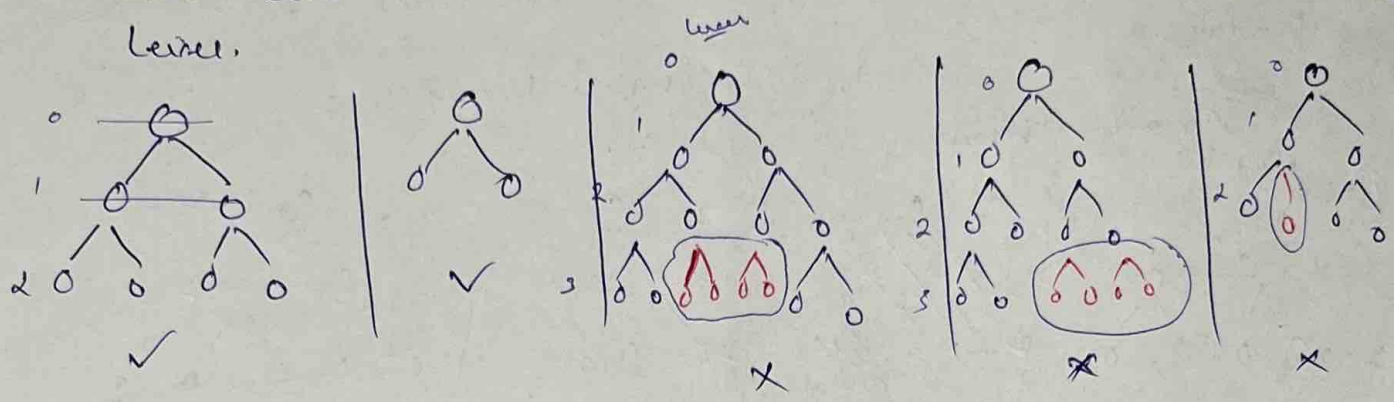
$i = 6/2 - 1 \Rightarrow 2$ (loop runs from 2 to 0)



→ let's say we have 2 levels (0, 1)
↳ all the elements should be completely filled

Perfect Binary Tree

↳ Binary tree in which all internal nodes have 2 children and all the leaf nodes are at the same depth (or) same level.



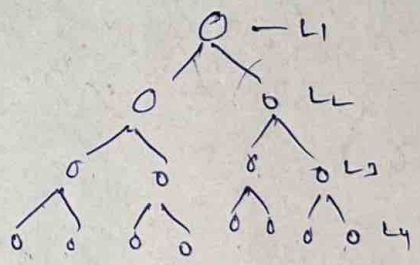
→ In simple terms all the node in a level should be completely filled.

*) Max no. of nodes in a binary tree with Height (h)

$= 2^0 + 2^1 + \dots + 2^h$

$= \boxed{2^{h+1} - 1}$ (or)

$= \boxed{2^{(\text{no. of levels})} - 1}$



$2^4 - 1 \Rightarrow 16 - 1 \Rightarrow 15$

↳ Height of tree = 3

→ Height of a perfect Binary tree with n nodes

i.e $n = 2^{h+1} - 1$

$(n+1) = 2^{h+1}$ ⇒ Apply log Base(2)

$\cancel{\log_2(n+1) = \log_2}$

$2^{h+1} = n+1$

$\log_2(2^{h+1}) = \log_2(n+1)$

$(h+1)\log_2 2 = \log_2(n+1)$

$h+1 = \log_2(n+1) \Rightarrow \boxed{h = \log_2(n+1) - 1}$

Rule: $(\log a^n = n \log a)$

$h = \log_2(n+1) - 1$

Let $n = 15$ (above example)

$h = \log_2(15+1) - 1$

$= \log_2 16 - 1$

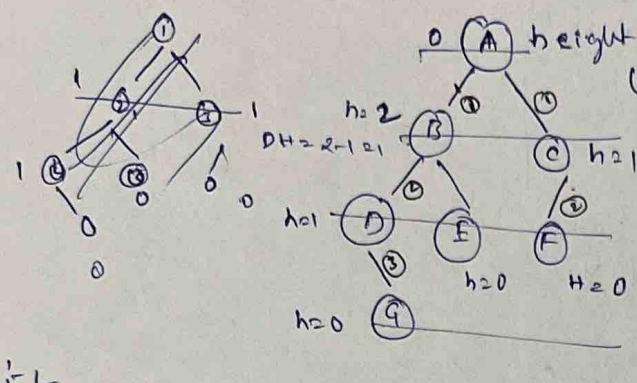
$= \log_2 2^4 - 1$

$4 \log_2 2 - 1$

$4 - 1 = 3$

Balanced Binary Tree :-

* Binary tree in which height of the left and right sub tree of every ~~binary~~ node may differ ~~by 1~~ at atmost 1 $\{-1, 0, 1\}$



height diff = height of left child − height of right child
(DH) = 3 − 1 ⇒ 2

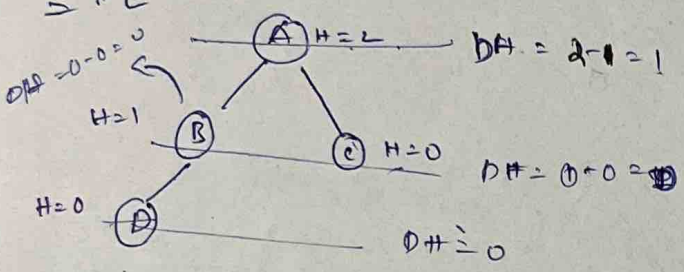
A → B → D → G, 3 edges

A → C → F, 2 edges

This difference is known as balance factor (BF)

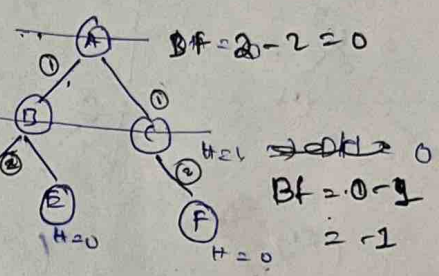
(✓)

Ex :- L



(✓)

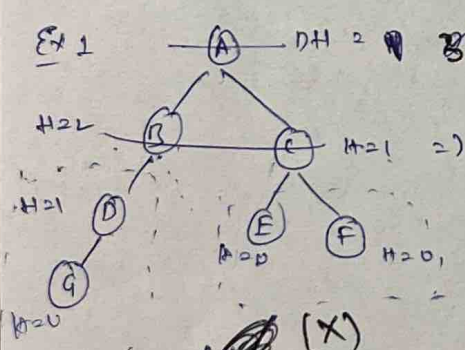
Ex :-



(✓)

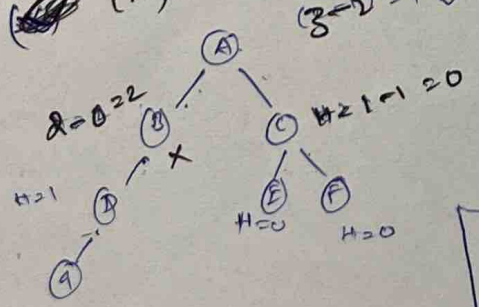
~~No Balanced Example~~

Ex 1



(✓)

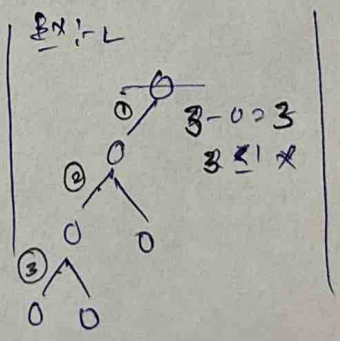
(X)



Node B → DH (1 − 0) = 1

Node C → DH ⇒ 0 − 0 = 0

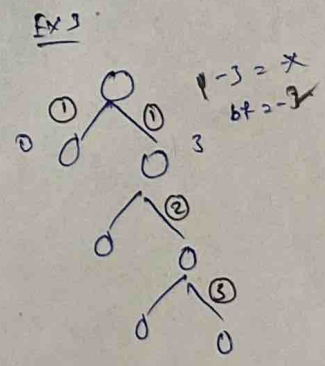
Ex :- L



(X)

(Non-Balanced)

Ex :



(X)

(Non-Balanced)

Important point

Balance factor (B.F) = height of left sub tree − height of Right sub tree

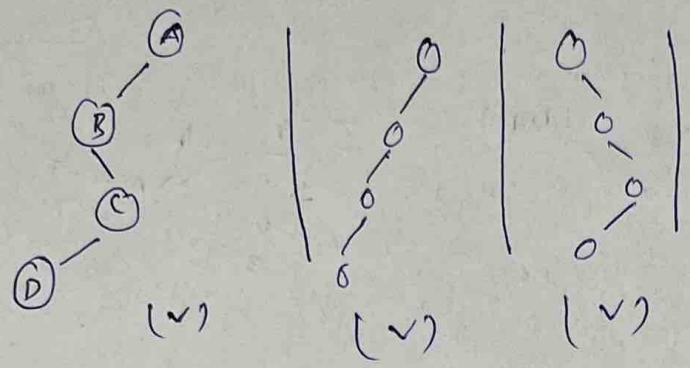
bf = $\{-1, 0, 1\}$ i.e

$|bf| = |h_L - h_R| \le 1$

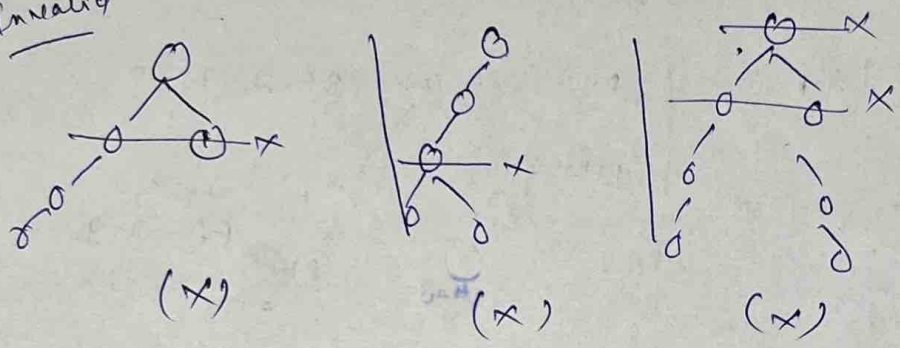
Degenerate Binary Tree

→ Binary tree where every parent node has only one child node.

Valid



Invalid



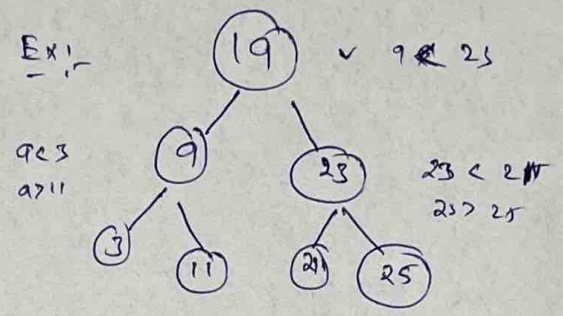
Binary - Search Tree (BST)

→ In BST all the elements in it's left are going to be Less than the root node value. And all the Element on it's right are going to be greater than the root node value.

→ ~~In short all the element on it's right are going to~~

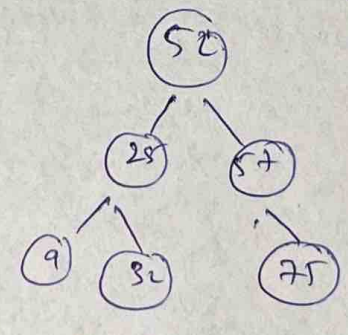
→ In short all the Elements in left subtree are going to be lesser & all the elements on right subtree are going to be greater.

Ex:-

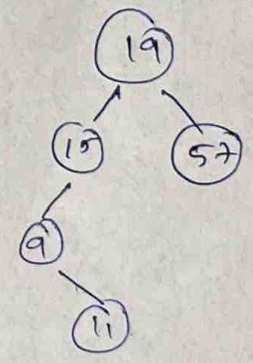


→ BST - Types

- → Balanced BST
- → Un Balanced BST



Balanced BST



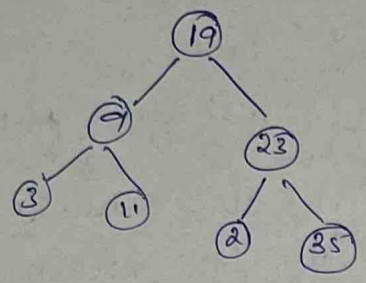
Unbalanced BST

⇒ for Balanced Binary the height is nearly Equivalent to $h = \log(n)$

→ For un balanced binary tree Height be n.

Balanced

BST - Searching → Search for "11"



19 > 11 → ~~False~~ True.

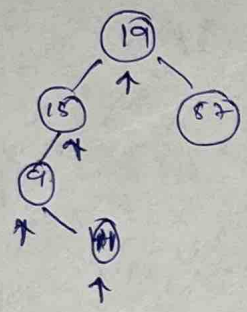
⇓ so neglect whole ~~left~~ Right subtree of the node (19) ↑ Root node

9 > 11 → ~~False~~ ~~True~~ False

↳ so neglect whole Right subtree of the node 9

↳ so Every Iteration we ~~halfpi~~ almost nullifying the half part so complexity will be ~~among~~ around $\log(n)$

→ Un-balanced BST

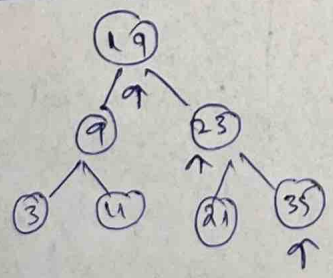


let say we don't 57 → i.e we almost visit all node i.e n

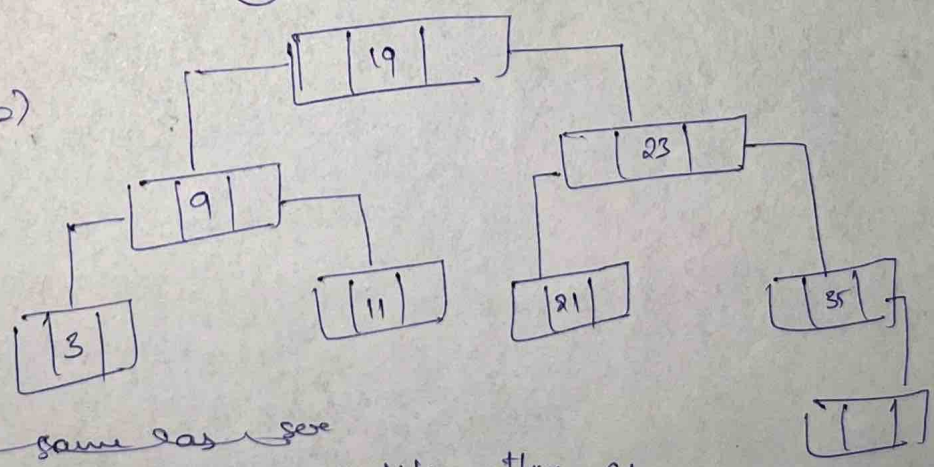
→ so for searching the BST

Best case/Avg case	→	$\log(n)$
Worst case	→	n.

→ Insertion BST :- Insert (37)



⇒



i.e → ~~for Traversing same as see~~

i.e → Traverse to get the right position then change the reference

→ And for adding node (37) and linking it is const time

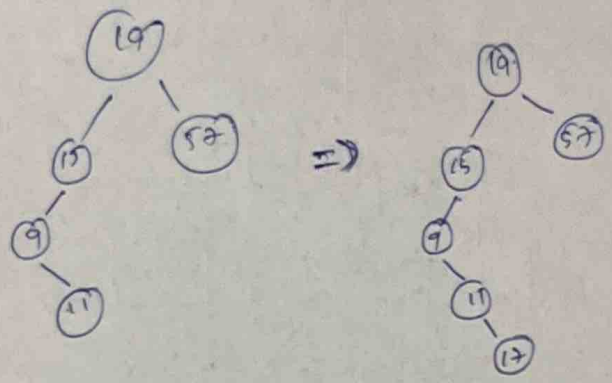
i.e $\log(n) + \cancel{c}$ ⇒ $\log(n)$

↳ Best / Avg case i.e for Balanced BST

~~Complete Binary tree~~

DS-19

(17) → Insert



→ n

Insertion Best case → log(n)

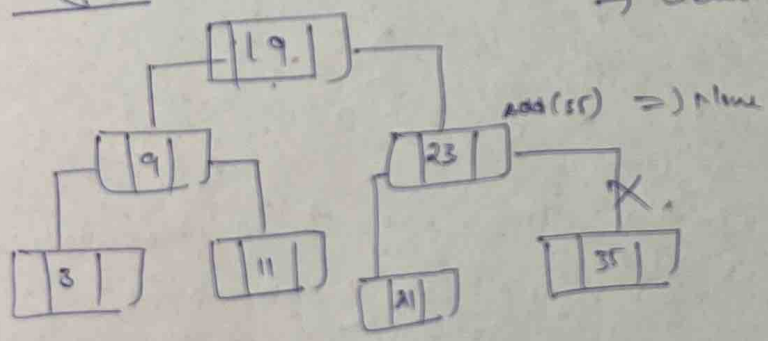
worst case → n

Delete (BST)

Node having
- → 0-Child → Deleting the leaf node → Traverse & Delete
  - → log(n)
- → 1 child
- → 2 Children

~~Node with 0 Child~~ Node with Zero Child

→ delete (35)



Traversing log(n) + const Time for Delete

⇓

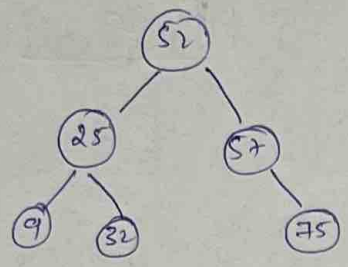
Time complexity O(log(n))

→ Worst case (un-balanced BST)

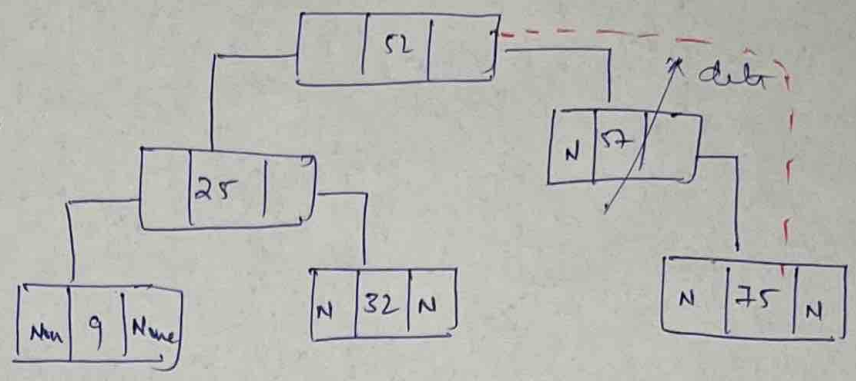
=> O(n)

Node with

1-Child → delete (57)



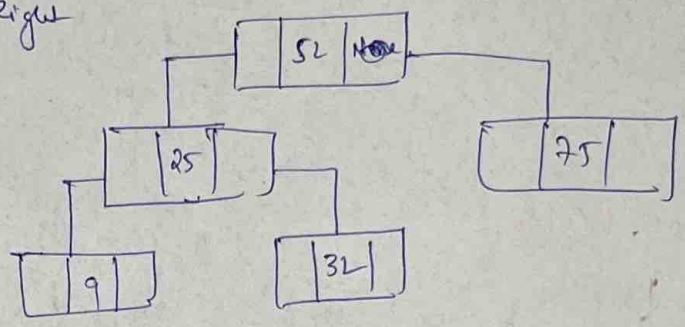
⇒



Here

Parent.Right = current.Right

⇓



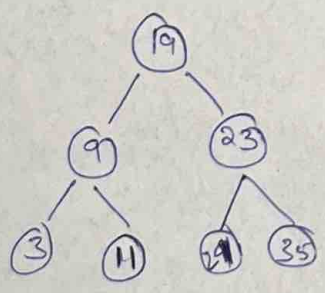
Best case / Avg case

↳ O(log(n))

Worst case — O(n)

~~2-Children~~

Node with 2-Children



→ let say we want to delete Root node (19)

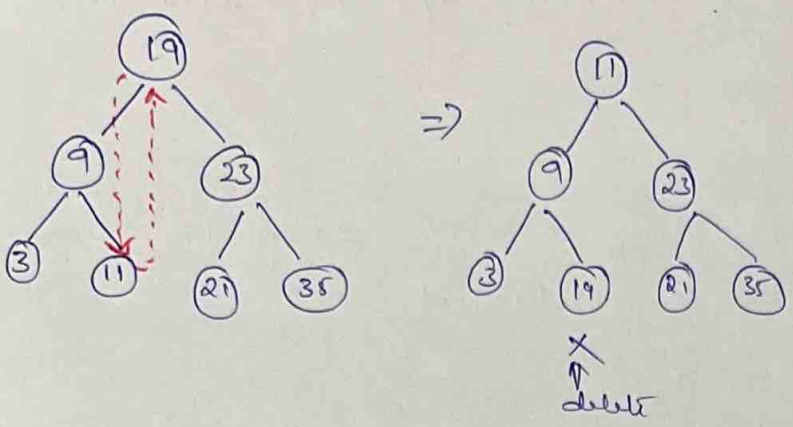
→ Then what will be Next Root

↳ either 11 (or) 23

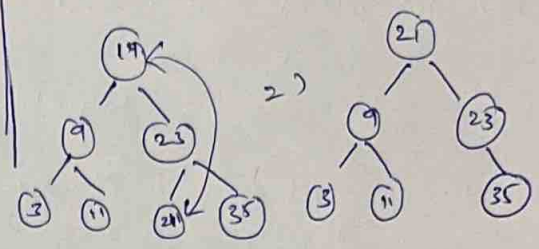
(or)
- ↳ i.e Highest value from left subtree
- ↳ lowest value from Right subtree

→ The easiest task is Highest value from ~~Right~~ left subtree

→ As we know deleting the Node with zero child is Easy → so do swapping first and then Delete the leaf node



If we choose lowest value from the right subtree



$\log(n) + \log(n) + O(1)$

- $\log(n)$ → Finding the node (19) → say if it not ~~found~~ root
- $\log(n)$ → finding leaf node i.e Here node (11)
- $O(1)$ → const time for deleting.

~~Ar~~ $\cancel{2}\log(n) + O(\cancel{1}) \simeq \log(n)$ Avg case

BST - complexity

	Searching	Insertion	Deletion
Avg (or) Best case	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$
~~Avg~~ worst case	$O(n)$	$O(n)$	$O(n)$

Deletion

→ search for node (n)
if n is a leaf
delete leaf (n)
else if n has 1 child
bypass node
↳ parent.Right = ~~C~~ current.Right
else replace (n) with successor

Adelson - Velskii Landis (AVL) Trees v/s Red black Trees

↳ Problem with BST

↳ While Inserting / deleting a node may be the Tree get's Balanced (or) unbalanced.

→ AVL

↳ AVL tree is a self balanced binary search tree (BST) where the difference b/w heights of left and Right Subtree cannot be more than one for all nodes

→ Why AVL ?

↳ Most of the BST operations (e.g. search, max, min, insert, delete, etc.) take O(h) Time where h is the height of BST

↳ The cost may become O(n) for unbalanced BST

→ So If we make sure that height of the tree remains O(log n) after Every Insertion & deletion then we can guarantee that the time complexity will be O(log n) for all cases.

→ Red - Black Tree

* A Red - Black tree is a kind of self - balanced binary search tree where each node has an Extra bit and that bit is often interpreted as the color. (red or black)

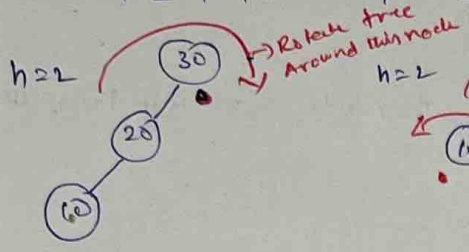
* These colors are used to ensure that the tree remains balanced during insertions and deletions.

→ Rotations Intro.

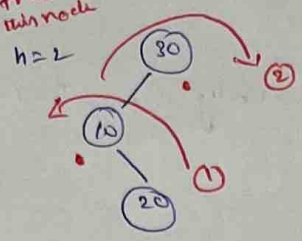
→ Say we have Key = 30, 20, 10

Insertion order

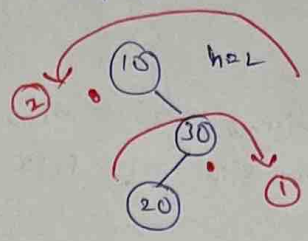
30, 20, 10



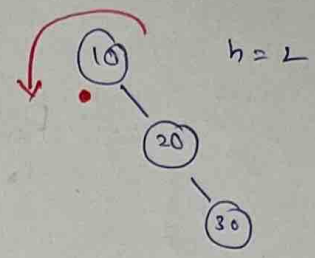
30, 10, 20



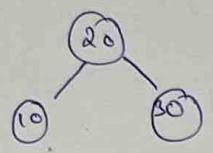
10, 30, 20



10, 20, 30

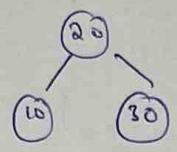


20, 10, 30



h=1

20, 30, 10



h=1

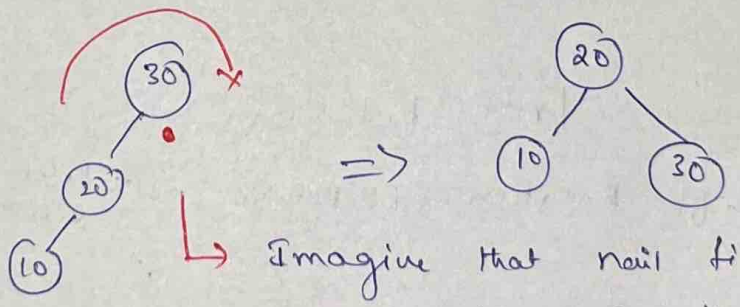
⇒ So for given keys we can have 3! (n factorial) ways of Insertion.
Here 3 key → 6 ways.

→ We always prefer minium hight binary search tree so that the search time can be reduced

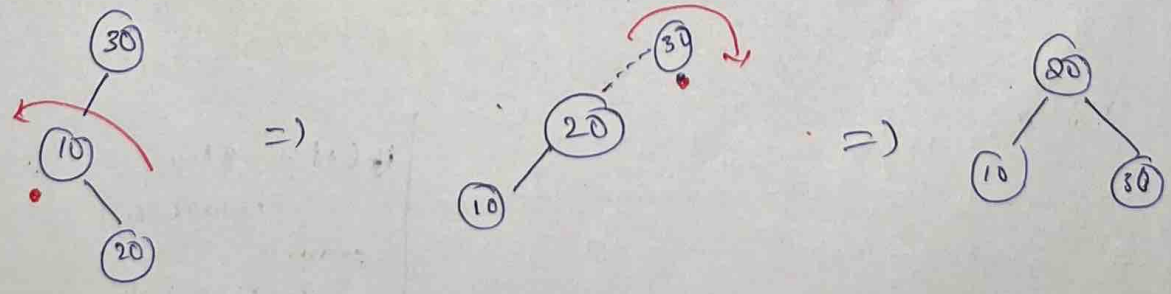
↳ So the draback of binary search is for same keys we get different shaped binary trees that can be of max ~~hight~~ hight (or) min hight → but we prefer min hight BST

→ So to achieve the min-hight BST we perform Rotations.

Ex ↳



↳ Imagine that nail fixed at the Red dot and around there is a thred and we pulling that thread.



→ Converting BST to balanced BST is the idea of AVL Tree.

→ We may get a doubt that we just have 3 nodes, What if we have many node in a BST. How Rotation Done?

so remember ↳ Rotations are done only on 3 nodes always irrespective to the size of BST

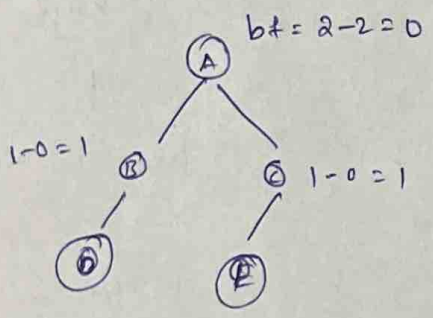
AVL - Trees

balance factor = height of left subtree − Height of right subtree

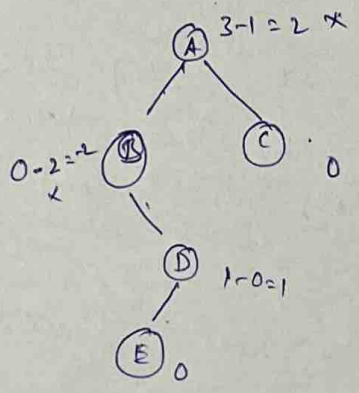
$$bf = hl - hr = \{-1, 0, 1\}$$

(or) bf should be either.

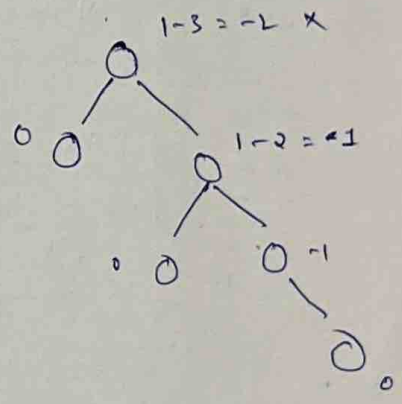
Absolute → $|bf| = |hl - hR| \leq 1$



Balanced



Un-balanced

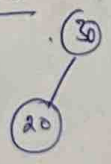


Un-balanced.

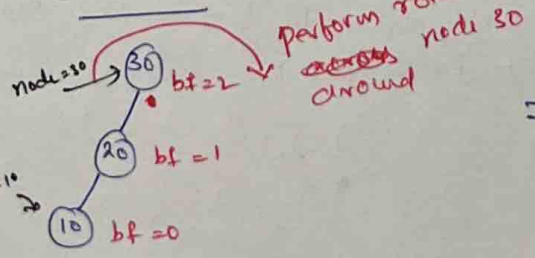
Tree Rotations

1) Left of left Rotation (LL Rotation) ⇒ i.e Rotate Right

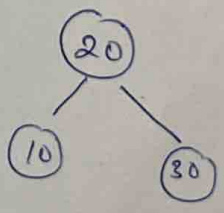
Initial



Insert 10



⇒



→ After Adding node "10" Tree became Unbalanced

~~10 < 30~~ 30 > 10 → False → go to left of 30

20 > 10 → False → go to left of 20

i.e Left - left Imbalanced

if (bf > 1 & key < node.left.key)

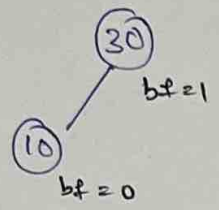
return

Rotate Right (node)

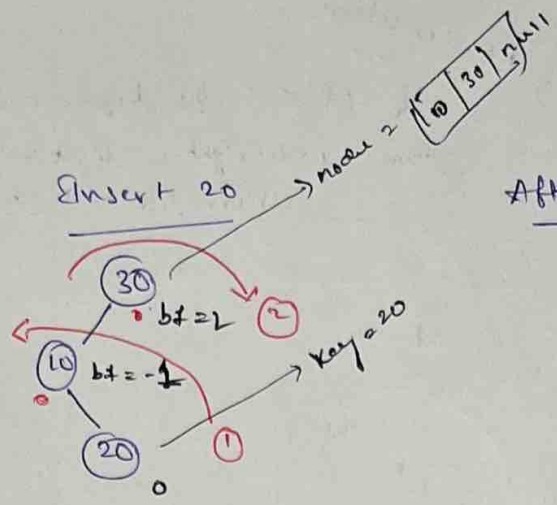
(pto)

2) LR - Rotation

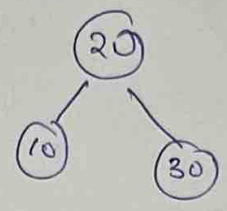
Initial tree



Insert 20



After Rotation



LR- imbalanced

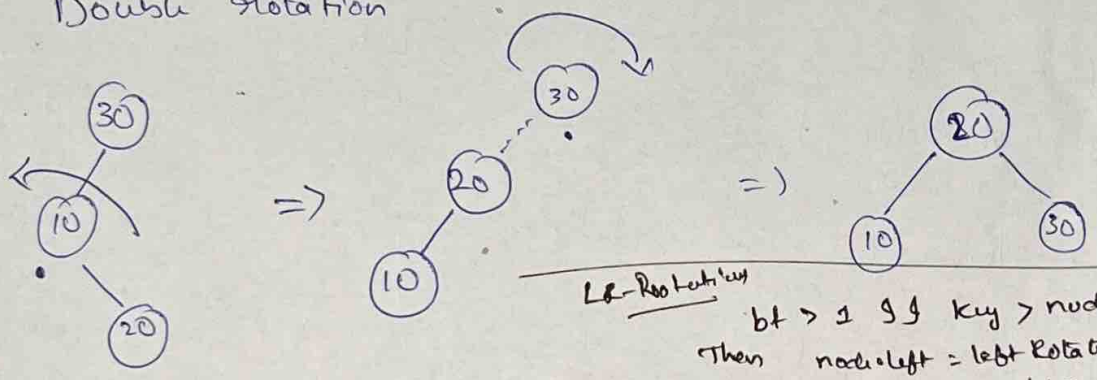
i.e 30 > 20 → False → go to Left (L) hand side of 30

20 > 10 → True → go to Right (R) hand side of 10

↳ ~~i.e. if at left~~ ↳ i.e first I traversed left & then Right

So LR - Imbalanced

→ In this case we need to perform 2 - step rotation i.e Double rotation



LR-Rotation condition

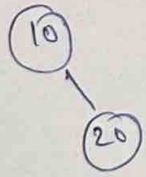
bf > 1 && key > node.left.key

Then node.left = left Rotate (node.left)

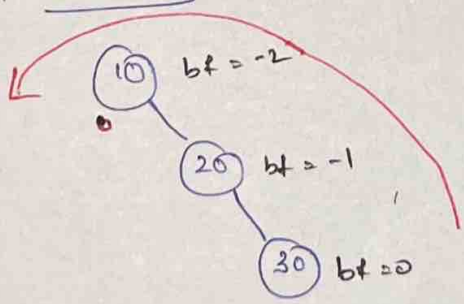
return right Rotate (node)

3) RR - Rotation :-

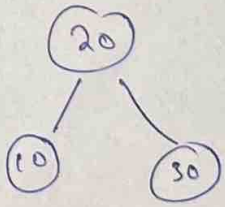
Initial tree



Insert 30



After Rotation



RR- imbalance

RR- Condition

if bf < -1 && key < node.right.key

Then ↳ left Rotate (node)

4) RL- Rotation

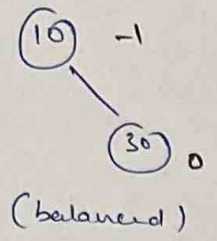
⇒ RL- condition

if bf < -1 && key < node.right.key

Then node.right = right rotate (node.right)

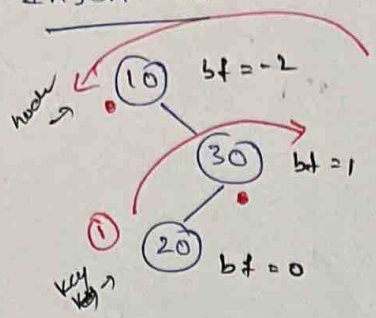
return left rotate (node)

Initial

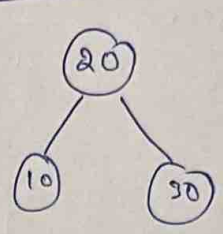


(balanced)

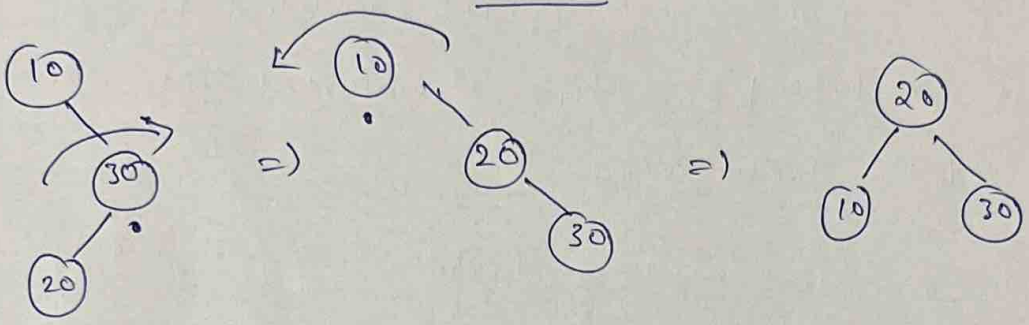
Insert 20



After Rotation



RL - imbalance



---

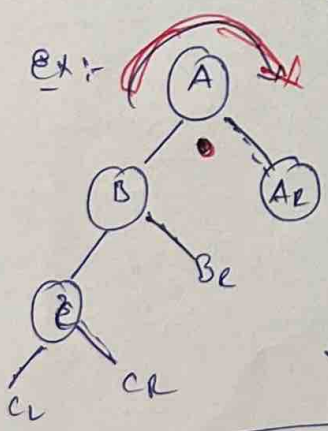
* LL - Rotation } single Rotation
RR - Rotation

LR } double Rotations
RL

* → Assume there is a big tree How To do you Perform rotation.

↳ Rule :- rotation can be performed only ~~on~~ on 3 nodes

Ex :-



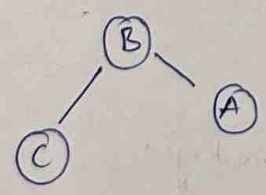
→ A_R → A Right Child with many sub tree.

→ Sly B_R, C_L, C_R.

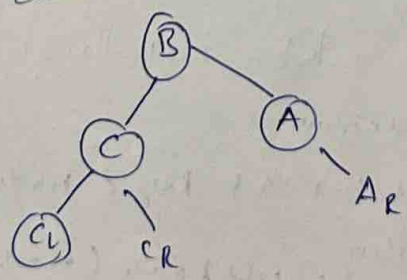
→ Assume that tree unbalanced at node A because insertion on left of left i.e LL - Rotation.

→ ~~So first~~ Don't look all node Just Look at A, B, C node and Rotate

---

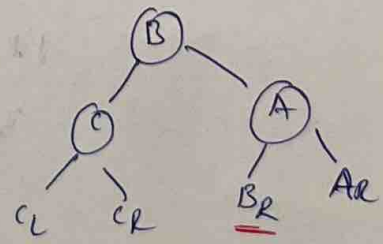
① 

→ Then Add it children



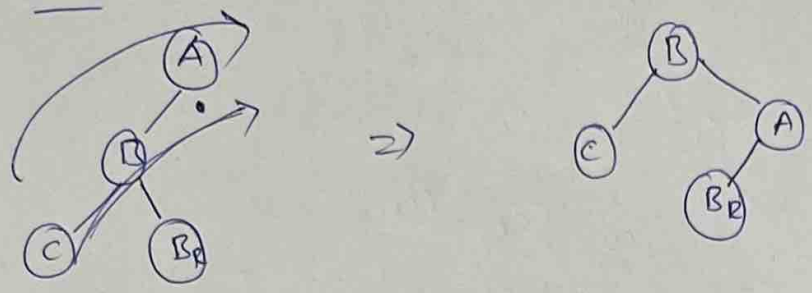
⇒ Remaining is (B_R) where the place remained it is at A - left side

⇒



DS- 20.3

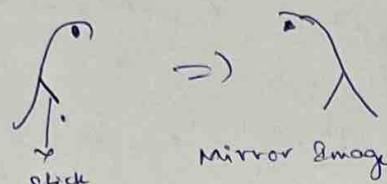
Hint :- LL - Rotation



LL Rotation

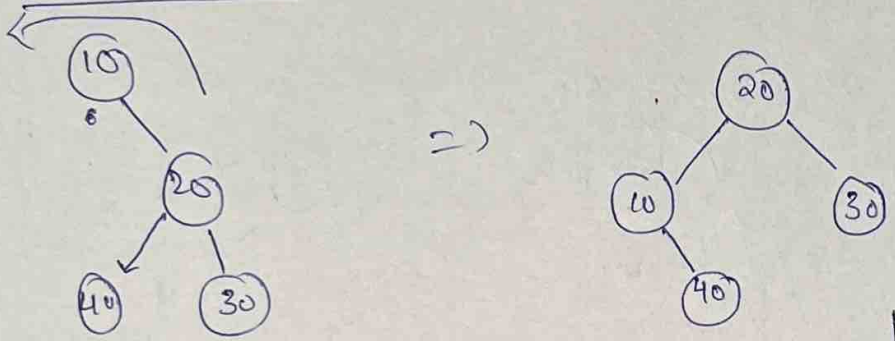
if (bf > 1 && key < node.left.key)
return rotateright(node)

Imagine there was a thread and pulled over the nail

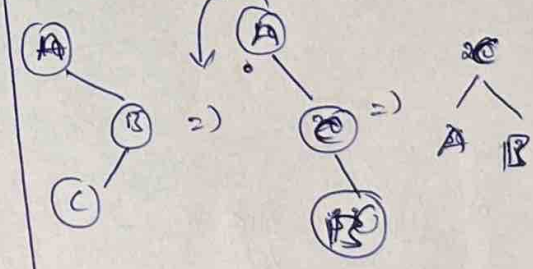


stick ⟶ This stick will appear left as the thread pulled

RR- Rotation (Hint)

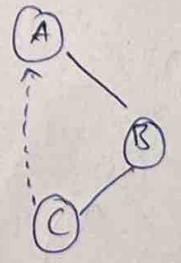


RL- Rotation

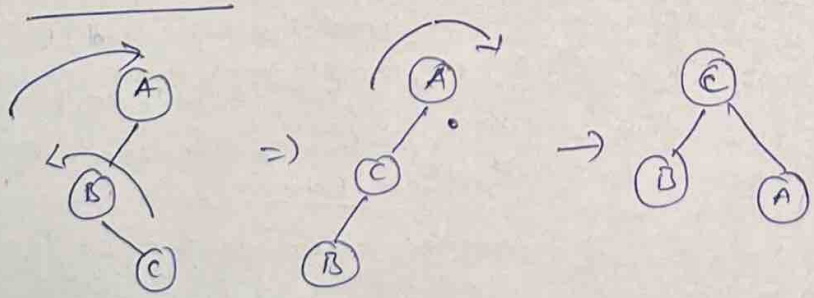


⇓

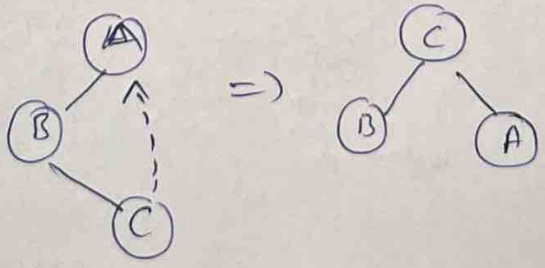
Directly in one step

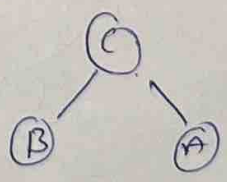


LR- Rotation

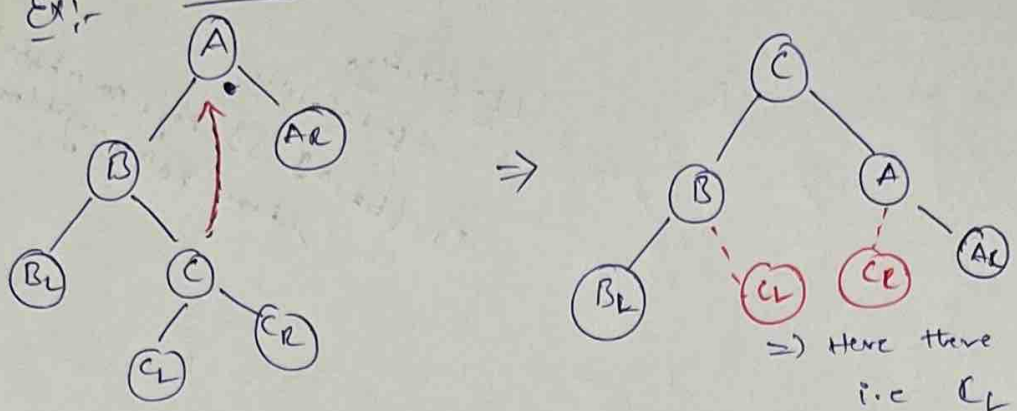


Single step



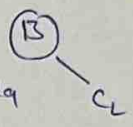


Ex:- LR-Rotation

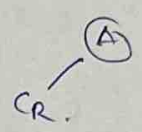


⇒ Here there are remaining 2 nodes i.e C_L & C_R

C_L ⇒ Initial place is left side so it be will at B → left



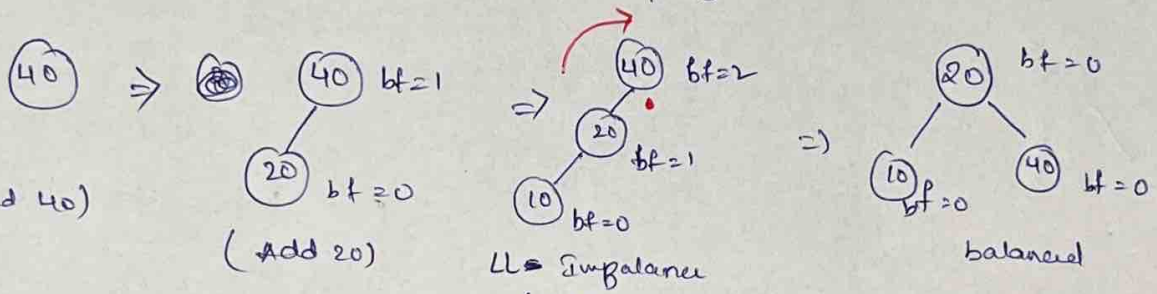
⇒ Similarly ⇒ C_R will pushed to A → Right side



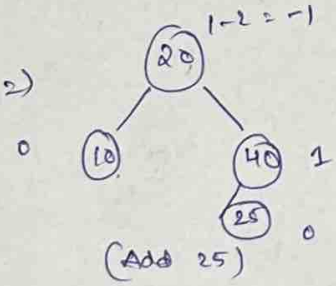
Examples (AVL - Generated)

Keys → 40, 20, 10, 25, 30, 22, 50

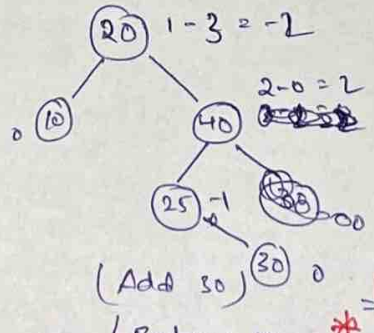
1)



(Add 40) (Add 20) LL Imbalance (Add 10) balanced



(Add 25) Balanced



(Add 30) (Balanced)

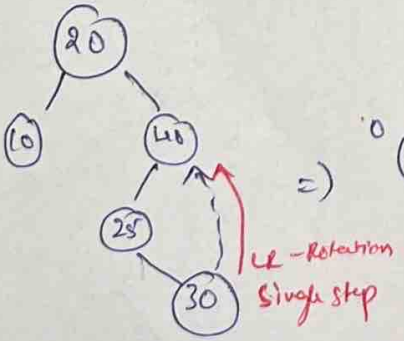
⇒ Here Two nodes got Imbalanced.

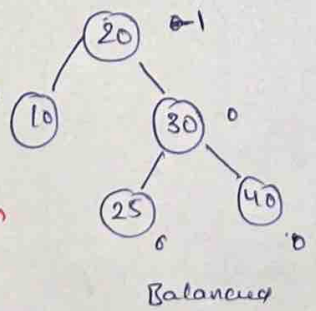
⇒ If the Multiple nodes get Imbalanced like this

⇒ Then we should perform rotation where you have Inserted

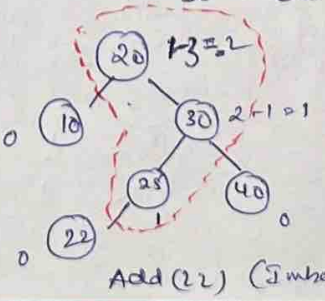
→ Here you Inserted L-R.

→ So - LR Imbalance.





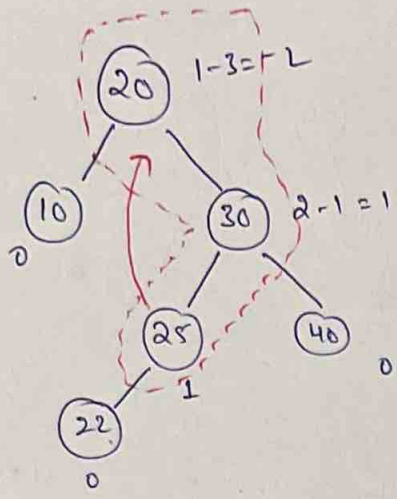
Balanced



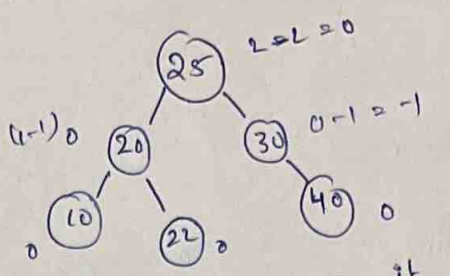
Add (22) (Imbalance)

⇒ RL ⇒ RL

Just take 2 steps



⇒



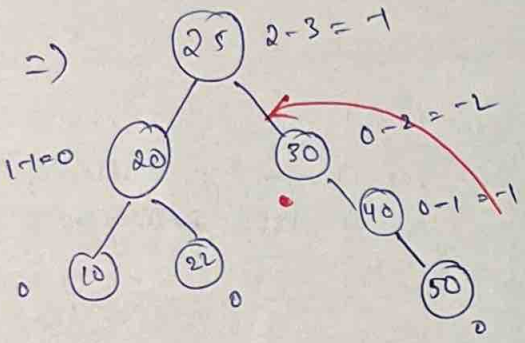
⇒ Balanced

↳ Previously it is left side of 25 so it still be left to 25 and the Remaining place is Right to (20) → 20 — 22

RL ⇕ - Imbalance

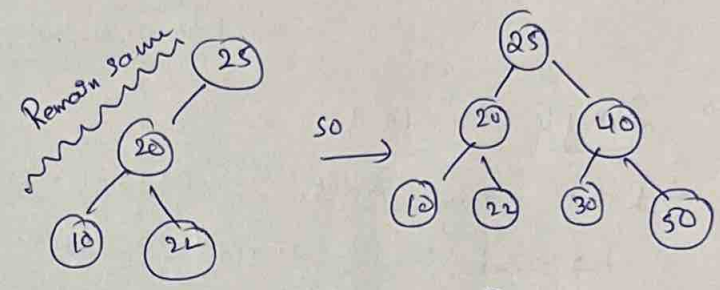
Just consider only 2 → RL - Imbalance

⇒



(Add 50)

RR - Imbalanced

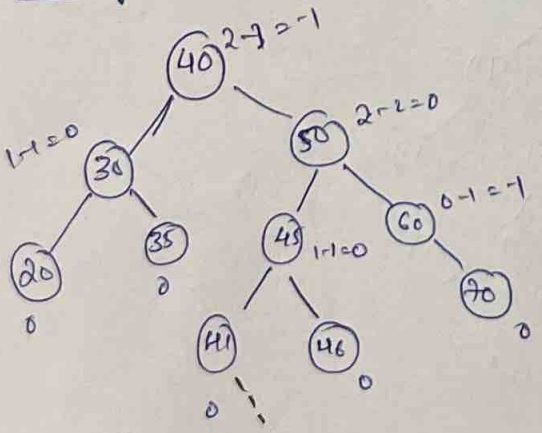


Balanced

* Don't allow any node to Exceed balance factor (B.F) from minus 2 (or) 2 you should never get -3 (or) 3

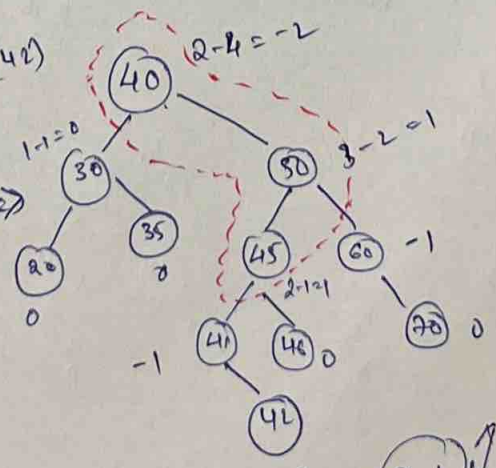
↳ While Adding / deleting - If tree getting Imbalanced then then perform balancing. Using rotation.

Example 3 :-



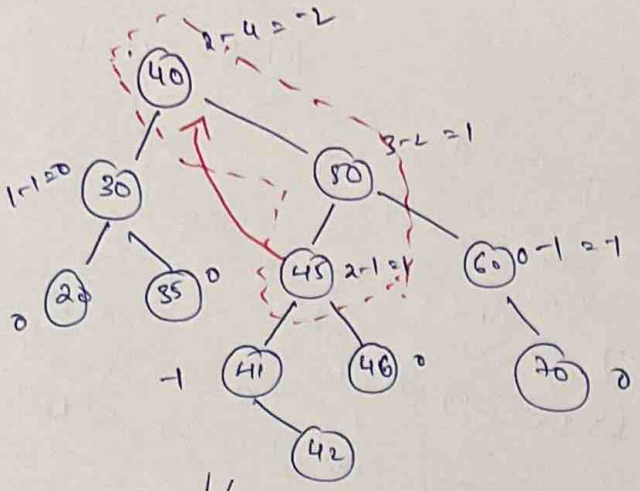
Add (42)

⇒ Add(42)

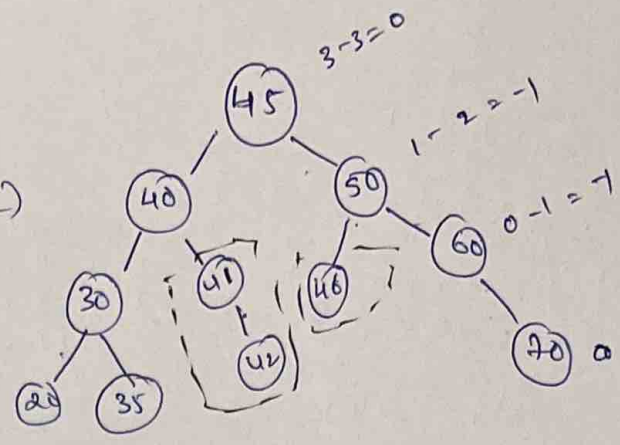


UnBalanced → RL ⇕

↳ consider only first 2



⇒



RL ↯ → Imbalance

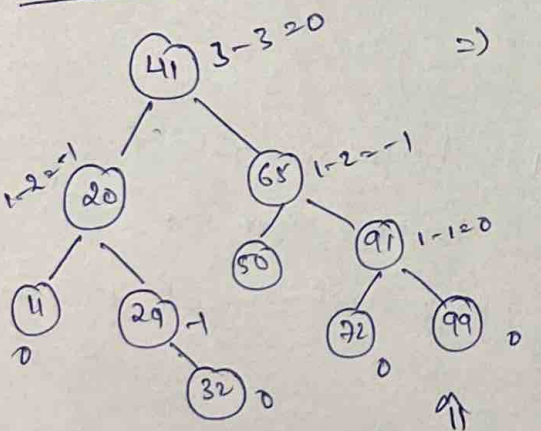
node 46 → It is Right to 45 Before swapping so there also it goes Right to 45 after swapping

S/y → Node 41, 42 it was there towards left now it is place towards left to node 45

→ Simple Funda :-

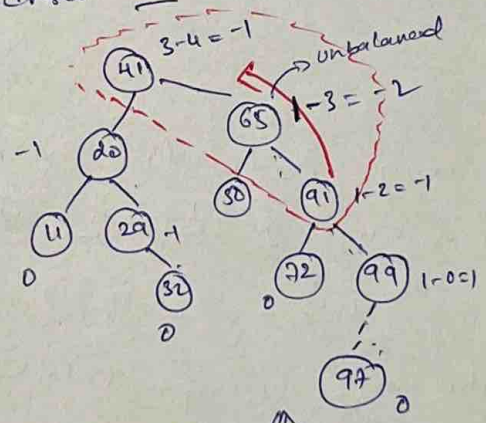
→ Rotate the node → Place all the unaltered nodes as it is.
↳ and the swapped nodes children → place it by following the general BST Insertion

Example 4 :-



⇒

Insert 97

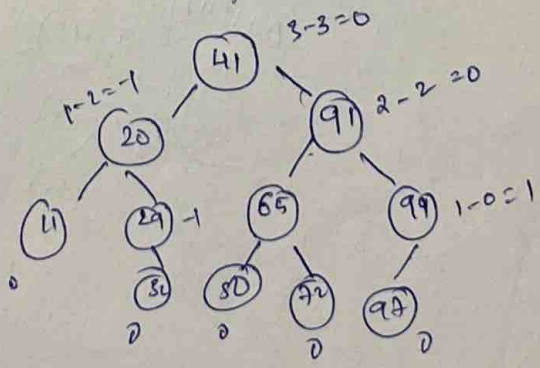


bf of root = 3 - 3 Balanced

LT (41) - (21) - (29) - (32)

RT (41) - (65) - (91) - (99)

RR ↯ L - imbalance



Balanced

Red black Tree (RB Tree)

* A R-B tree is a kind of self-balancing binary search tree where each node has an extra bit, and that bit is often interpreted as the colour (Red/Black)
* These colors are used to ensure that the tree remains balanced during insertion and deletions
* Rules that Every Red-black Tree follows.
  1) Every node has a colour either Red/Black
  2) Root of the tree is always black
  3) There are no adjacent "red" nodes (A Red node can not have a red parent (or) red child)
  4) Every path from Root to leaf has the same no. of Black nodes
  5) New Insertions are always Red.
  6) All leaf nodes (null nodes) are black nodes.
* If the Tree violates the above rules then we have to fix that tree.

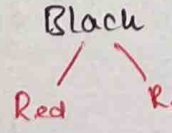
Rebalance

will be performed only 3 nodes

1 → If we have Black Aunt (or) uncle (BAR) Then → Rotate

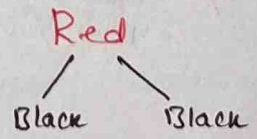
→ If we have Red Aunt (or) uncle → Color Flip of parent And Aunt/uncle

After Rotation



Parent Should be "Black"
And Child Should be Red

After Color Flip



Parent Should be Red and children should be Black.

I/p : 10, 20, 30, 9, 36, 56, 5, 12, 33, 1, 80

Rules

Black Aunt → Rotation

Red Aunt – Flip color.

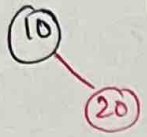
Rotation

Flip

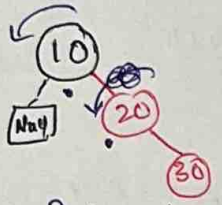
1) Insert 10 (root) => Always black

2) Insert 20 (Inserting node Always Red)

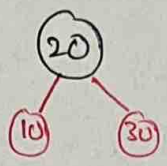
↳ Insert towards right size as 20>10



3) Insert 30



=>

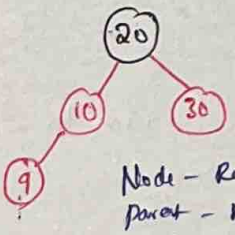


Red node, Red parent (voilation)
And it's Aunt is Black (i.e
There is no node means Null
all null as Black as per Rule 6)

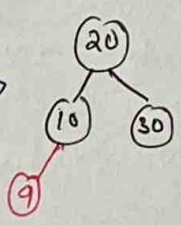
↳ Black-Aunt → Rotation (R R)

(When we Rotate
The parent will be
Black and it's
Children will be Red)

4) Insert 9

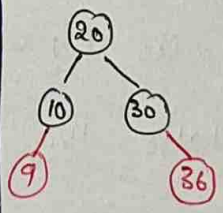


=>



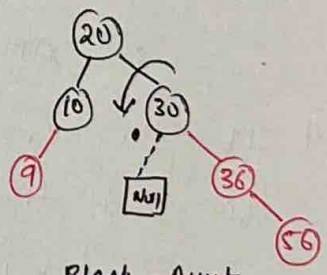
Node – Red
Parent – Red
Aunt – Red => Red Aunt
color flip

5) Insert 36

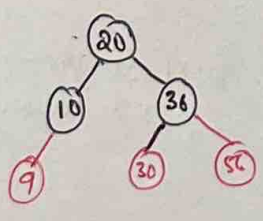


Balanced
No-Action required
~~Just plain~~

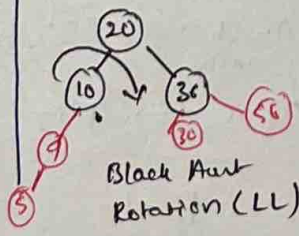
6) Insert 56



Black – Aunt
Rotation (RR)

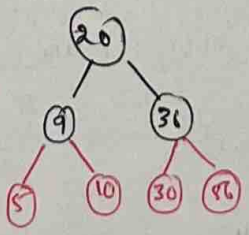


7) Insert 5

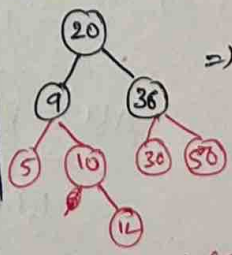


Black Aunt
Rotation (LL)

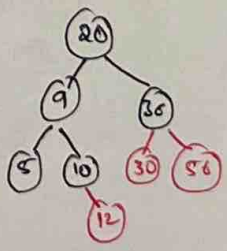
=>



8) Insert 12

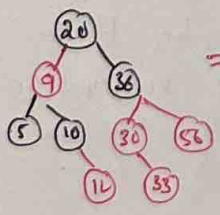


=>

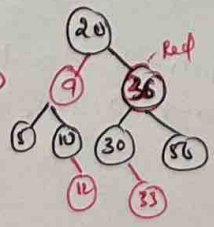


Red Aunt (5)
Flip colors
⇓
Push Black color from grand parent

9) Insert 33

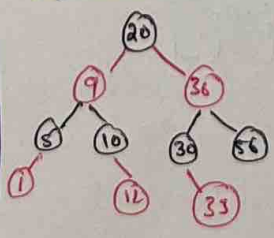


=>



Red Aunt (56)
Flip colors

10) Insert 1



AVL v/s RBT

* The AVL ~~tree~~ trees are more balanced compared to red black trees, but they may cause more rotation during insertions and deletion.
* So, If the Application involves frequent insertions and deletion then Red-Black tree is preferred.
* If the Insertion and deletions are less frequent & search is a more frequent operation then AVL should be preferred over RBT

Heaps (1

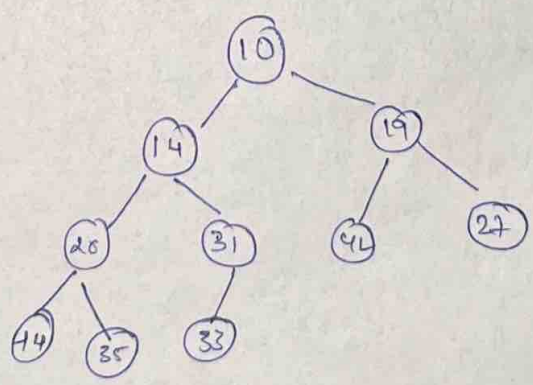
* A Heap is a special Tree-based data structure in which the tree is a complete binary tree.

* A Heap is a special case of balanced binary tree data structure where the root node key is compared with it's children and arranged accordingly.

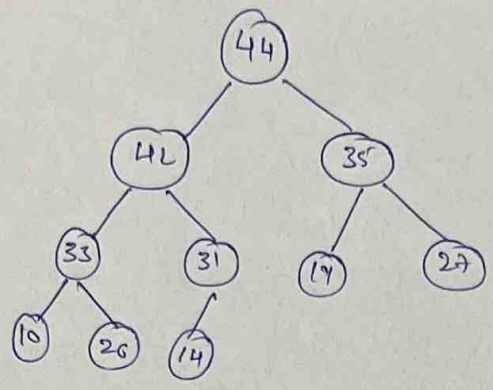
↳ As the value of parent is greater than that of child this property generates Max heap

Heaps → Max Heap
→ Min Heap.

Min-heap :- Where the value of the root node is less than (or) Equal to either of it's children.



MAX - Heap :- Where the value of the root node is greater than (or) Equal to either of it's Children.



~~Heap can be~~

→ Let's say we want to insert bigger element than the Root/Parent it should be at Right, Vise versa in case of small elements.

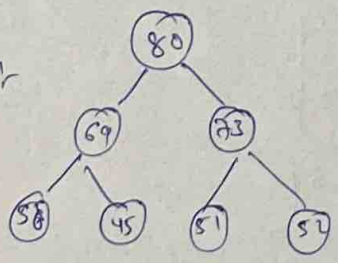
→ Where as in heap, If any thing lesser than root it Should be it's Child. It can be on left (or) right side, both of them are lesser than root.

→ For Heap we Follow Complete Binary tree, i.e If we are going to add any element we will be starting from left to Right.

↳ i.e for heap Height (h) = Log (n)

⇒ Heap can be implemented with the help of may be simple Array

↳ Ex:-



(Max - Heap)

⇒

0	1	2	3	4	5	6
80	69	73	56	45	51	52

Follow the level and Add elements from left to Right.

left Child → $2i+1$
Right Child → $2i+2$
} Observation

Ex:- $i=0$ ⇒ 80 (Root node)

↳ left Child ⇒ $2i+1$ ⇒ $(2\times 0)+1$ = 1 (Index) i.e 69 (value)

↳ Right Child ⇒ $2i+2$ ⇒ $(2\times 0)+2$ = 2 i.e 73

↳ Finding Parent

$i = i/2$

⇒ find parent of 45 ⇒ Index $i = 4$

$4/2 = 3/2 = 1.5$ ⇒ (floor(1.5)/2)

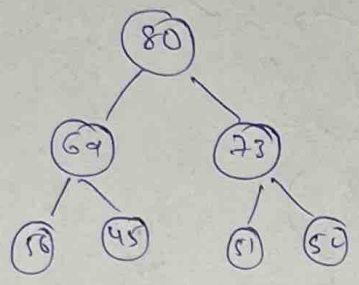
parent of 56 ⇒ $i = 3$ lower = 69

$3/2$ ⇒ $2/2$ ⇒ lower (floor) ⇒ 1

→ let's say we are working with MAX Heap
then array[0] → max value
for min heap array[0] → min value.

Heap Search

↳ don't use search and Heap together. because it will take the complexity O(n)

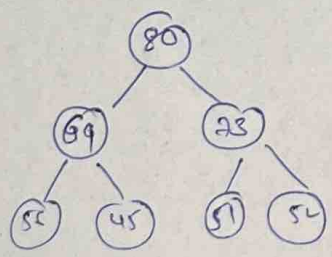


⇒ Search 52 → as 52 < 80 (Root) we just know it is going to be it's child ~~because~~.

visit ⇒ 80 → 69 → 56 → 45 (Not Found)
go back
80 → 73 → 51 → 52
go back

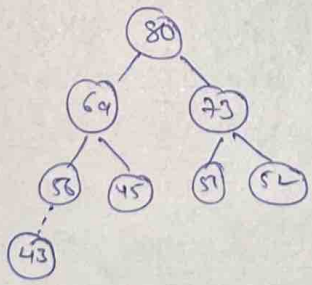
80	69	73	56	45	51	52

Insertion



→ Remember we follow complete Binary tree structure
↳ i.e we do insertion from left to Right.

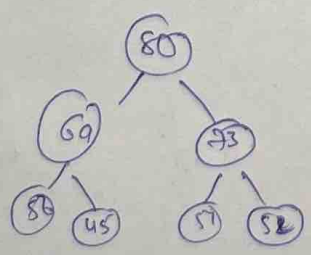
→ Case 1 :- Add smallest ~~Arrow~~ value than ~~from~~ the Root
i.e 43 → less than all Ancestors
43 < 80 < 69 < 56



0	1	2	3	4	5	6	7
80	69	73	56	45	51	52	43

i.e O(1)

→ Case 2 :- Add value greater than the root value
↳ Example Insert 85 → let's say we working with ~~max~~ Max heap. → i.e it should sit as a root node.



⇒

→ Easy way to Insert Element in case 2 is

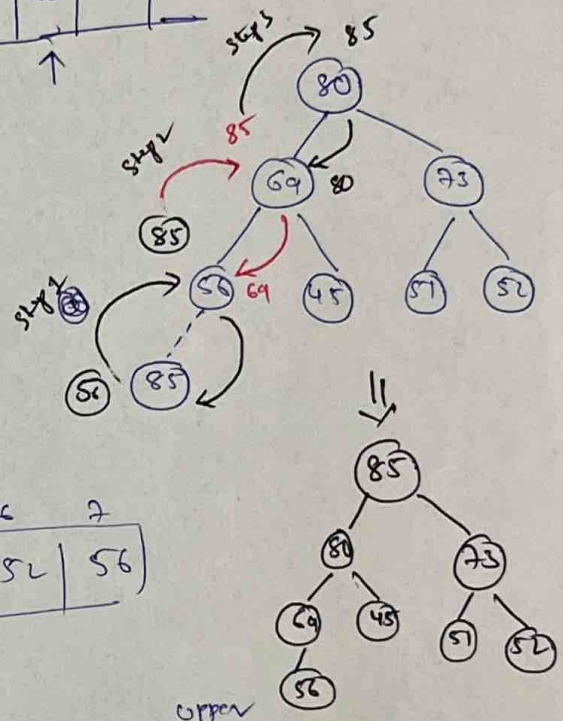
* ↳ Add at the last and do all the necessary swapping

→ Rule is

↳ ~~All the~~ Root will hold the Max value than Child.

Step 1 :- Insert at last

0	1	2	3	4	5	6	7
80	69	73	56	45	51	52	85



Step 2 :- Contact it's Parent

P_i = upper (i/2 - 1)

$P_i = 7/2 - 1 = 3.5 - 1 = 2.5 \Rightarrow 3$ upper value

$P_i = 3 \Rightarrow$ i.e 56

85 > 56 (True)

Then swap

0	1	2	3	4	5	6	7
80	69	73	85	45	51	52	56

→ Now 85 is at Index 3

$P_i \Rightarrow 3/2 - 1 \Rightarrow 1.5 - 1 \Rightarrow 0.5 \Rightarrow 1$ upper

$P_i \Rightarrow 1 \Rightarrow$ array[1] $\Rightarrow$ 69

85 > 69 $\Rightarrow$ True $\Rightarrow$ swap.

0	1	2	3	4	5	6	7
80	85	73	69	45	51	52	56

→ Now 85 is at index position $\Rightarrow$ 1

$P_i = 1/2 - 1 = 0.5 - 1 \Rightarrow -0.5 \simeq 0$ upper

$P_i = 0 \Rightarrow$ array[0] $\Rightarrow$ 80

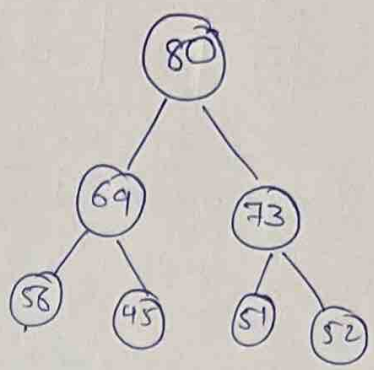
85 > 80 $\Rightarrow$ swap.

85	80	73	69	45	51	52	56

→ If we opsource Insertion is dependent on hight of tree

i.e $h = \log(n)$ $\Rightarrow$ so complexity is $\boxed{O(\log(n))}$

Heap deletion



→ Case 1 :- ~~dele~~ delete last Element 52

i.e O(1) const time

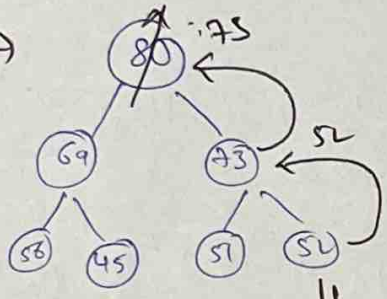
→ Case 2 :- delete 51

80	69	73	56	45	51	52

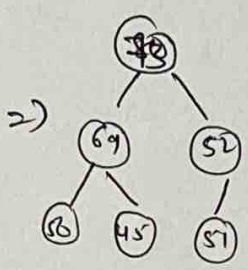
↳ If we delete 51 it is totally effect our Concept of binary Heap, which is tree should be complete Binary tree (last level nodes should be at left side)

↳ let say I want to delete the root node (80) and I added 73 as my new root Element

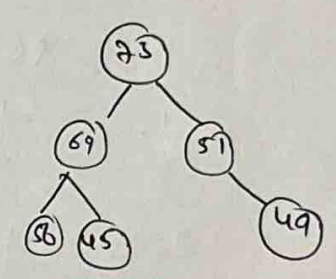
→



⇒



⇒



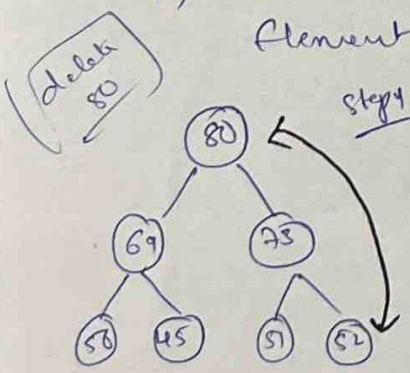
↳ What if the value is 49 instead 52

↑ Invalid Heap Structure As the last Element is at Right side.

→ To overcome the above problemes (case 2) we will be following this method

↳ So what we know is we can delete the last element which wont distrub the complete BT Heap structure

step 1 ↳ So we can swap Root & Last Element

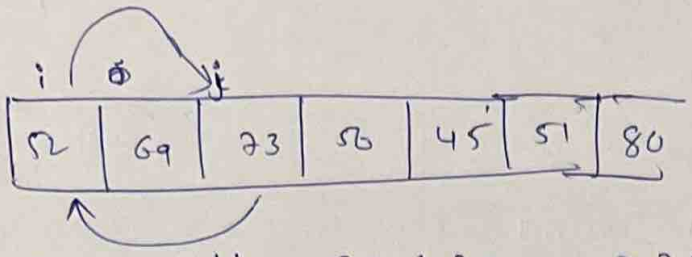


52	69	73	56	45	51	80

→ This was Messed because 52 < 69
52 < 73

→ So we compare both of the childs with Root (52)

0	1	2	3	4	5	6
80	69	73	56	45	51	52

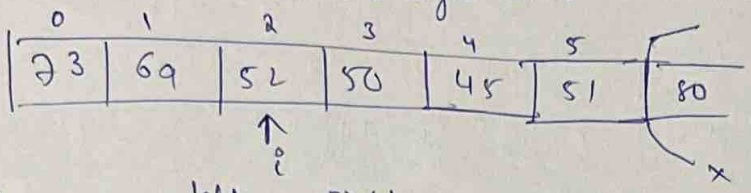


↳ compare child 69 & 73 → 73 is greater. so swap 73 with 52

52 → index ⇒ 0

↳ left child → $2i+1 \Rightarrow (2\times0)+1 = 1$

↳ Right Child → $2i+2 \Rightarrow (2\times0)+2 = 2$



$i = 2$ left $\frac{}{5}$ Right $\frac{}{6}$

↳ There is no element at index 6, so Compare the element with value at index 5

i.e $P_2 = 52$

$P_5 = 51$ $52 > 51$ ✓ (so no swapping)

→ Here 2 swaps

80 → 52

52 → 73

So the complexity is $O(\log(n))$

↑ To remove one Element

↳ To Remove all Elements

$$\boxed{O(n \log(n))}$$

Heap complexities

Search	Insert	Delete
$O(n)$	$\log(n)$	$\log(n)$

Heap Program (Max Heap)

Parent position is $\left(\frac{i-1}{2}\right) \Rightarrow i \Rightarrow$ Child position.

left $\rightarrow 2i+1$
Right $\rightarrow 2i+2$ } i = parent position

```
-> Public class MaxHeap {
    Private ArrayList<Integer> heap = new ArrayList<>();
    Public ArrayList<integer> getheap() { return this.heap }
                                  -> Child position (left/right)
    Private int getparentposition (int position) {
        if (position == 0) { return 0; }
        return (position-1)/2
    }
```

Finding parent Element position => $\left(\frac{i-1}{2}\right)$

```
    Private int getleftchildposition (int position) {
        return (2* position) + 1;
    }
```

Finding left child Element position using parent Index => $2i+1$

```
    Private int getrightchildposition (int position) {
        return (2* position) + 2
    }
```

Finding Right child Element position using parent index $2i+2$

```
    private void swap (int position1 (P1), int position2 (P2)) {
        int temp = this.heap.get(P1)
        this.heap.set (P1, this.heap.get(P2))
        this.heap.set (P2, Temp)
    }
```

// Heapify up => will be used while Inserting. (We Always add Element at the end & then Heapify To get the right position)

```
    private void Heapifyup (int position) {
        int parent pos = getparentposition((position);
        if(this.heap.get (parent pos) < this.heap.get (position)) {
            Swap (position, parent position)
            this.Heapifyup( parent position)
        }
```

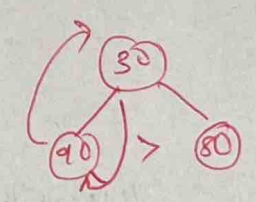
// Insert

```
    public void Insert (int value) {
        this.heap.add( value)
        if( this.heap.size > 1) {
            int index = this.heap.size() - 1;
            this.heapifyup (index)
        }
    }
```

// Heapify Down - This is used while deleting

```
private void heapifydown (int position) {
    if (this.heap.size() > 0) {
        int elementAtposition = this.heap.get (position);
        int leftchildpos = getleftchildposition (position);
        int rightchildpos = getrightchildposition (position);
        int largerelementpos = position.
        if (leftchildpos < this.heap.size() && this.heap.get(leftchildposition) >
                                                elementAtposition) {
            largerelementpos = leftchildpos;
        }
        if (rightchildpos < this.heap.size() && this.heap.get (rightchildpos) >
                                                this.heap.get (largerelementpos) {
            largerelementpos = rightchildpos;
        }
        if (largerelementpos != position) {
            swap (position, largerelementpos)
            heapifydown (largerelementposition)
        }
    }
```

If above if satisfied we will compare with the ~~element~~ left element with right element



// Delete

```
public Integer delete (int position) {
    if (this.heap.isEmpty) { print ("Heap is empty"); return null; }
    if (position >= this.heap.size()) { print ("overflow") ); return null;
    int lastindex = this.heap.size() - 1
    // move element to last -> And then remove
        swap (position, lastindex)
    int ele = this.heap.remove (lastindex)
    this.heapifyDown (position)
    return element
}
```

for priority queue Always the first element will be popped out delete()

Heap sort

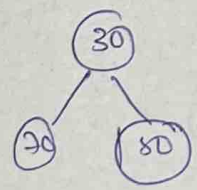
↳ Heap sort is the ~~form~~ arranging in the form of either descending to ascending (or) accending to decending.

↳ let say we are working with max Heap → we first find the max element and place the max element at the begining. we repeat the same process for the remaining.

Alogrithm for Sorting in Increasing order

1) Build a max heap from the Input data
2) At this point the largest item is sorted at the root of the heap.
   ↳ Replace it with the last item of the heap followed by reducing the size of heap by 1. heapify the root of the tree.
3) Repeat the step 2 while the size of the Heap is ~~greater~~ > 1

Heapify Example

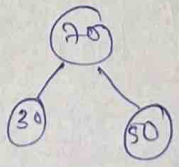


=) Child 70 is greater than the parent 30

=) swap child 70 with the parent 30

0	1	2
30	70	80

↳



70	30	80

i = 0

L = 2i + 1 => 1

R = 2i + 2 => 2

70 > 80 ✓

30 > 70 ✗

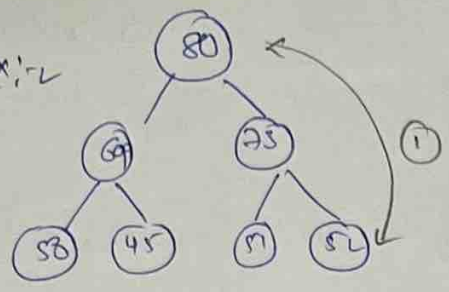
⇓

swap

⇓

70	30	80

Ex:-2



Max Heap

80	69	73	58	45	51	52

⇓

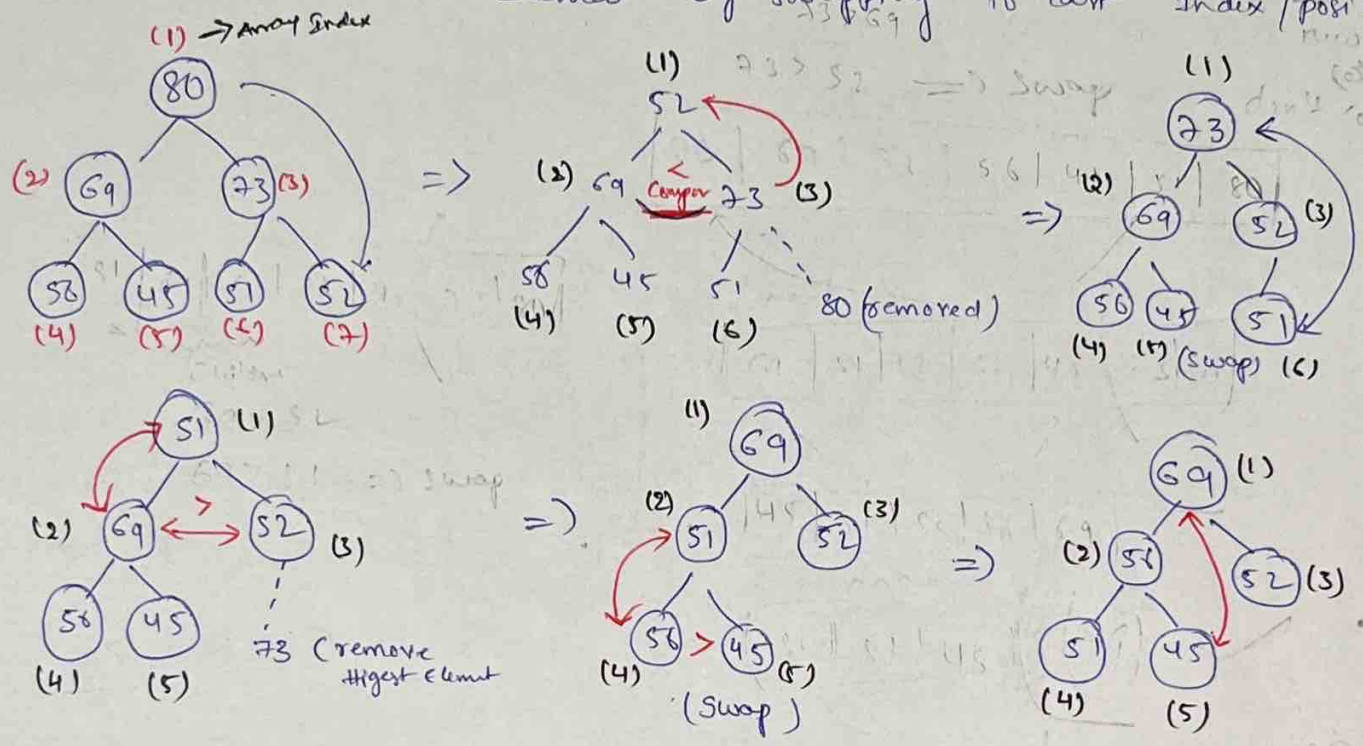
step 1 :- swap Root & Last node

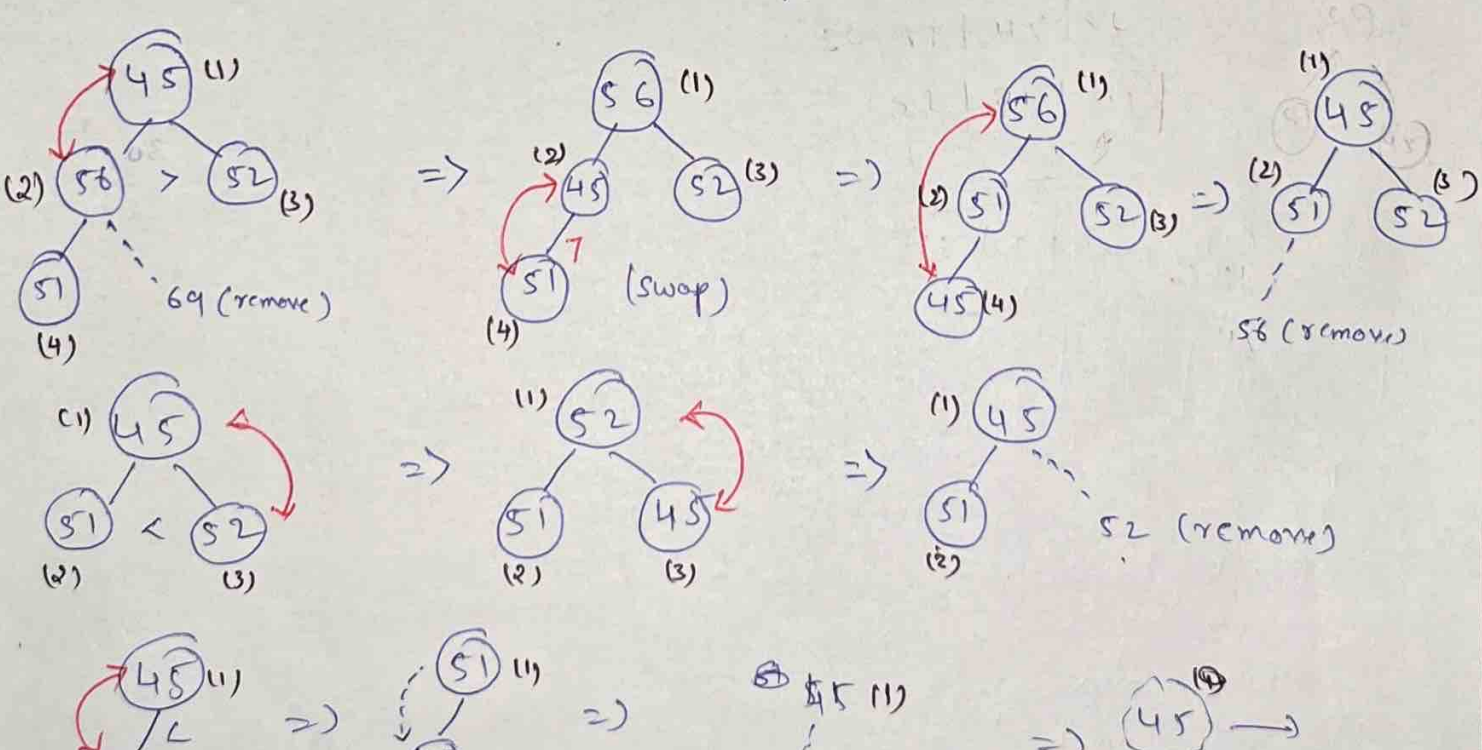
52	69	73	58	45	51	80

✗ delete

80
73
69
58
52
51
45

= Delete the Max Element by swapping to last Index / position





Priority Queue (Heap + Queue)

* Priority Queue is an abstract data type, which is similar to Queue, however in the priority queue, every element has some priority

* The priority of the elements in a priority queue determines the order in which elements are removed from the priority queue. Therefore all the elements are either arranged in an Ascending (or) descending order.

→ So Priority Queue is an Extension of the Queue with the following properties

→ Every item has a priority associated with it

→ An element with priority is dequeued before an Element with low priority.

→ If two elements have the same priority. they are served according to their order in the queue.

Elements — 73, 56, 80, 69, 45, 51, 52

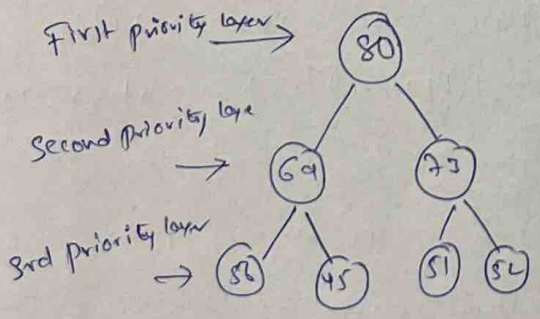
Normal Queue (FIFO) is

0	1	2	3	4	5	6
73	56	80	69	45	51	52

↑ First element to remove.

↳ But we combine Heap & Queue

↳ i.e we can arrange them in a sorted manner of this Max value so Here we giving priority to the Max value



For normal Queue Insertion will be

→ O(1)

→ Deletion = n + n = 2n ≈ N

↓ Finding an Element

↓ Shifting the element after Deleting.

73	56	80	69	45	51	52

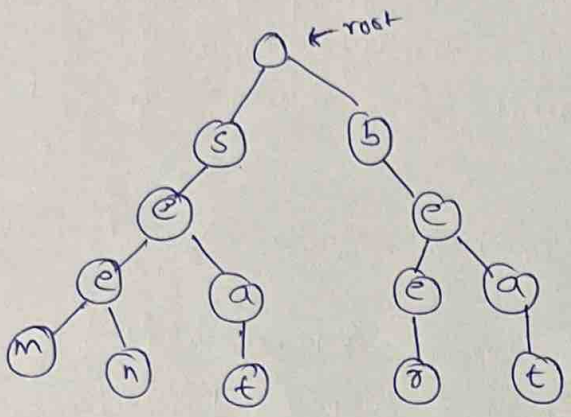
↑ delete

For Heap

→ Deletion = log(n)

Tries

Trie :- Trie is a specialized tree structure in which we can store character, so here in each node we are going to store some character, And If we are following a path we are going to form a word



for Example

s — e — e — m

s — e — a — t

b — e — a — t

Word - list

seem
seen
sea
seat
be
bee
beer
beat

→ Motive of trie is to bring out quick search result
↳ For Example google search.

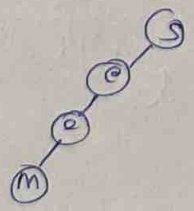
→ In Trie → we have root and this root node is Empty in the form of data (i.e no data) and then root node is connected with several alphabets, in English we have 26 so we are going to connect with 26 Alphabets

→ In Binary tree we can connect at most 2 nodes but in tries we can connect with 'n' no. of nodes.

*→ If we observe the Word - list we are utilizing the previous character to make different characters, that means we are making search Easy.

↳ Suppose I am at root node & if I need to search the word (seem) and see this word Existing in trie (or) not

↳ first I will search for "s" followed by "e" . . . .



=) ~~Otherwise if the trie is not there~~

→ If I am going to store Everything in an Array & have to search Each and Every word.
↳ this is going to be very difficult if there are millions of word.

→ But with "trie", I JUST have to search their Child
↳ In our case we need to search with word "s" that means we can ignore of searching all other alphabets (A, B. . . . z except s)

→ So "Trie" is an Efficient information retrieval data structure
→ Using "Trie" Complexities can be brought to optimal limit (key length)

→ Trie – Internals

Words

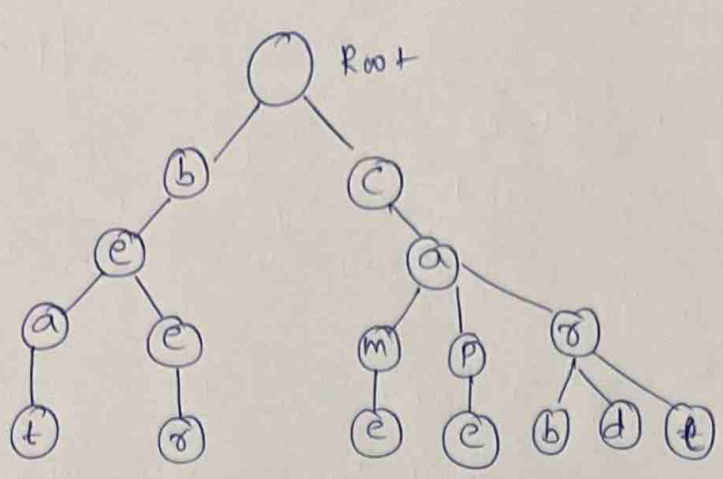
be	cap
beat	carb
beer	card
cam	car
came	cart
cape	

→ Every node of Trie consists of Multiple branches
→ Each branch represents a possible characters of keys.
→ We need to mark the last node of Every key as end of word node.
→ A Trie node field "isEndOfWord" is used to distinguish the node as end of word node.

→ Sample structure
↳ Class TrieNode {

```
    private static int Alphabet-size = 26
    private bool isEndOfWord;
    private TrieNode[] children = new TrieNode[26]
    //Constructor
    TrieNode () {
        isendofword = false;
        for (int i = 0; i < Alphabet-size; i++) {
            Children (i) = null
        }
    }
```

→



Words

be
beat
beer
cam
carb
card
car
cart
came
care

Insertion

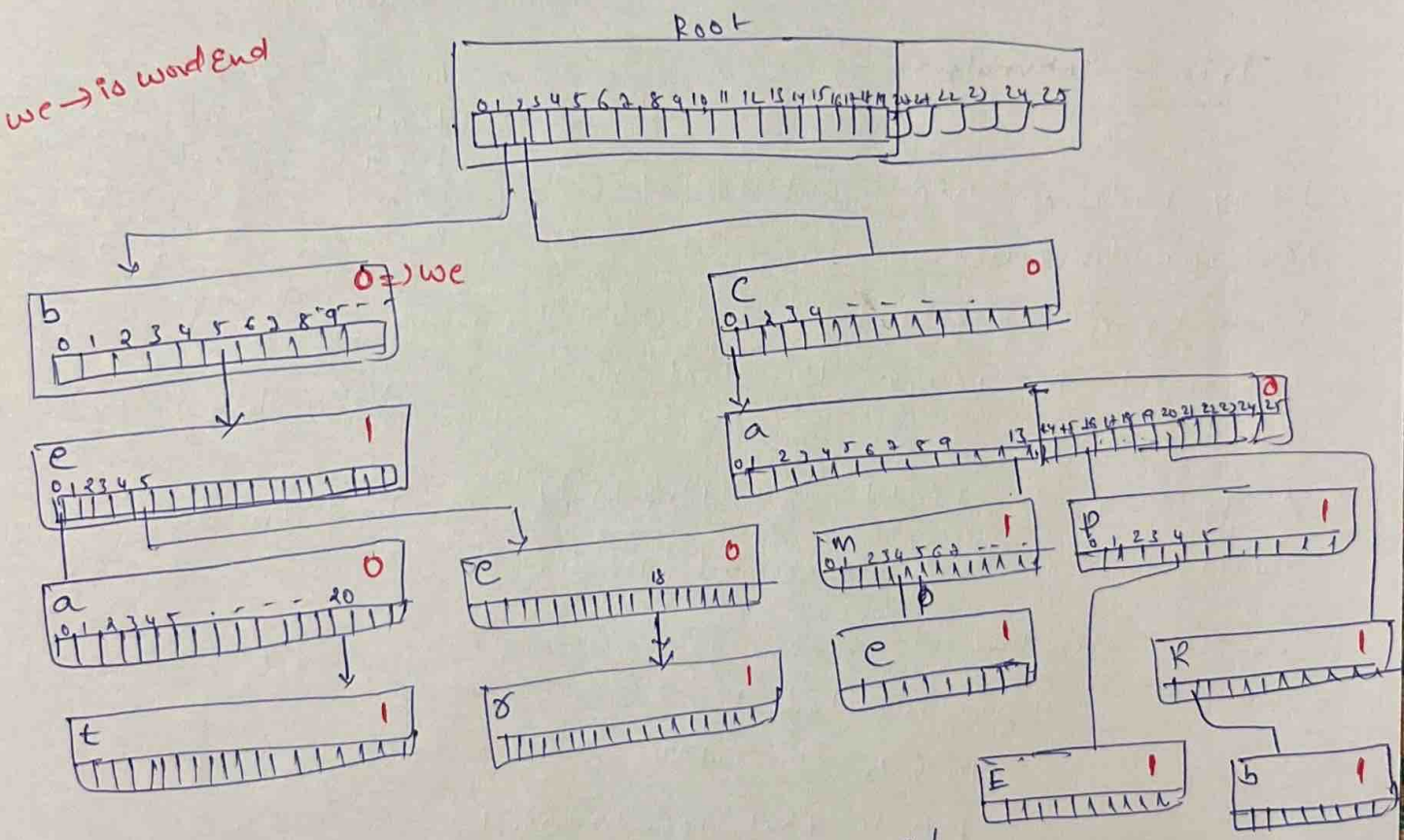
↳ Calculating the Position ⇒ Every Alphabet has a ASCII key

a = 97
b = 98
c = 100
D = 101
e = 102

→ Let say we want to insert "be"

b ⇒ b − a ⇒ 98 − 97 = 1

e ⇒ e − a ⇒ 102 − 97 = 5



→ We will set word End (we) flag as 0/1

Let say word be → so there is no word "b" in our word list so we set flag as "0"

→ for e node → we have the word (i.e be) → e is the Ending Alphabet so we set the flag as "1"

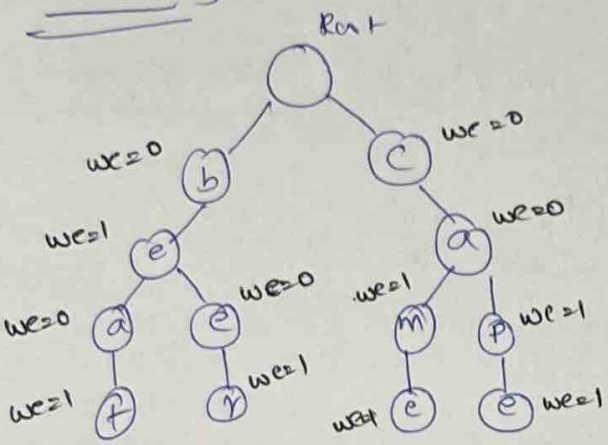
Trie → complexity

Search → $O(k)$
↳ length of the word

Insert → $O(k)$
↳ length of the word.

Delete → $O(k)$
↳ length of the word.

Searching



Word
↳ be, beat, beer, can, came, cape, capc

→ Search for cape

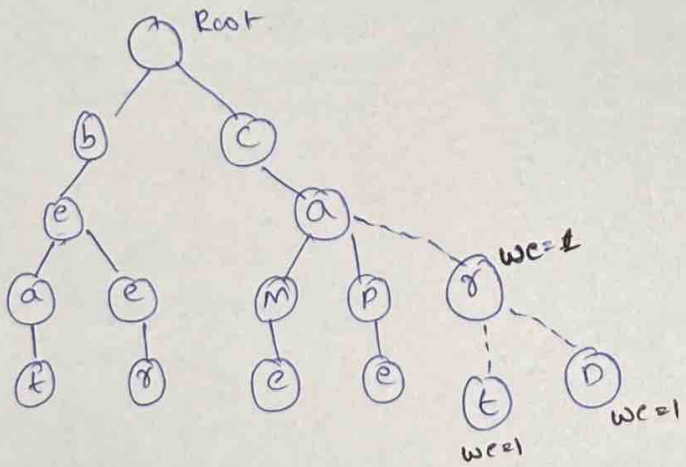
Root $\xrightarrow{1}$ c $\xrightarrow{2}$ a $\xrightarrow{3}$ p $\xrightarrow{4}$ E

4-path = length (cape) = 4

Hence complexity is $O(k)$
↳ length of word.

Insert

↳ Insert word car, cart, card



1) C A R

↳ Root → C — A → R

There is no letter R
So Add new node "R"
& we = 1

2) C A R T

Root $\xrightarrow{1}$ C $\xrightarrow{2}$ A $\xrightarrow{3}$ R $\xrightarrow{4}$ T

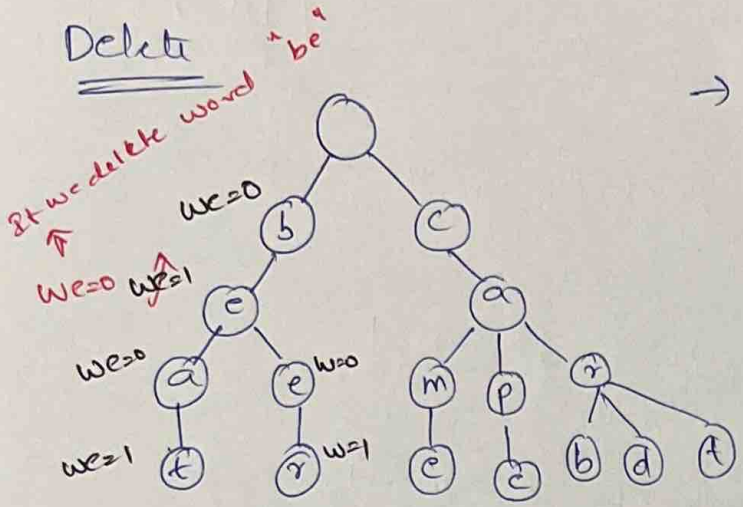
There is no letter T
So Add new node of "T"
& we = 1

Observation

for letter C A R T → 4 paths = length(word) to cart

Hence complexity
↳ $O(k)$ ↳ length of word

Delete



→ let say I want to delete word "be"

↳ As the other words get linked we can not physically delete the node "b" & "E"

↳ If we delete the chain link will break

↳ so instead we can just mark the word end (WE) as "0" from "1"

→ So to delete the word we need to traverse the word till last letter and mark "We" as "Zero"

↳ so complexity is $O(k)$

↳ length of the ~~text~~ word.

Trie - Advantages

↳ less time complexity for insertion, deletion, search

↳ Faster than BST,

↳ Faster than Hashing → we no need to compute any Hash Functions.

→ we can easily print all words in alphabatical order

↳ which is not easily possible with hashing.

→ we can efficiently do Prefix-search / Auto-complete

Disadvantages

↳ It needs lots of memory for storing strings, For each node we have two many pointers (Equal to no. of Charaters of the Alphabet)

→ If space is constraint then "Ternary search Tree" can be preffered for dictionary implementations.

TST → Time complexity $O(h)$ → Height of the Tree

It also supports

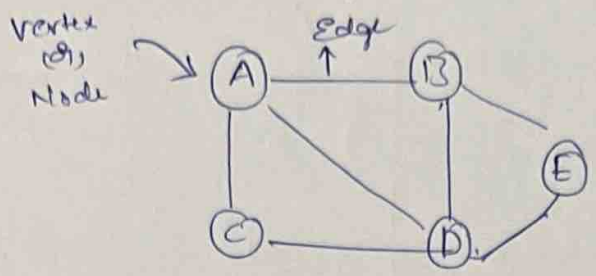
↳ prefix search

↳ Alphabatical order printing

↳ nearest neighbour search

Graphs

→ Like Trees Graph is a non-linear data structure.



Graph
- ↳ Node / vertex
- ↳ edges

→ Till now, we have been discussing about Trees, we have nodes, we refer some as root and other as child etc. But in graphs we are not going to follow like that

- ↳ In graph we have node also called as Vertex
- ↳ Nodes are connected by edges.

→ In simple words, graph are mathematical representation, mathematical structure that helps us to represent relationships b/w two objects

→ In tree we can ~~go~~ Traverse from parent to child but not child to parent, but in graphs relationship b/w nodes are different

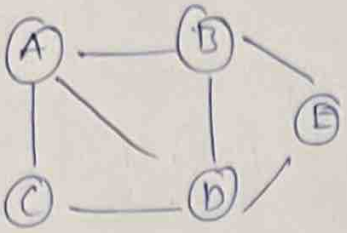
→ In Tree we are starting from the root and then all the children, but in graph this is not the case we can start ~~from~~ according to our requirement.

→ Suppose, i am going to start from "A" and there is no hierarchical relationship at this point of time. If you see single line that means it's a unidirectional relationship

↳ where as In trees it is a directional relationship i.e we have direction like A → B → C → D

⇒ Graphs are used to solve real-life problems → Graphs are used to represent networks may include paths in city (or) telephone N/w (or) circuit N/w. These are also used in social N/w like FB, linkedin → In FB → each person is represented with a node.

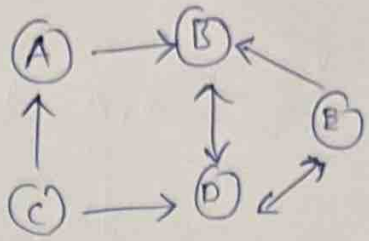
1) Graph Types — → undirected, → directed } Based on direction



Undirected graph

ie I can Traverse A → B & B ↔ A

Both - direction can Traverse



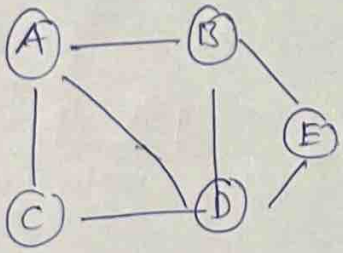
Directed graph

A → B (✓) } B → D ✓
B ← A (✗) } D → B ✓

Both side Arrows

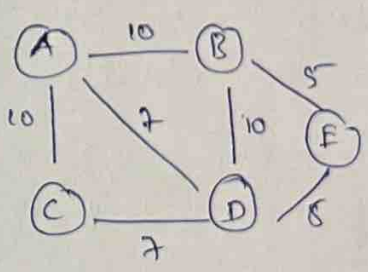
→ Basically we can specify directions.

2) Graph Types — → Unweight graph, → weighted graphs } Based on Edge weight



Unweighted

It means the the connection b/w A to B & A to C & A to D & D to E are Equal

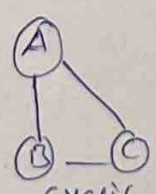


→ Here we have weight for Edge i.e Reaching from A → B is not same as A → C

→ Consider nodes as ~~cost~~ City name

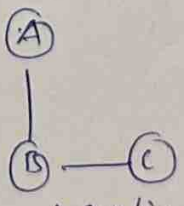
→ There will be diff distance b/w two points in city.

3) Graph Types — Cyclic graph, → A cyclic graph



Cyclic

→ Start from a node Traverse & get Back to the node (or) origin



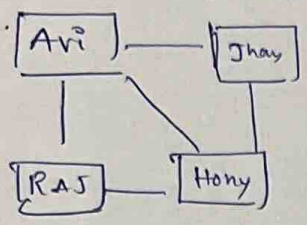
A cyclic

→ not a closed loop vise-versa to cyclic

Use cases of graphs

1) Undirected graph ⟹ Ex facebook

↳ In FB we have options to connect with people as a friend

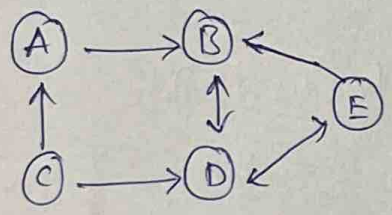


⇒ Let's say Ari sent Req to Hony and she Accepted it, i.e Un direction graph relation

↳ I can view her status & she can view My status / Activity

↳ i.e there is no restriction

2) Directed Graph ⇒ Instagram / twitter.

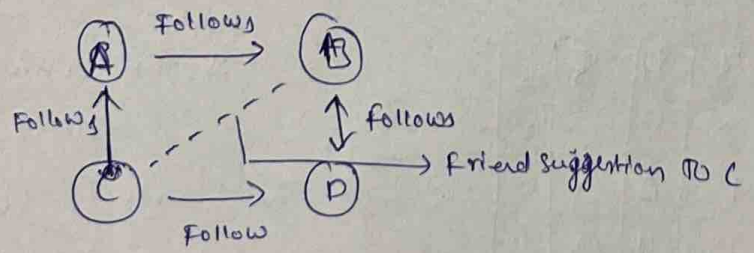


Let say "C" Requested to Follow "D" then "C" can see all Activities because "C" is following "D". but this does not mean that "D" can see all the activities of "C" Because D → still not connected

→ D ↔ E → Bidirectional → Both can see each other

→ This is more Important to take some business decisions

Say

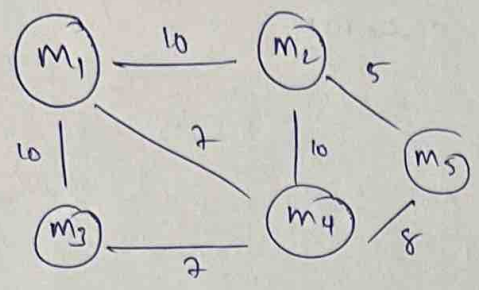


Here Instagram has info that A follow B and Even D Following B. So Insta start suggesting "C" to follow "B"

→ similarly Add suggestions etc.

3) Weighted - graph

⇒ Youtube video suggestions. (or) Netflix to suggest the Next videos to watch



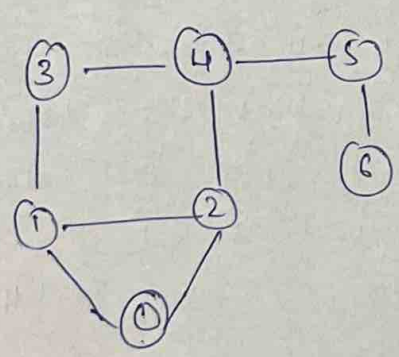
→ Cab sharing suggestion based on distance

⇒ Graph Implementation

↳ A graph can be represented by

1) Adajacency matrix (Ad matrix)
2) Adajacency List (Ad List) → Popular, widely used
3) Edge List

Ex:- Undirectional / Unweighted :-



no. of vertexes / nodes = 7

no. of Edges = 8

1) Adjacency matrix

Nodes

	0	1	2	3	4	5	6
0	0	1	1	0	0	0	0
1	1	0	1	1	0	0	0
2	1	1	0	0	1	0	0
3	0	1	0	0	1	0	0
4	0	0	1	1	0	1	0
5	0	0	0	0	1	0	1
6	0	0	0	0	0	1	0

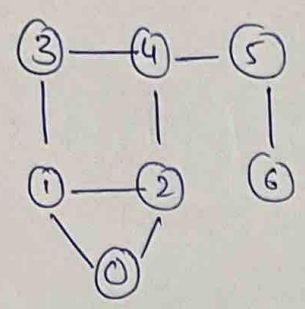
Nodes

2) Adjacency List

Ex:- 0 is connected with 1 & 2

1 is connected with 0, 2, 3

so in matrix if there is a connection we mark as "1" else 0.



=> **Adjacency list representation** (Ad list)

↳ Most widely used.

=) When ever we create a vertex (or) node we create a field for node and it's child (or) Ad list.

Value → String, Int etc.

Adlist → Array (or) Hash Table

for example node (0) → Adlist = [1, 2]

node (1) → Adlist = [0, 2, 3]

~~Adjacency list~~ ~~Adjacent~~

nodes

Nodes			
0:	1	2	
1:	0	2	3
2:	0	1	4
3:	1	4	
4:	2	3	5
5:	4	6	
6:	5		

Ad List Elements

Edge List Representation :-

↳ In this representation → we refer the graph and count Edge not nodes

Edge no (can be of Any values) → Not Node Remember.

0:	0	1
1:	0	2
2:	1	2
3:	1	3
4:	2	4
5:	3	4
6:	4	5
7:	5	6

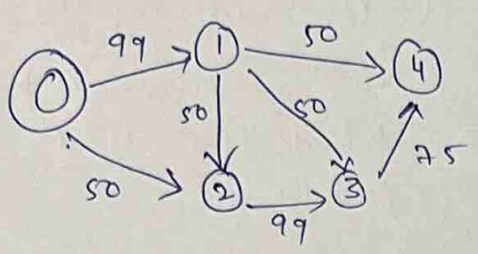
=) Edge 0 → connected with 0 & 1

=) Edge 1 → connected with 0 & 2

=) Edge 2 → connected with 1 & 2

Any no. for understanding purpose we use seq numbers.

Directed / weighted Representation



no. of vertexs / nodes = 5

no. of Edges = 7

Adjacency Matrix

Nodes

	0	1	2	3	4
0	0	99	50	0	0
1	0	0	50	50	50
2	0	0	0	99	0
3	0	0	0	0	75
4	0	0	0	0	0

Nodes

Adjacency List

0:	(1, 99)	(2, 50)	
1:	(2, 50)	(3, 50)	(4, 50)
2:	(3, 99)		
3:	(4, 75)		
4:			

node value ← (4, 75) → weight

Edge List

Edge No can be Any value

Nodes → weight

0:	0	1	99
1:	0	2	50
2:	1	2	50
3:	1	3	50
4:	1	4	50
5:	2	3	99
6:	3	4	75

Recursion

DS-31

→ How recursion works ?

↳ Topics
- → Main memory
- → Function call
- → Nested Function call
- → Examples.

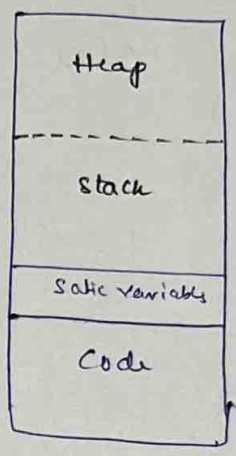
Program

```
void main () {
    int a,b,c;
    float x;
    Char c;
    Printf("Enter 3 Numbers")
    Scanf("%d %d %d",@);
    - - -
    - - -
}
```

} Data

} Instructions

Memory / RAM



→ Code :- Instructions of a program / Function are kept here

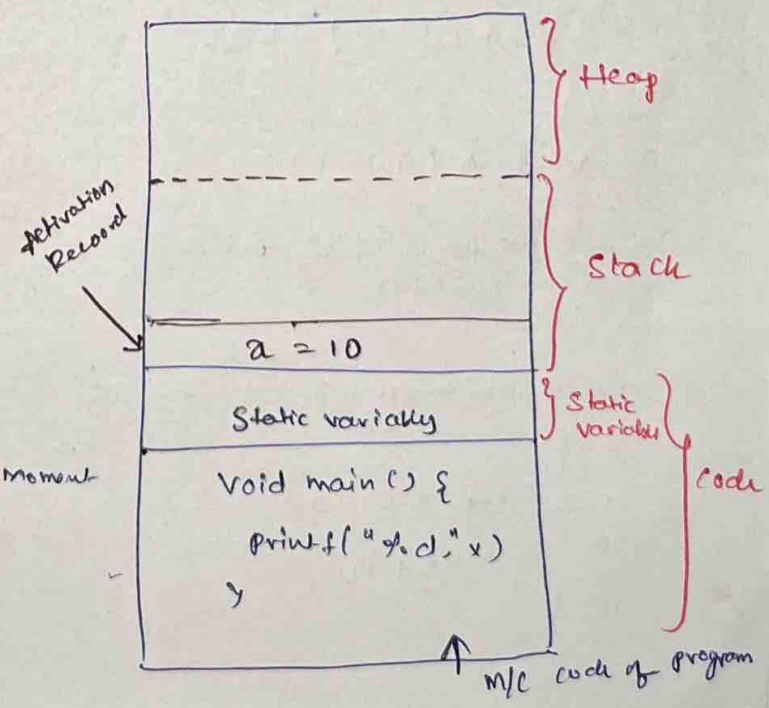
→ Static variables → It is also a part of code, it stores any global (or) static variables of a Function.

→ Stack :- Here, the variables / Data a Function will be stored

→ Heap :- Used for dynamic Allocation of Memory

→ Loading of Program

```
void main () {
    int x = 10;
    Printf("%d", x);
}
```

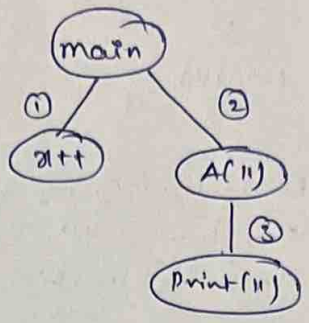


→ Activation record is created at the moment Function is called.

function call

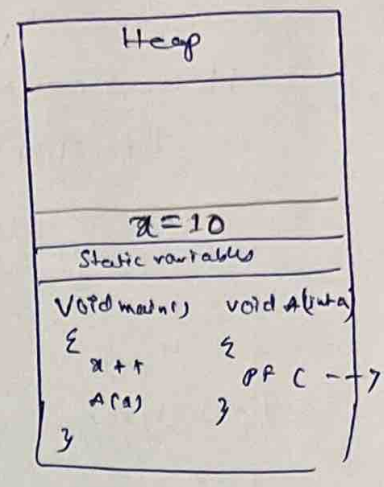
```
1  void A (int a)
2  {
3      printf("%d", a);
4  }

5  void main() {
6      int x = 10;
7      x++;
8      A(x)
9  }
10
```

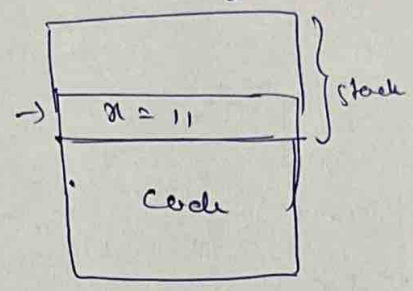


O/P = 11

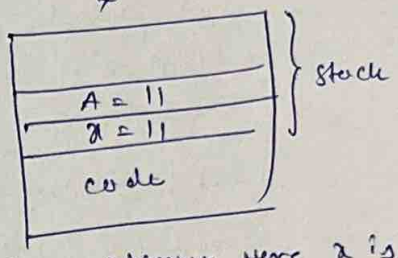
Initial



When Main Executed



When A Called



If we observe here x is still in stack as the main function Not yet terminated.

⇒ When Function Executes and Terminates → the Activation Record of that fn will be removed.

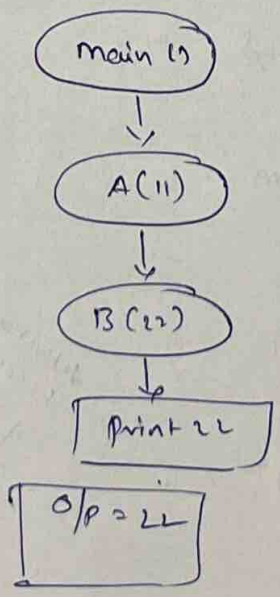
↳ When the function A Terminates → Activation record "A" will be Terminated. Slly for main. → x will be terminated

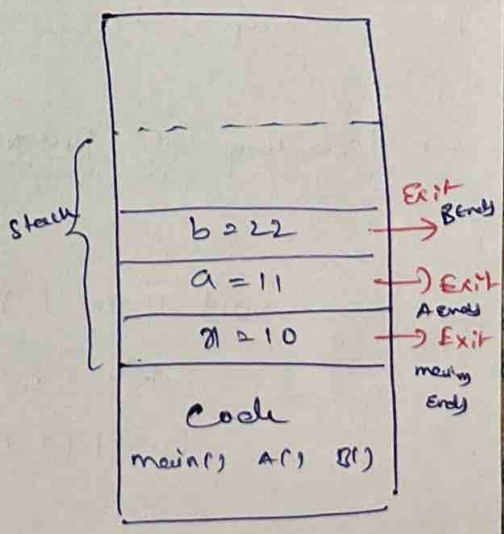
→ Nested Function Call

```
1  void B (int k)
2  {
3      printf("%d", k);
4  }

5  void A (int i)
6  {
7      ~~printf("%d", i);~~
       B(2×i)
8  }

9  void main ()
10 {
11     int x = 10;
12     A(x+1);
13 }
14
```





DS - 31:7

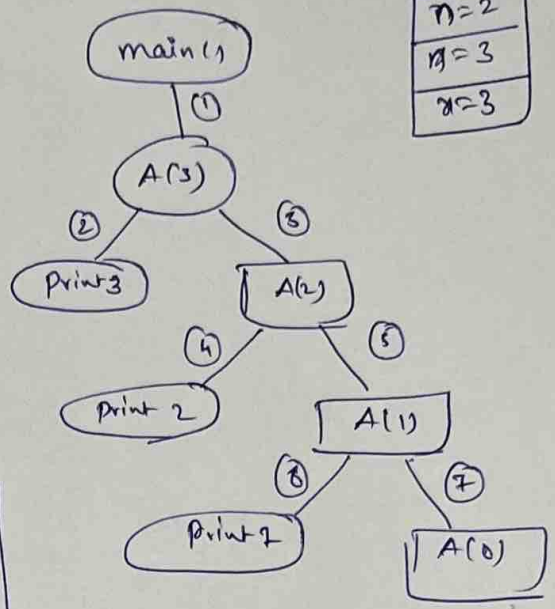
~~Recur~~

Recursion (Tail Recursion)

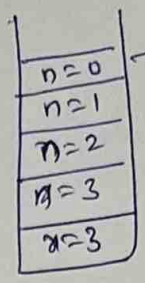
→ Last statement in a Recursive call

```
void A (int n)
{
    if (n>0)
    {
        Print f ("%d", n);
        A(n-1)
    }
}

void main ()
{
    int x = 3;
    A(x);
}
```



O/P
3
2
1



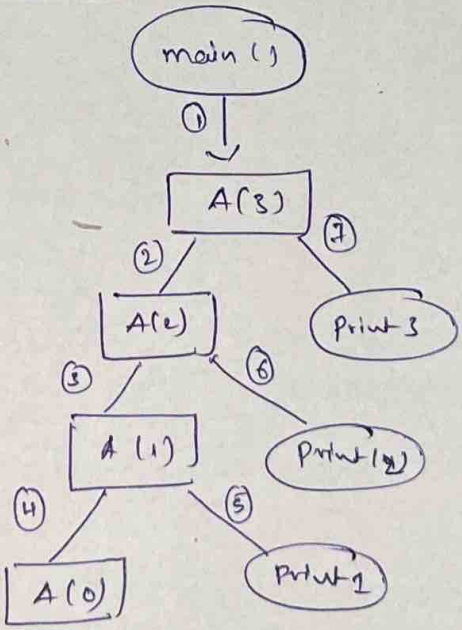
→ Terminate condition Failed. ⇓ goes to previous call. A(1)

→ Recursion (Head Recursion)

→ First Recursion call And then other statements

```
void * ()
{
    if (n>0)
    {
        A(n-1);
        Print f ("%d", n);
    }
}

void main ()
{
    int x = 3;
    A(x);
}
```



O/P
1
2
3

Recursion

* Recursion is a function that calls itself.
* Whenever we use recursion we need to remember base case (or) base condition
  - ↳ Base case is the final stop point (the condition where loop need to stop calling itself)
  - → It helps us to stop our infinite loop.

Ex:-
```
def recursion():
    print("Hellowoorld")
    recursion()
recursion()
```
Here what happens is function will run ~~for~~ infinite times. as there is no condition to break. (i.e Base case) is missing

→ So, it is always important to provide the base case when we are working with recursion. Same function can be written as

```
def recursion(count):
    if count == 500:
        return 0        # break
    else:
        print("Hello world")
        recursion(count + 1)
count = 0
recursion(count)
```

Ex:- factorial

```
def factorial(num):
    if n == 0
        return 1
    else
    return num * factorial(n-1)
print(factorial(5))
```

Searching Algorithm

1) Linear Search

Input = [5, 3, 1, 9, 2]

Search = 9

```
for ( i = 0 ; i < input.length ; i++ ) {
        if ( input[i] == search) {
                return i
        }
}
```

Space O(1)

Time complexity O(n)

2) Binary Search :-

↳ Pre requisite → The Input Array should be sorted.

→ If you have sorted Array only then we can perform binary search.

$$\boxed{mid = \frac{Start + End}{2}}$$

Input = [2, 4, 5, 7, 8, 9, 15, 20]
0 1 2 3 4 5 6 7

→ Unlike linear search we will not loop through all the elements, Instead we break the array into half

Search for Element 9 ↳ middle => (0 + 7) // 2 = 3 (round off)

① $x = [\overset{0}{2}, \overset{1}{4}, \overset{2}{5}, \overset{3}{7}, \overset{4}{8}, \overset{5}{9}, \overset{6}{15}, \overset{7}{20}]$

↑ middle → 7 > 9

$x[3] > 9$ × → Consider second half as the Input is sorted

② $x = [\overset{4}{8}, \overset{5}{9}, \overset{6}{15}, \overset{7}{20}]$

↑

middle => (4 + 7)/2

= 11/2 = 5 round off

③ $9 \geq x[5]$ → True

↳ so consider first half

$[\overset{4}{8}, \overset{5}{9}]$

↑ ✓

Binary Search (Example-2)

$x =$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
3	6	8	12	14	17	25	29	31	36	42	47	53	55	62
low(l)							m							high(h)

key = 42
To search

		low (l)	High (H)	m = floor(l+H/2)
①	Iteration 1	0	14	(0+14)/2 = 7
②	1) $x[7] = 29 = 42$ key → False 2) 29 > 42 → False 3) 29 < 42 → True ↳ Ignore first half (As the list is sorted)	l = m+1 l = 7+1 l = 8	14	8+14/2 = 22/2 = 11 8 9 10 11 12 13 14 31 30 42 47 53 55 62 ↑ m ↑ H
③	$x[11] = 47$ 47 = 42 ✗ 47 > 42 ✓ ↳ Ignore second half	l = 8	h = m−1 h = 11−1 h = 10	$\left(\frac{8+10}{2}\right) = 18/2 = 9$ 8 9 10 31 30 42 47 L m H
④	$x[9] = 30$ 30 > 42 → False 30 < 42 ✓ ↳ Ignore	l = m+1 l = 9+1 l = 10	h = 10	$\left(\frac{8+9}{2}\right) = \frac{18}{2} \Rightarrow 8.5 \Rightarrow 8$ $\left(\frac{10+10}{2}\right) = 20/2 = 10$

$x[m] = x[10] = 42$ ✓

42
L m H

② Search key = 12

		l	h	m = (l+h/2)
①		0	14	(0+14)/2 = 7
②	$x[7] = 29$ 29 = 12 ✗ 29 > 12 ✓ ↳ Search left side	0	h = m−1 h = 7−1 h = 6	(0+6)/2 = 3

$x[3] = 12 =$ key Found ✓

* Search for key not Exist in List say 30 = key

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
x =	3	6	8	12	14	17	25	29	31	36	42	47	53	55	62
	l							m							h

$m = 0+14/2 = 7$

Iteration 1

$l = 0$; $h = 14$ $\Rightarrow m = 0+14/2 = 7$

$x[7] = 30$ (key) $\Rightarrow 29 = 30 \rightarrow$ False

$29 > 30 \Rightarrow$ False

$29 < 30 \Rightarrow$ True $\rightarrow$ so Ignore left part

Iteration 2

prev m value

$l = m+1 \Rightarrow 7+1 = 8$

$h = 14$

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Ignore	~~3~~	~~6~~	~~8~~	~~12~~	~~14~~	~~17~~	~~25~~	~~29~~	31	36	42	47	53	55	62
								m	l						h

$\Downarrow$

$m = \left(\frac{l+h}{2}\right) \Rightarrow \left(\frac{8+14}{2}\right) = \frac{22}{2} = 11$

Iteration : 3

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Ignore	~~3~~	~~6~~	~~8~~	~~12~~	~~14~~	~~17~~	~~25~~	~~29~~	31	36	42	47	53	55	62
									l			m			h

$x(m) \Rightarrow x(11) = 47$, $47 = 30$ ✗

$47 > 30 \rightarrow$ so ignore Right & shift h

$h = m-1 \Rightarrow 11-1 = 10$

Iteration : 4

$l = 8$

$m = l+h/2 = \frac{8+10}{2} = 18/2 = 9$

	8	9	10	11
	31	36	42	47
	l		h	m

Iteration 5

$x(8) = 31$

$31 >$

	8	9	10
	31	36	42
	l	m	h

$\Rightarrow$ $x(m) = x(9) = 36 \Rightarrow 36 = 30$ ✗

$36 > 30$ ✓ So Ignore Right shift h

$h = m-1 \Rightarrow 9-1 = 8$

	8
	31
	m, l, h

DS-33

8	9	10
31	36	42
l	m	H

Key = 30

$x(m) = x(9) = 36 \Rightarrow 36 \geq 30$ ✗

$36 > 30$ ✓

⇓

Ignore Right → shift H

Iteration 5

$H = m-1 \Rightarrow 9-1 = 8$

8
31
l, m, H

$\Rightarrow x(m) \Rightarrow x(8) = 31$

$31 = 30$ ✗

$31 > 30$ ✓

~~So shift to~~

⇓

⇐ Ignore left → Shift l

$l = m+1 = 8+1$

$l = 9$

$l > H$ → Invalid case

⇓

Break the loop

↳ Key 30 → not found.

Binary Algorithm

0	1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8	9
S								E

input = [1, 2, 3, 4, 5, 6, 7, 8, 9] → Target = 8

```
def binary_search(input: List(int); target: int) -> int:

    start = 0
    end = len(input) -1
    m = 0
    while(start <= end):
        m = (start + End)//2      => It will give floor value
        mid_value = input[m]
        if target == mid_value:
            return m
        elif target > mid_value:      => consider Right half
                                         i.e shift start position
            start = m+1
        else:                         => If target > mid value
                                         i.e shift high position
            End = m-1
```

PTO

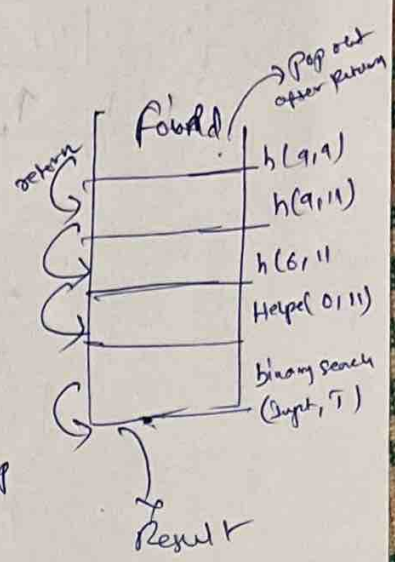
Binary Search using recursion :-

```
def binary_search (input, target):
    left = 0
    Right = len(input) - 1
    result = helper (input, left, right, target)
    return result

def helper (input, left, right, target):
    ~~if left~~ // Base condition to break recursion loop
        if left <= Right:
            middle = (left + Right)/2
            middle_element = input (M)
        if target == middle element:
            return m
        elif target > middle_Element:
            left = middle +1
            return helper( input, left, right, target)
        else:
            Right = middle -1
            return helper( input, left, right target)

    else
        return = -1    //If no element found
```

Return is mandatory to get the call ~~before~~ control from stack to previous call



Pto

Sorting Algorithms :-

* Sorting algorithms are methods of reorganizing a large number of items into some specific order such as hight-low, low-high

S.No	Algorithm	Best	Average	Worst
1)	Quick sort	$n\log(n)$	$n\log(n)$	$O(n^2)$
2)	Bubble sort (elementary) ①	$O(n)$	$O(n^2)$	$O(n^2)$
3)	Merge sort	$n\log(n)$	$n\log(n)$	$n\log(n)$
4)	Insertion sort (elementary)	$O(n)$	$O(n^2)$	$O(n^2)$
5)	Selection sort (elementary)	$O(n^2)$	$O(n^2)$	$O(n^2)$
6)	Heap sort	$n\log(n)$	$n\log(n)$	$n\log(n)$
7)	Radix sort	$O(k)$	$O(nk)$	$O(nk)$
8)	Bucket sort	$O(n+k)$	$O(n+k)$	$O(n^2)$

Bubble Sort :-

(P.T.O)

1) Bubble Sort

* Bubble Sort, also reffered to as "Comparison Sort", it is a simple Sorting Algorithm that repeatedly goes through the list.

→ Compares adjacent Elements and swaps them if they are in the wrong order.

→ This is simpilist Algorithms and inefficient at the same time.

Example

0	1	2	3	4
8	5	7	3	2

⇒ for understanding purpose let's draw vertically.

↳ let's say I am sorting in Ascending order.

→ pass means we are going through list

1st pass

8 5 7 3 2

8 > 5 so swap

And remaining Elements As it is

5	5	5	5	5
8	7	7	7	7
7	8	3	3	3
3	3	8	2	2
2	2	2	8	8

larger element came down.

5 - Elements
4 - swaps (max)
4 - Comparision (max)
1 - Element sorted

⇒

2nd pass

5	5	5	5
7	7	3	3
3	3	7	2
2	2	2	7
8	8	8	8

→ If not greater it won't swap.

→ next Higest Element "7" Sorted

3 - Max swaps
3 - Max Comparison
2 - Element Sorted

3rd pass

5	3	3	3
3	5	2	2
2	2	5	5
7	7	7	7
8	8	8	8

3 - Element Sorted
2 - max comparision
2 - max swaps

4th pass

3	2
2	3
5	5
7	7
8	8

→ full list sorted

1 - Max comparision
1 - max swap

Here $n = 5 \Rightarrow 4$ passes

for n = Elements ⇒ (n-1) passes

for $n = 5 \rightarrow 1 + 2 + 3 + 4 \ldots \rightarrow$ comparisions

4th pass max swaps; for 3rd pass; 2nd pass; 1st pass

for n Elements ⇒ $1 + 2 + 3 + \cdots n-1$

$= \frac{n(n-1)}{2} = O(n^2)$

i.e for n Elements → $O(n^2)$ ⇒ max comparisions $= \frac{1}{2}(n^2 - n) \Rightarrow O(n^2)$

S/ly for n Elements → $O(n^2)$ ⇒ max swaps.

$\boxed{\text{Time Complexity} = \text{no. of Max Comparisions} \Rightarrow O(n^2)}$

→ Why it is Called Bubble Sort

↳ It is called bubble sort because the movement of Array Elements is just like air bubbles in the water

↳ Bubbles in water rise up to the surface; similarly the array Elements in bubble sort move to the End in Each iteration.

→ Observation

→ for First pass → 1 Element sorted → In our case 1 large count

→ for 2nd pass → 2 Element sorted → 2 largest Elements

→ slly For k pass → k Elements will be sorted → k largest Elements

* ↳ Suppose we don't want to sort whole Array And we just need

→ 1st largest element → then perform only 1 pass

→ 2 largest Elements → 2 passes

→ k largest Elements → k-passes.

Algorithm

↳ If we observe → for Every pass there will be one comparison reduced

```
public sort ( input ) {
    n = len(input)
    flag = 0
    for ( i=0 ; i < n-1 ; i++) {
        flag = 0 ;
        for( j=0 ; j < n-1-i ; j++) {
            // comparing Adjacent items
            if ( input(j) > input(j+1) {
                swap(input(j), input(j+1))
                flag = 1
            }
        }
        if ( flag == 1 ) {
            Break ;
        }
    }
}
```

→ (n-1) because for n-Elements there will be (n-1) passes.

→ i.e last i Elements are Already sorted.

→ (n-1-i) → for Every pass there only (n-1-i) swaps

This flag is useful To check whether the input is already sorted. → it can be done at any stage → so setting flag=0 inside for.

→ If already sorted there no need of any swap and can be break.

Ex:-

n	i	Pass #	max Swaps
5	0	1	4
5	1	2	3 (5-1-1)
5	2	3	5-1-2 = 2

→ Is bubble sort Adaptive ?

↳ bubble sort can be made Adaptive by using "flag".

Say we took the sorted Array

2	2	2	2	2
3	3	3	3	3
5	5	5	5	5
7	7	7	7	7
8	8	8	8	8

1st pass

↳ In 1st pass there is no swapping it means that the array is already sorted.

↳ It can be happen any pass. When Ever it is happening it will stop.

→ Analysis

↳ If the list is already sorted it will perform $(n-1)$ comparisions. i.e $O(n-1) \Rightarrow O(n)$

So for bubble sort

min time (or) Best case → $O(n)$

max time (or) worst case → $O(n^2)$

→ Is bubble sort stable or not ?

↳ Bubble sort is stable → i.e it wont exchange the order of the elements → observe Red "8"

↳

Ex :- (→ It won't change as 8 > 8 (false))

8	8	8	8	8
8	8	3	3	3
3	3	8	5	5
5	5	5	8	4
4	4	4	4	8

Duplicate →

Example Best case :- (14, 20, 25, 31, 44, 55)

Avg case :- (14, 25, 31, 20, 55, 44)

Worst case :- (55, 44, 31, 25, 20, 14)

Space complexity → $O(1)$

2) Selection Sort

To remember → Keeping one as fixed & selecting others.

DS-36

* The Selection Sort Algorithm sorts an array by repeatedly finding the min Element (considering Ascending order) from Unsorted part and putting it at the beginning.
* The ~~alg~~ algorithm maintains two sub Arrays in a given Array.
  - ↳ SubArray which is already sorted (left side)
  - ↳ Remaining sub Array which is Unsorted
* In Every iteration of selection sort, the min Element (considering ascending order) from the unsorted subarray is picked and moved to the sorted sub Array.

Algorithm

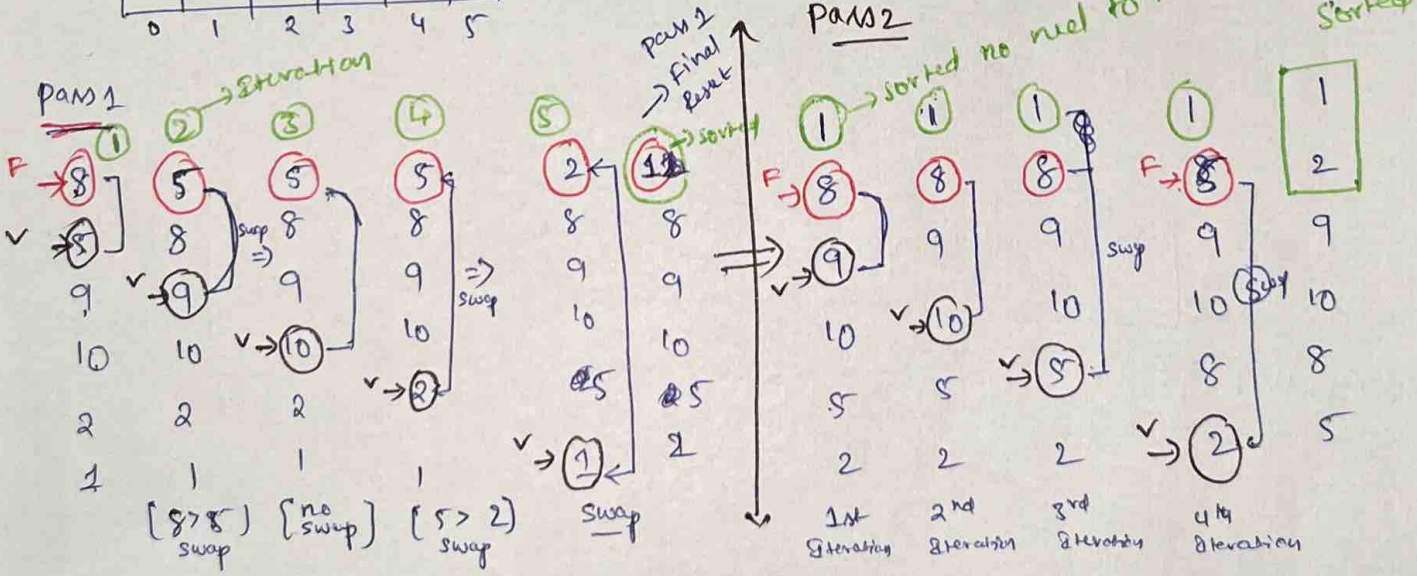
↳ In selection sort there is a fixed pointer (f) and the variable pointer (V) → which always initially points to the next position of the fixed pointer in Every pass.

For every Iteration V → will move To next position ↳ so

* if array [fixed] > arr [variable] (for Ascending order) then swap the positions.

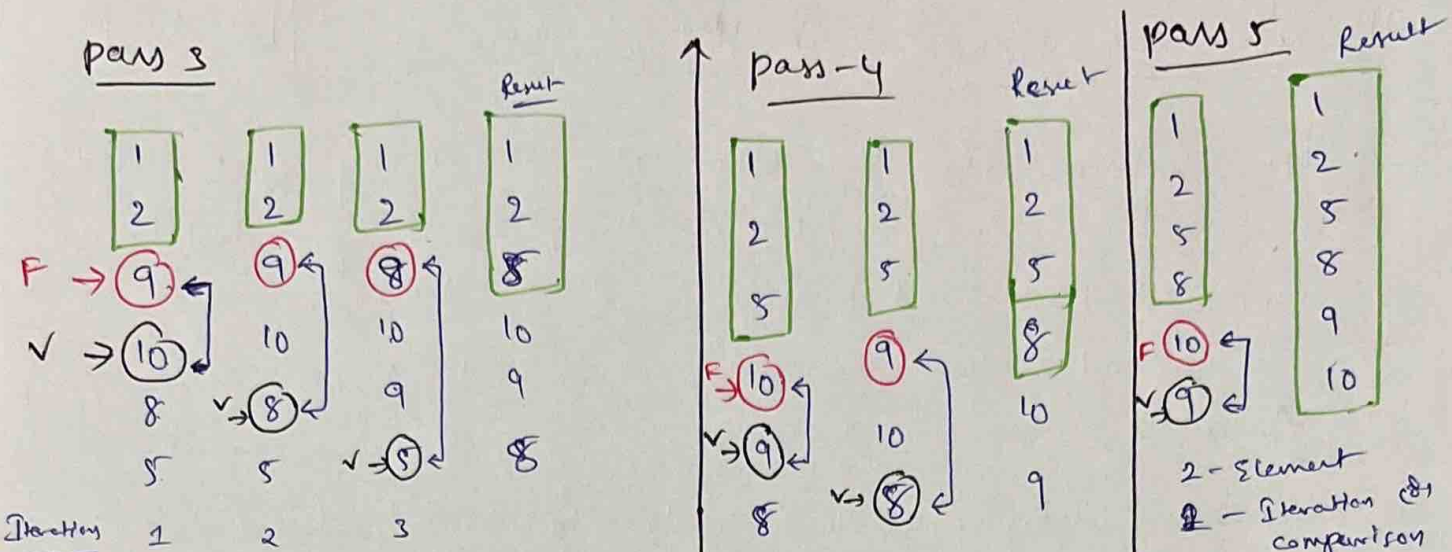
Ex :-

0	1	2	3	4	5
8	5	9	10	2	1



6- Elements
5- Iterations / comparisions
5- (Max swaps possible
Here 3 - swaps)
1- Element sorted.

5 - Element
4- Iterations / comparisions
4-) (MAX - swaps possible
1 → Element sorted.



4- Element
3- Iterations / comparisons
3- max swaps
3 -> Elements sorted at the end

3- Elements
2- Iterations / comparisons
2- max swaps
4- Elements sorted at the end

2- Element
1- Iteration / comparison
1- max swap

Observations

-> 6 Element -> 5 passes (6-1)

-> n Element -> (n-1) passes

-> for 6 Element -> 1, 2, 3, 4, 5 (MAX comparisons / Iterations)

pass 5, pass 4, pass 3, pass 2, pass 1

for n- Elements -> 1, 2, 3, 4 ---- n-1 => $\frac{n(n-1)}{2}$

=> $O(n^2)$

$$\boxed{\text{Time complexity} = \text{MAX Comparisions} = O(n^2)}$$

=> Selection sort can be made Effective by using flag. Similar like bubble sort

Selection sort Algorithm

```
def selection_sort (input: List[int]):
    n = len(input)
    is_swapped = 0
    for i in range(n):
        is_swapped = 0
        for j in range(i+1, n):
            if (input[i] > input[j]):
                # swap
                temp = input[i]
                input[j] = input[i]
                input[i] = temp
                is_swapped = 1
        if is_swapped == 0:
            break
    return input.
```

Here i = fixex
j = variable

As variable part always be the next position of the fixed.

Here we need to run "n" times as the sorted Elements placed at the top

Observation

→ Unlike bubble sort we need to run loop here for n times instead of n → As the sorting elements placed at the top in section sort → where as in bubble sort it will be placed in the bottom.

→

(PTO)

→ Sliding window Approach

3) Insertion Sorting :-

Analogy

Here Insertion means → Inserting an element in a sorted list at a sorted position.

Insertion sort is a simple sorting Algorithm that works similar to the way you sort playing cards in your Hands.

→ The Array is partially placed at the correct position in the sorted part.

Analogy :-

* Insertion sort works similar to the sorting of playing cards in hands. It is assumed that the first card is already sorted in the card game, and then we select an unsorted card. If the selected card is greater than the first card it will be placed at the right side. Otherwise it will be placed at the left side.

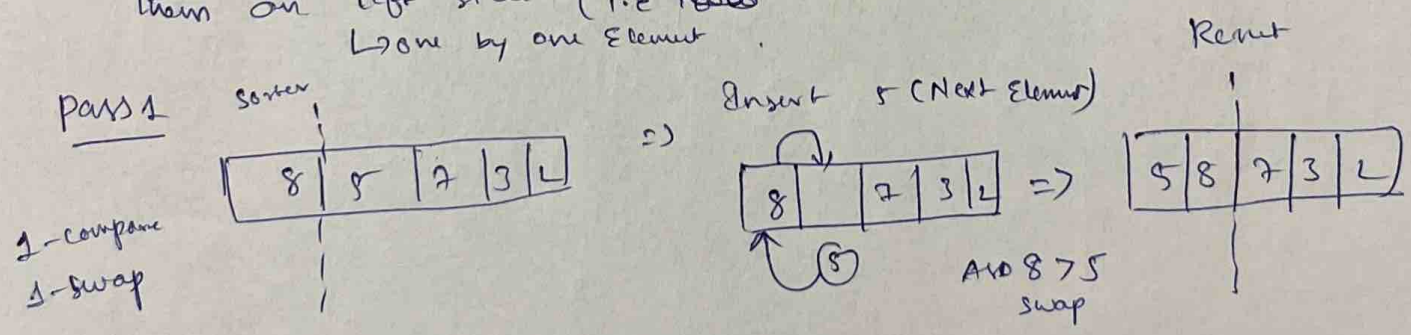
Ex :-

0	1	2	3	4
8	5	7	3	2

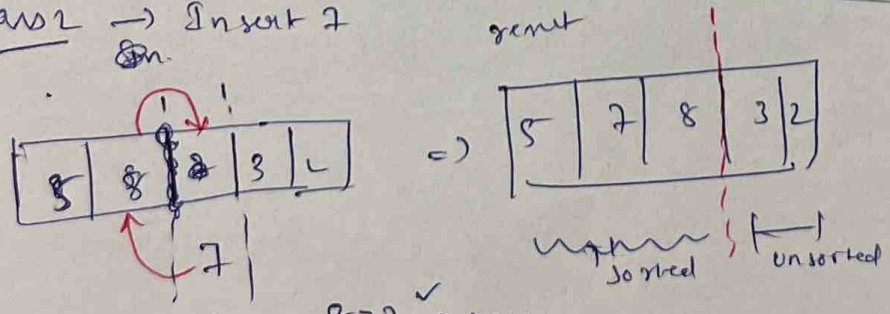
→ Step 1 Assume that the first Element is already sorted, and the remaining Elements are not sorted.

→ Now the remaining Elements will take out and insert them on left side. (i.e towards sorted elements side)
↳ one by one Element.

Pass 1



Pass 2 → Insert 7

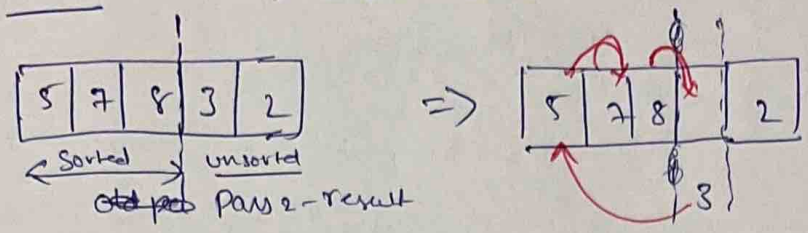


2 - max comparision
2 - max swaps

8 > 7 ✓ ⇒ shift

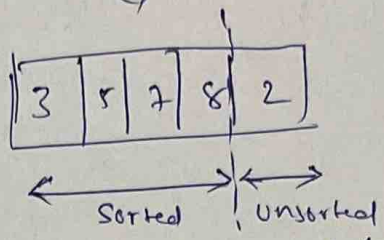
5 > 7 ⇒ False ⇒ so Insert after 5

Pass 3 → Insert - 3 (Next Element in unsorted Array)



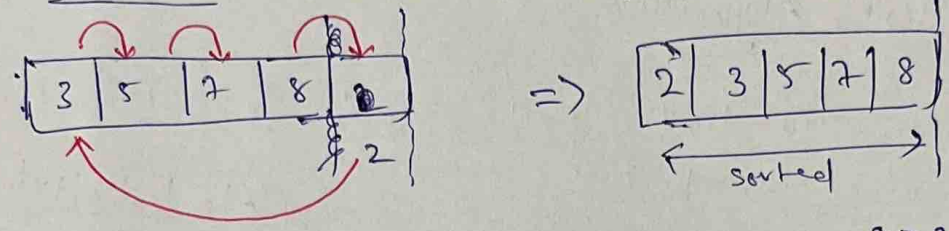
8 > 3 ✓ → swap
7 > 3 ✓ → shift
5 > 3 ✓ → shift

⇓ result



3 - max comparison
3 - max swaps

Pass - 4 → Insert 2 next Element in unsorted Array



4 - max comparisons
4 - max swaps.

8 > 2 ⇒ True ⇒ shift
7 > 2 ⇒ True ⇒ shift
5 > 2 ⇒ True ⇒ shift
3 > 2 ⇒ True ⇒ shift

Observations

* 5 elements → 4 passes

i.e n-elements → (n-1) passes.

* 5 Elements → 1, 2, 3, 4 (max comparisons)
  pass1, pass2, pass3, pass4

  n- Elements → 1, 2, 3, 4 ---- n-1 ⇒ $\frac{n(n-1)}{2}$
  ⇒ $O(n^2)$ comparisons

Time complexity of Insertion sort = no. of comparisons = $O(n^2)$

* n- Elements → 1, 2, 3, ... n-1 ⇒ $\frac{n(n-1)}{2}$
  ↳ max swaps ⇒ $O(n^2)$

5 - Elements → 4 swaps.

→ Intermidate results i.e after 1 pass, 2 pass will not give any useful results like bubble sort.

→ Insertion sort is more compatiable with Linked list than Array as we no need to shift the Memory blocks in linked list → but in Array we need to shift memory blocks.

↳ so Insertion sort is designed for Linked list

→ Insertion sort Algorithm

* As we consider the first Element is already sorted, so loop starts with index 1 instead zero

```
public void
insertion_sort (input) {
    // loop for passes
    for(i=1 ; i<n; i++){
        j = i-1
        x = input [i]
        while (j>-1 && input(j) > x){
            input (j+1) = input(j)
            j--;
        }
        input (j+1) = x;
    }
```

↳ sorted position for insert element

6	7	8	12	14	18	20	
						j	i

x = 10

* say Till Here sorted

→ means i is at last position

→ Now to insert 10 we need to shift

12	14	18	20

↑ 10

So → J starts with i-1 & decrement the loop → until J reaches -1

And

A[J] > x

2-loops

— O(n²)

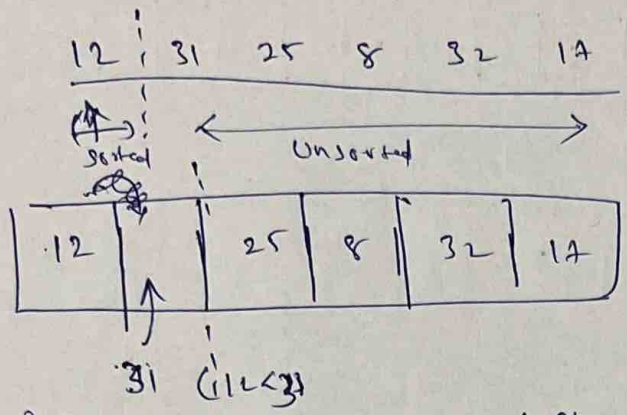
Insertion sort Algorithm

1) If the element is the First Element, Assume that is already sorted & return 1
2) Pick the next Element & store it seperately in a "key"
3) Now compare key with all Element in the sorted Array
4) If the Element in the sorted Array is smaller than the current Element, then move to the next Element, Else shift greater Elements in the array towards the right
5) Insert value
6) Repeat until the Array is sorted.

Insertion Sort => Example 2

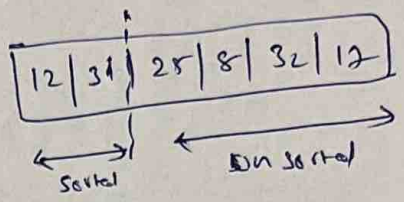
12	31	25	8	32	17

1) Assume Element at position "0" is already sorted.



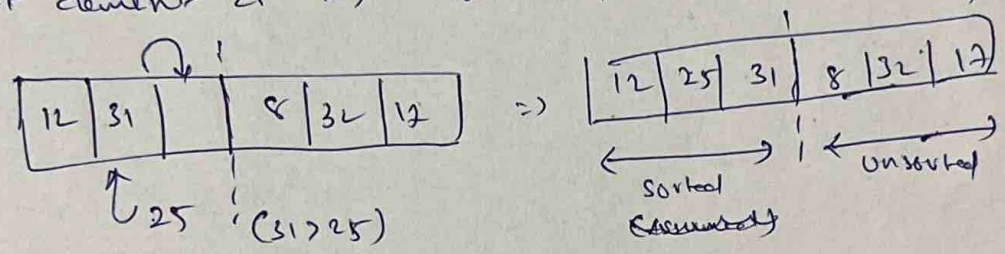
Insert 31

pass-1



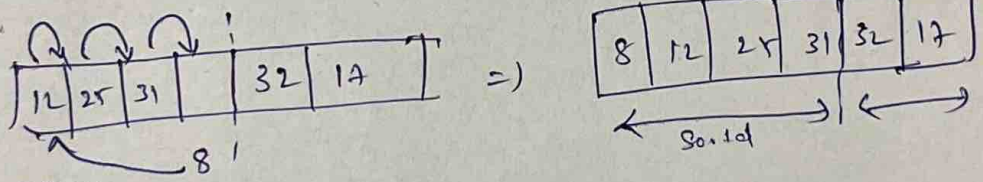
-> next Element 25 (From unsorted Array) -> insert 25 in sorted Array

pass-2



-> Insert 8 (next Element)

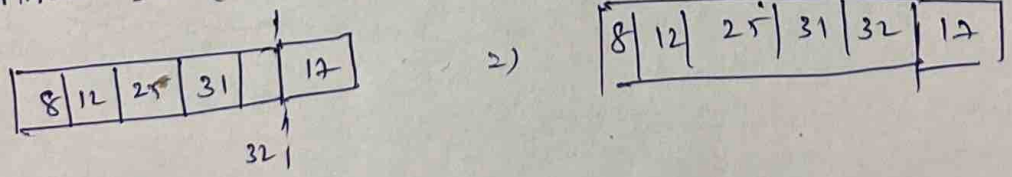
pass-3



31 > 8 => shift
25 > 8 => shift
12 > 8 => shift

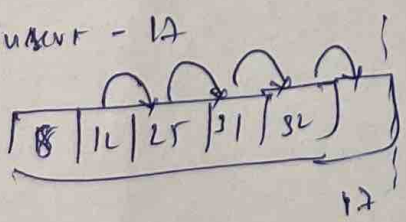
-> Insert 32 (next Element)

pass-4

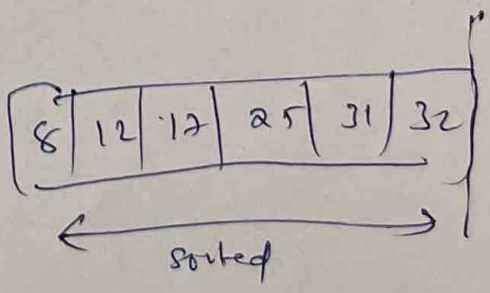


31 > 32 => False
Insert at same place

-> Insert - 17



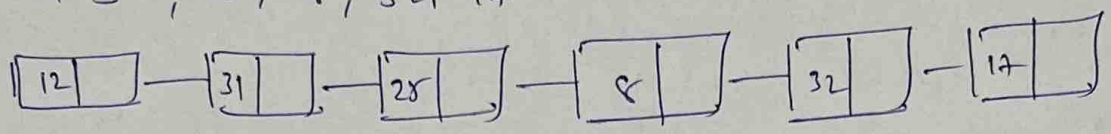
32 > 17 - shift
31 > 17 - shift
25 > 17 - shift
12 > 17 -> False
↳ so Insert after 12



Insertion sort Using Linked List

* Insertion sort more compatiable with linked list as we no need to Move the memory blocks while shifting

12, 31, 25, 8, 32, 17



Counting Sort :-

→ This sort is only used to sort the Integers because the Integers we want to sort will be used as the indices of Another Array

↳ non-compare sorting

1) find out the maximum Element (max) from Given Array

4	2	2	8	3	3	1

max = 8

2) Initilize a new Array of length (max + 1) with all Elements as "0". This Array will be used for ~~sorting~~ storing the Count of the Elements in the Array.

0	0	0	0	0	0	0	0	0

Index → 0 1 2 3 4 5 6 7 8

3) Store the Count of Each Element at their respective index in Count Array.

Occurance of zero → occurance of 2 → occurance of 2 in I/p Array

C =

0	1	2	2	1	0	0	0	1

0 1 2 3 4 5 6 7 8

I/p
4 → 1 time
2 → 2 time
8 → 1 time
3 → 2 times
1 → 1 time

4) Store the cumulative sum of Elements of the Count Array. It helps in placing the Elements into the correct index of the sorted Array.

0	1	2	3	4	5	6	7	8
0	1	3	5	6	6	6	6	7

C[0]+C[1] : 0+(1) ← prev step value

C[1]+C[2] : 1+2 ; C[2]=3

C[2]+C[3] : 3+2 ; C[3]=5

C[3]+C[4] : 5+1 ; C[4]=6

C[4]+C[5] : 6+0 ; ....

↳ i.e C =

0	1	2	2	1	0	0	0	1

0	1	3	5	6	6	6	6	7

5) find the Index of Each Element of the original Array in the count Array. This gives the cumulative count. place the Element at the Index calculated as shown in fig below

I/p Array

0	1	2	3	4	5	6
4	2	2	8	3	3	1

count

0	1	2	3	4	5	6	7	8
0	1	3	5	6	6	6	6	7

$3-1=2$ $(6-1)=5$

O/p

0	1	2	3	4	5	6
1	2	2	3	3	4	8

1st Element → 4 → get cumulative count value from the count Array at position[4] ⇒ c[4] = 6

6-1 ⇒ 5 ⇒ at this position in the O/p Array the value will be stored.

↳ After finding position reduce the count by 1 so that if there any duplicates it can be recorded

2nd Element → 2 ⇒ c[2] ⇒ 3 ⇒ 3-1 = 2 ⇒ op[2] = 2

3rd Element → 2 ⇒ c[2] = 2 ⇒ 2-1 = 1 ⇒ o/p(1) = 2

↑ Previous step reduced by 1

4th Element → 8 ⇒ 2(8) = 7 ⇒ 7-1 = 6 → o/p(6) = 8

→ ~~Counting sort is stable~~

→ If you see / observe the order of Element "2" is not maintained, it mean we broke the stability property

Imp

↳ In simple stability means the order of the Element in the I/p & o/p should be same (for duplicate case)

↳ So How to make Algorithm Stable?

↳ while Adding the o/p → loop from right to left for the I/p Array instead of left to Right

I/p

0	1	2	3	4	5	6
4	2	2	8	3	3	1

← loop From Here

1	2	2	3	3	8

So count sort is stable.

→ What if we have negative values in the Array. → And as we know negative values can not be used as indices. (to refer count Array)

Ex:-

5	2	7	4	-2	2

→ This problem can be solved by assigning the zero index to the smallest value stored inside our initial Array.

In this Array ↳ min value = -2, max value = 7 } so we are going to create an Array that can hold "10" slots in Total and Index "0" will store the number of occurances of "-2"

⇓

But why 10 slots not 9? Because we have -2, -1, 0 thru 1 to 7

Slly Index 9 will store the occurance of "7" (max value)

10 → Slots = 7 - (-2) + 1 = 10

0	1	2	3	4	5	6	7	8	9
-2	-1	0	1	2	3	4	5	6	7

occurances of -2 … occurances of 7

$\boxed{\text{index} = \text{value} - \text{min}}$

Index of 5 = 5 - (-2) => 7

index of -2 = -2 - (-2) = -2 + 2 = 0

-2 => -2 - (-2) = 0
2 => 2 - (-2) = 4
7 => 7 - (-2) = 9
4 = 4 - (-2) = 6
5 => 5 - (-2) = 7

I/p

0	1	2	3	4	5
5	2	7	4	-2	2

Count

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

1+0=0 + + + 4+1=5 + + + +

O/p

⇓

0	1	2	3	4	5
-2			2		2

(Iterate I/p from Right)

Cumulative count

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

1-1=0 3-1=2

I/p

0	1	2	3	4	5
5	2	7	4	-2	2

=>

0	1	2	3	4	5
-2	2	2	4	5	7

O/P

→ To get the count Array size

↳ Here In our case ⇒ size is 10 (Not)

i.e $\boxed{\text{Count Array size} = (max - min + 1)}$

Algorithm (In Java) (Geeks For Geeks)

```
Public void Sort (arr) {
    int min = Array.stream(arr).min().orElse(0);          → O(n)
    int max = Array.stream(array).max().orElse(Integer.max_value)  → O(n)
    int[] o/p = new Int(arr.length)
    // Count Array size (CS)
        CS = max - min + 1
    // Initializing count Array
    int[] CountArray = new int[CS]   → Initially all elements will be zero
    // Count the occurances of store in CountArray
    for (int value : arr) {                               → O(n)
        int index-position = value - min
        CountArray [index position] = CountArray [index position] + 1
    }
    // iterating the CountArray → To store the cumilative sum,
        And
        find the index of Each element of the original Array
        in the count Array
        ~~int arrayIndex = 0~~                            → O(k)
                                                          ↳ Range of element
                                                          arr = I/P Array
        for (int i = 0 ; i < CS ; i++) {
            arr [arrayIndex] = i + min
            arrayIndex ++ ;
            CountArray [i] --
        }
```

Starting cumulative

```
for(int i=1 ; i< countArray.length; i++)
    count[i] += count[i-1];
}
for(int i = arr.length -1 ; i>=0 ; i--){
    index position = value - min
    o/p[index position -1] = arr[i]
    count[index position] --;
}
for(int i=0 ; i< arr.length; i++){
    arr[i] = o/p[i]
}
```

Time complexity $O(k+n)$

n = no. of Elements in the I/P

k = range of I/P

Count sort Observations

1) Time complexities

↳ There are mainly 4 for loops

1st → finding MAX → $O(n)$

2nd → finding min → $O(n)$

3rd → counting the occurances → $O(n)$

4th → Forming o/p → $O(k)$

↳ Range of Elements i.e max - min + 1

$O(n) + O(n) + O(n) + O(k)$

$3O(n) + O(k) \Rightarrow O(k+n)$

Best / Worst / Avg Time Complexity = $O(k+n)$

↑ No matter How Elements are placed in Array, the Algorithm goes through $(n+k)$ times.

2) It is a stable Algorithm.

3) There is no comparison b/w any Elements, so it is better than comparison based sorting techniques.

Drawback ↳ But it is bad if the integers are very large becoz the Array of that size should be made.

Say :- To sort this Array [1, 2, 300]

↳ Then we need (max - min + 1) ⇒ 300 - 1 + 1

⇒ 300 Slots

↳ waste of memory and waste of time in Iteration

↳ To overcome this problem we use Radix sorting

→ Space complexity

$O(k)$ ⇒ larger the range of Elements → larger the Time Complexity.

↑ Range

1cpt

Count Sort Algorithm

RADIX Sort :-

* Radix sort is the linear sorting algorithm that is used for integers.
* In radix sort there is a digit by digit sorting is performed that is started from Least significant digit to the most significant digit.
* The process of radix sort is similar to the sorting of students names according to the Alphabatical order.

Radix Sort process

* first we have to find the largest element (suppose max) from the given array
  - ↳ This is calculated to go through the significant places of all Elements.
* After that, go through one by one Each significant place, Here we have to use stable sorting Algorithm (say insertion) to sort the digits of Each significant place
  - ↳ first take unit place, Tens place, Hundred's place etc...

Ex :-

| 181 | 289 | 390 | 121 | 145 | 736 | 514 | 212 | 39 |

100th	10th	Unit
0	3	9

↳ Above I/p ⇒ Largest no :- 736 ⇒ 3 digits ⇒ so Loop will run for 3 times (i.e 100th place)

Index

0	181
1	289
2	390
3	121
4	145
5	736
6	514
7	212
8	39

Units place ⇒

Bin

0	0	390
1	1	181 121
2	2	212
3	3	
4	4	514
5	5	145
6	6	736
7	7	
8	8	
9	9	289 39

1st pass

After 1st pass the result would be

⇒

0	1	2	3	4	5	6	7	8
390	181	121	212	514	145	736	289	39

Pass - 2 ⇒ 10th place

0	390
1	181
2	121
3	212
4	514
5	145
6	736
7	289
8	39

⇒

Bins	
0	
1	212, 514
2	121
3	736, 39
4	145
5	
6	
7	
8	181 289
9	~~390~~ 390

0	1	2	3	4	5	6	7	8
390	212	514	121	736	39	145	181	289

⇒

0	1	2	3	4	5	6	7	8
212	514	121	736	39	145	181	289	390

Pass - 3

0	212
1	514
2	121
3	736
4	039
5	145
6	181
7	289
8	390

⇒ Here 39 2 digits so consider 0

⇒

0	39
1	121 145 181
2	212 289
3	390
4	
5	514
6	
7	736
8	
9	

⇒

⇓

Result

0	1	2	3	4	5	6	7	8
39	121	145	181	212	289	390	514	736

Sorted

→ 9 Element → 3 Steps

↳ depending no. of digits (d)

Space $O(n)$ no. of Bins

$O(dn)$ d = no. of digits of Max Element

We are coping all the Element and coping back so n times

Algorithm

1) radix sort (arr)
2) max = largest Element in Array
3) d = no. of digits in the largest Element
4) No create a buckets of size 0 - 9
5) for i ⇒ 0 to d

~~Result~~

6) Sort the Array using stable sort (Insertion or counting sort) According to the digits at i^{th} place.

⇓

getting ith place

unit 1) $(A[i]/1)\%10$

10th place 2) $(A[i]/1\times10)\%10$

100th 3) $(A[i]/100)\%10$

↳ 235 $(235/100)\%10$

$2\%10 \Rightarrow 2$

Radix Sort Algorithm

→ Internally uses the count sort

```
Class Radix Sort {
    // Using counting sort to sort the Elements in the basis of significant places.
    Public void countSort (int array[], int size, int place) {
        int output = new int [size+1];
        int max = Array.Stream (array).max().orElse (Integer.MAX_VALUE)
        int min = Array.Stream (array).min().orElse (0);
        int[] count = new int [max - min + 1];
        // Initilize all the values in all index position as zero
        for(int i = 0; i < count.size; i++) {
            count[i] = 0;
        }
        // calculate count of elements
        for(int i = 0; i < size; i++) {
            int value = (array[i] / place) % 10;
            int indexposition = value - min
            count [indexposition]++;
        }
        // Calculate the cumulative count.
        for(int i = 0; i < count.length; i++) {
            count[i] += count[i-1]
        }
        // Place the Elements in sorted order (Sterting from the I/p Array end)
        for(int i = size-1; i >= 0; i--) {
            int value = (array[i] / place) % 10
            int indexposition = value - min
            output [indexposition -1] = array[i]
            count [indexposition] - -;
        }
        for (int i = 0; i < size; i++) {
            array[i] = output [i]
        }
    } // End of count sort function.
```

```
Public void radix sort (int[]array , int size) {
    // Get Max Element
    int max = Array. stream (array) . max() . orElse (Integer. MAX_VALUE)
    // Apply counting sort to sort Elements Based on place value
      for( int place = 1 ; max/place > 0 ; place *= 10) {
          this.Count sort ( array, size , place )
      }
  }
// Driver code
Public static void main ( String args[]){
    int [] data = { 121, 432, 564, 23, 1, 45, 788};
    int size = data. length ;
    Radix sort rs = new Radixsort ();
      rs . radixsort ( data , size)
  }
}// class End.
```

Quick Sort — Divide and conquer method

Analogy :- Suppose in a class, teacher asked arrange themselves in the increasing order of their height. Here there are two Options. 1) A teacher can show their places. ~~like you question~~

→ 2nd option → A teacher can ask the students to arrange themselves so every student will find their place in the sorted order.

→ So, Here the faster Method or Quick Method will be one if the students are finding their place. So yes this is the idea of Quick sort

↳ the shortest person in the class he knows what is his place and he will quietly come and stand in the beginning of a list. nobody has to help him he can check himself. similarly tallest person in the class will stand at the End of the Row. and remaing person b/w short & tallest person will arrange themselves by checking individual heights.

→ Quick sort follows divide and conquer (D&C) Approach. D&C is a technique of breaking down the algorithms into sub problems, then solving the subproblems and combining the results back together to solve the original problem.

Divide :- In divide, First pick a pivot element. After that, partion or rearrange the array into two sub-Arrays such that each element in left sub-array is ≤ to pivot element and each element in right sub-array is > pivot element.

Conquer :- Recursively, sort two subarrays with Quick sort

Combine :- Combine all the sorted Arrays

Ex:-

→ A = | 24 | 9 | 29 | 14 | 19 | 27 |
0 1 2 3 4 5

left ↓ (L); pivot ↑ (P); Right ↑ (R)

> If A[pivot] > A[Right]
> ↳ swap (A[P], A[R])
> And pivot moves to right

→ A[P] = 24 ; A[L] = 24 ; A[R] = 27

↳ It means iwant to find the sorted position of the pivot Element

↳ i.e this Element should come in position such that all Element smaller than pivot Element should come left side and all the greater Element should come right side

→ Now Check A[pivot] ? A[Right] → so Algorithm moves forward one position towards ~~right~~ left

Pass 1

⇒ | 24 | 9 | 29 | 14 | 19 | 27 | (L, P, R) ⇒ | 24 | 9 | 29 | 14 | 19 | 27 | (L, P, R) ⇒ | 24 | 9 | 29 | 14 | 19 | 27 | (L, P, R)

24 > 27 (False)
Move Right Pointer

24 > 19 (True)
~~Move right position~~
swap.

| 24 | 9 | 29 | 14 | 19 | 27 | (L, P, R)

24 < 29 (True

A = | 10 | 16 | 8 | 12 | 15 | 6 | 3 | 9 | 5 |
0 1 2 3 4 5 6 7 8

Algorithm

1) Step 1 :- Choose the pivot Element → it can be any ~~random~~ position in the List

Step 2 :- Take two pointers ~~Conditions are follows~~ one is left (l) and another as Right (r)

Step 3 :- Conditions

1) A[Left] > A[Pivot] && A[Right] < A[pivot] ⇒ Swap (Left, Right)
2) A[left] ≤ A[pivot] ⇒ Increment left Pointer
3) A(Right) ≥ A(pivot) ⇒ Increment Right pointer

Quick Sort Algorithm :-

1) $A = [4, 2, 1, 5, 3, 7, 6]$

Pivot (P) ↑ 4, Left (L) ↑ 2, Right (R) ↑ 6

1) Choose the pivot (P) Element position it can be any element but we choose the begining element for ease
2) Choose the left (L) pointer as the next position to the pivot
3) Choose the Right pointer (R) as the last position
4) Conditions to check at each iterations.

C1 1) * $(A[L] > A[P])$ && $(A[R] < A[P])$

↳ Swap (left, Right)

C2 2) $A[L] <= A[P]$

↳ Increment left pointer

C3 3) $A[R] >= A[P]$

↳ ~~Increment~~ decrement Right pointer.

pass 1

⇒ [4 | 2 | 1 | 5 | 3 | 7 | 6] (P, L, R) ⇒ [4 | 2 | 1 | 5 | 3 | 7 | 6] (P, L, R)

C1 × | C2 ✓ | C3 ✓

[4 | 2 | 1 | 5 | 3 | 7 | 6] (P, L, R) ⇒ [4 | 2 | 1 | 5 | 3 | 7 | 6] (P, L, R) ⇒ [4 | 2 | 1 | 3 | 5 | 7 | 6] (P, L, R)

Swap

[4 | 2 | 1 | 3 | 5 | 7 | 6] (P, R, L)

⇒ Here R > L ⇒ Then Break

5) * When R < L ⇒ swap pivot with Right, that means pivot Element placed at the right sorted position. Elements towards right will be greater, To left will be smaller &

[4 | 2 | 1 | 3 | 5 | 7 | 6] ⇒ [3 | 2 | 1 | 4 | 5 | 7 | 6] (P)

→ [3, 2, 1, 4, 5, 7, 6] (p → 4)

Smaller Element Unsorted — Larger Elements Unsorted.

→ Now consider as the individual Arrays & run the same steps i.e recursion

pass 2

3	2	1
p	L	R

A[R] > A[P] & A[R] < A[P]
(F) — ↳ False (T)

→ no swap

A[L] ≤ A[P] ⇒ True → L++

A[R] ≥ A[P] ⇒ False

3	2	1
p	L	R

3 > 1 && 3 < 1
F (no swap)

3 ≤ 3 → T → L++

1 2 3 ×

swap ⇓

3	2	1
	L	R

(Break R < L)

1	2	3

unsorted | Sorted

pass 3 ⇓

1	2

Already sorted

pass 4

5	7	6
p	L	R

C1 7 ≤ 5 && 6 ≤ 5 (F)

7 ≤ 5 (F) ~~Increment L~~

6 ≥ 5 (T) → Increment R

5	7	6
	R	L

~~Break (R < L)~~

5	7	6
p	R	L

Break

swap p & R

Here Both are in same place & will be in same place

⇓

5	7	6

pass 5 ⇓

7	6
p	R L

6 ≥ 7 && 6 ≤ 7
(F)

6 ≤ 7 → T ⇒ L++

6 ≥ 7 → F.

7	6
p	R L

(Break R L)

↳ | 6 | 7 |

Final

1	2	4	5	6	7

So for Iteration
↳ for full Array of length 9

Quick sort Condition

$C_1 = A[L] > A[P]$ & $A[R] < A[P]$
↳ swap (L, R)

$C_2 = A[L] \le A[P] \Rightarrow L^{++}$

$C_3 = A[R] \ge A[P] \Rightarrow R^{--}$

$C_4 = R < L \rightarrow$ Break
↳ swap (A[P], A[R])

A =

0	1	2	3	4	5	6	7	8
10	5	8	9	3	6	15	12	16
P								

I/P Array	P	L	R	A(P)	A(L)	A(R)	C_1	C_2	C_3	O/P Array
0 1 2 3 4 5 6 7 8 10, 5, 8, 9, 3, 6, 15, 12, 16 P L R	0	1	8	10	5	16	F	L^{++} 1+1	R^{--} 7	10, 5, 8, 9, 3, 6, 15, 12, 16 P L R
0 1 2 3 4 5 6 7 8 10 5 8 9 3 6 15 12 16 P L R	0	2	7	10	8	12	F	T L^{++} 2+1=3	True R^{--} 7-1=6	0 1 2 3 4 5 6 7 8 10, 5, 8, 9, 3, 6, 15, 12, 16 P L R
0 1 2 3 4 5 6 7 8 10, 5, 8, 9, 3, 6, 15, 12, 16 P L R	0	3	6	10	9	6	F	L^{++} 3+1=4	False R	0 1 2 3 4 5 6 7 8 10, 5, 8, 9, 3, 6, 15, 14, 16 P L R
0 1 2 3 4 5 6 7 8 10, 5, 8, 9, 3, 6, 15, 14, 16 P L R	0	4	5	10	3	6	F	L^{++} 4+1=5	F R	0 1 2 3 4 5 6 7 8 10, 5, 8, 9, 3, 6, 15, 12, 16 P L R
0 1 2 3 4 5 6 7 8 10, 5, 8, 9, 3, 6, 15, 12, 16 P L,R	0	5	5	10	6	6	F	L^{++}	F R	0 1 2 3 4 5 6 7 8 10, 5, 8, 9, 3, 6, 15, 12, 16 R L

R > L → Break ⇒ swap a[P], a[R]

10, 5, 8, 9, 3, 6, 15, 12, 16 ⇒

0	1	2	3	4	5	6	7	8
6	5	8	9	3	10	15	12	16

Smaller Elements | P | greater Elements

↳ Now Apply the same process for Smaller Elements Array & greater Elements Array } Recursion

Quick sort Algorithm :-

```
def quicksort (A: list[int])
    qshelper (A, 0, len(A)-1)
    return A

def qshelper (A, start, end):
    # Base-case - for recursion
    if start >= end:
        return

    pivot = start
    left = start + 1
    right = end

    while right >= left:
        if A[L] > A[P] and A[R] < A[P]:
            # swap left & Right
            ~~A[left], A[right] = A[left], A[Ri~~
            A[left], A[right] = A[right], A[left]

        if A[L] <= A[P]:
            left++

        if A[R] >= A[P]:
            Right--

    // outside while loop => swap pivot & Right
    A[P] A[R] = A[R] A[P]

    // Sort left part
    qshelper (A, start, right-1)
    qshelper (A, right+1, end)
```

P = pivot
R = right
L = left

// start = end => only one item in Array

Start < end
↳ Already sorted

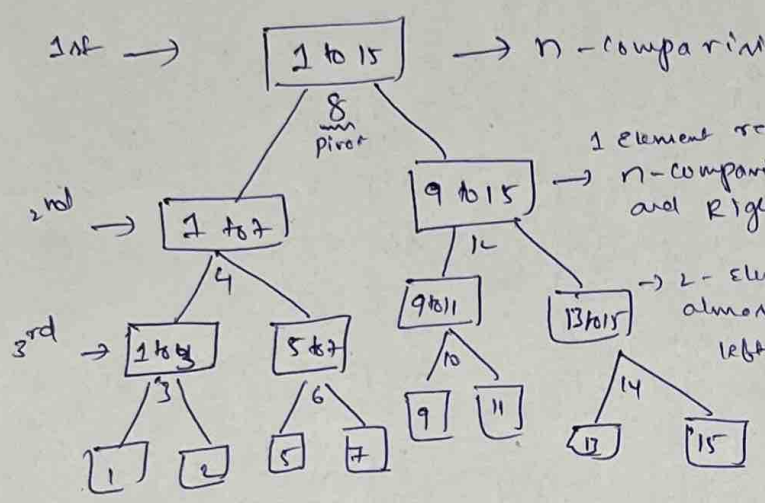
↳ At this point "pivot" Element will be in sorted position

↳ Sort Right part

Quick sort Analysis => Quicksort => Recursive Algorithm

→ Ex:- [4, 5, 6, 1, 3, 7, 2, 9, 8, 10, 11, 13, 12, 15, 14]
(15 - Element from 1-15 in unsorted order.

→ Then suppose the partion is always happening in the middle.



→ 1 element reduced by almost n-comparisons combining left and Right

→ 2- Element reduced but it almost n- comparisons both left & Right

```
Quicksort (l, h)
{
    if (l < h) {
        j = partion(l, h)
left    → Quicksort (l, j-1)
Right   → Quicksort (j+1, h)
```

→ i.e If the partion is always happening at middle for Every pass list will become half i.e n/2/2/2/2 ...
↳ till reaches 1
↳ i.e Equalent to log (n) => levels.

~~↳ for ea~~

→ i.e If we have "n" Element then => log(n) decompositions.
↳ for each decomposition we do Max. no. of Comparison => "n"

↳ $\boxed{\text{Hence Time complexity for the best case} = n(\log(n))}$

↳ Best case is achieved only when the partion is achieved in the middle.

→ Partion of the list in middle can happen randomly that we can not Control. so achieve best case in Quick sort not possible → It may randomly happen

Quick Sort worst case

↳ Suppose i have a list that already sorted, then if we perform quick sort

at the end of ~~iteration~~ 1st pass

→ 2 4 8 10 16 18 14 ⇒ (2) 4 8 10 16 18 14

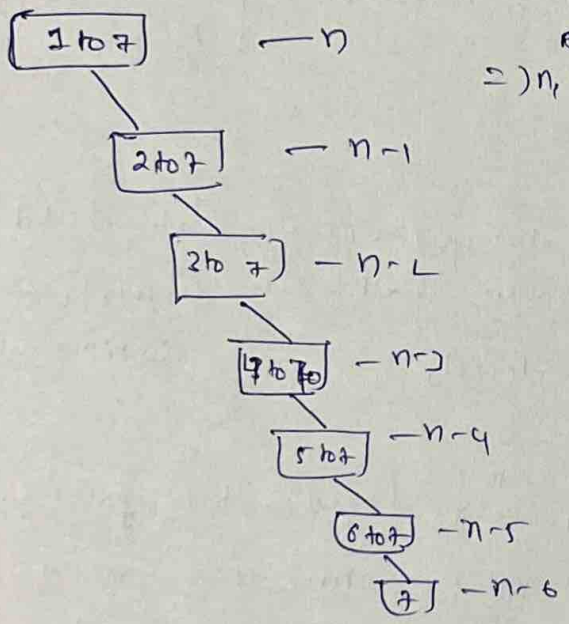
(P, L, R; P, R, L)

2nd pass

→ 2 4 8 10 16 18 14 ⇒ 2 4 8 10 16 18 14

(P, L, R; P, R, L)

→ So if we observe the partion is happening in the same place of (pivot) → i.e begining of the list



⇒ $n, n-1, n-2, \ldots 2, 1 = n(n-1)/2$

⇒ $O(n^2)$

!!

So worst case is $O(n^2)$

→ To improve the worst case - ~~some~~ suggestions

↳ Don't always start the pivot as the 1st Element

↳ Select middle Element as pivot (or)

↳ Select Random Element as pivot

So if the list is already sorted it will become best case

↳ nlog(n) ↳ If we want to choose pivot from middle then swap with first Element and Run the quick sort Algorithm

But still there is a possibility for worst case → we can't ~~out of~~ guarantee

→ Space - complexity

↳ Quick sort is recursive algorithm it does not need any Extra space but it uses stack for recursive results so

log(n) → n

Best case ← log(n) ; ↳ worst case

$\boxed{\log(n) - n}$

Merge sort :-

To understand Merge sort we need to understand Merge

* Merge → Combining two sorted list into a single sorted list

	A	B	Result(C)
i →	2	5 ← j	← k
	8	9	
	15	12	
	18	17	

Procedure

1) Compare A[i] with B[j] if A[i] is smaller then copy that value to C. at C[k]
2) And the element which is copied there you move index to next location. & Increment k → as new value got added.

1)

A	B
i → 2	5 ← j
8	9
15	12
18	17

C

2 ← k [i=0, j=0]

A[0] = 2
B[0] = 5
2 < 5
so copy 2

A	B	C
2	5 ← j	2 ← k (5 < 8)
i → 8	9	5
15	12	
18	17	

A	B	C
2	5	2
i → 8	9 ← j	5
15	12	k → 8 (9 < 9)
18	17	

Increment → i

A	B	C
2	5	2
8	j → 9	5
i → 15	12	8
18	17	k → 9 (9 < 15)

increment j

A	B	C
2	5	2
8	9	5
i → 15	j → 12	8
18	17	9
		k → 12 (15 < 15)

Increment j

A	B	C
2	5	2
8	9	5
i → 15	12	8
18	j → 17	9
		k → 12
		15 (15 < 17)

increment i

A	B	C
2	5	2
8	9	5
15	12	8
i → 18	j → 17	9
		12
		k → 15
		17 (17 < 18)

increment j

⇓ stop ~~Break loop~~ as there are no more elements

A	B	C
2	5	2
8	9	5
15	12	8
i → 18	17	9
		12
		15
		17
		18

→ 18 → Remaining element from A will be copied as there is no element to compare in B

⇒

C
2
5
8 → sorted
9
12
15
17
18

→ i.e If two lists are already sorted then the result is sorted.

Time taken

A	B	
m Elems	n Elems	O(m+n)

→ Suppose we have more than 2-list

A	B	C	D		R
→ 1 (i)	→ 3 (j)	→ 8 (k)	2 ← l		2
6	5	10	4		3
12	9	16	8		4
					4 ...

→ Take smallest among the four and copy (or) merge to result (R)

4 → list merging so we called as "4-way-merging"

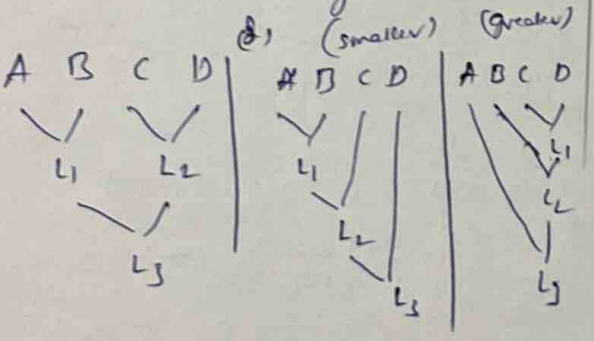
→ For m-Lists
⇓
m-way merging

→ 2 list at a time
↳ 2-way merge

This is the most commonly used ⇒

→ If we have multiple list we can take 2-pair of lists and can be merged

Ex:- A B C D (8) | A B C D (smaller) | A B C D (greater)



→ 2-way merge sort and merge sort are different
↳ 2 way merge sort is a iterative process / Repeating procedure → i.e using loop
↳ merge sort → Recursive Algorithm / procedure
↳ which Internally uses 2-way merge

Merge Algorithm

```
merge (A, B) {
  C = {}
  i = 0; j = 1; k = 0
  m = A.length
  n = B.length
  while (i <= m && j <= n) {
      if (A[i] < B[j]) {
          C[k] = A[i]
          k++; i++
      }
      else {
          C[k] = B[j]
          k++; j++
      }
  } // while loop End
  // copy the remaing elements either A (or) B
  for ( ; i <= m; i++) {
      C[k] = a[i]
      k++
  }
  // copying remaining elements from B
  for ( ; j <= n; j++) {
      C[k] = B[j]
      k++
  }
```

perform this copy untill either of list reaches its last position

no initialization start from its current i value

(Empty) start from current j value

2-way-merge

$A = [\overset{0}{9}, \overset{1}{3}, \overset{2}{7}, \overset{3}{5}, \overset{4}{6}, \overset{5}{4}, \overset{6}{8}, \overset{7}{2}]$

↳ let's say we have a single Array and need to be sorted using Merge process. But 2-way need's two List → so ~~with~~ what can be done is

↳ we Assume that Each Element in an Array is a List, so for below Example there are 8 List contains 1 Element Each. → If ~~there~~ is only single Element means the list is already sorted

↳ i.e Now 8 - ~~single~~ sorted Lists ~~i.e~~ into one single sorted list

→ Now these 8 - List will be Merged Using 2-way merge

9	3	7	5	6	4	8	2

We need one more Array to store Result

1st pass → 3 9 | 5 7 | 4 6 | 2 8 → Merging n Element

But To maintain one list copy Result Back to I/P

2nd pass → 3 5 7 9 | 2 4 6 8 → Merging n Element

3rd pass → 2 3 4 5 6 7 8 9 → n.

* If there are Odd Elements Just simply copy

⇒ 8 Elements ⇒ 3 pass i.e ⇒ $8/2^3 = 1$ ⇒ $\log_2 8 = 3$ pass

n Element ⇒ ($\log n$) passes

$\boxed{\text{So Total time is } O(n \log n)}$

Merge Sort :-

↳ In merge sort we have to break list as smaller problem (i.e only single element)

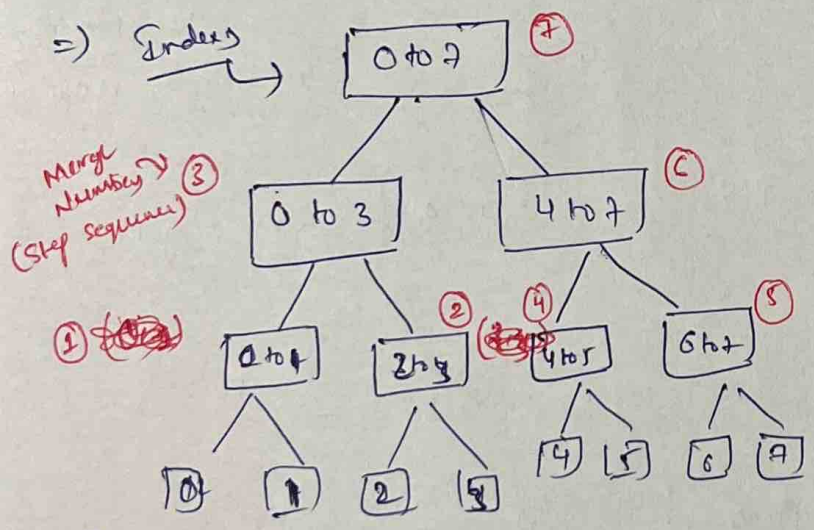
→ Merge sort follows divide and conquer therefore it is a recursive algorithm.

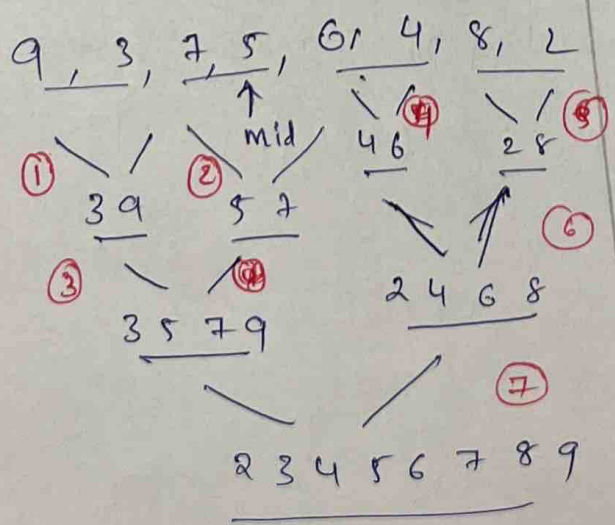
→ Here single Array we are Breaking into Two list by the mid position.

$$A = \begin{array}{cccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \hline 9, & 3, & 7, & 5, & 6, & 4, & 8, & 2 \end{array}$$

↑ Low(l) ↑ mid ↑ High(h)

⇒ Indexes →





Algorithm (merge sort)

```
merge sort (l, h) {
    // l = h => only single Element in List
    // L < H => at least two Elements in List
    if (L < h) {
        mid = (l+h)/2
        left part → merge sort (l, mid)
        Right part → merge sort (m+1, h)
        left part merge → merge (l, mid)
        Right part merge → merge (mid+1, h)
    }
}
```

// without comment

Algorithm

```
merge sort (l, h) {                    → T(n)
    if (L < h) {
        m = (l+h)/2                    → O(1)
        merge sort (l, m)              → T(n/2)
        merge sort (m+1, h)            → T(n/2)
        merge (l, m)                   → n
        merge (m+1, h)                 → n
    }
}
```

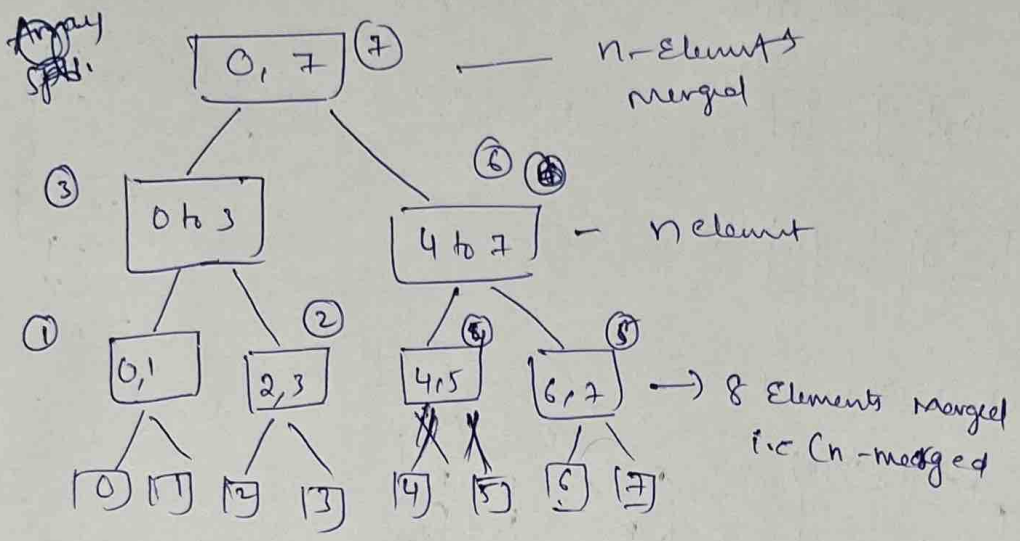
$$T = 2(T(n/2) + n)$$

$$= \Theta(n \log(n))$$

Merge Sort Analys

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

7 - passes
first It will couple left & then Right



Array — splitting

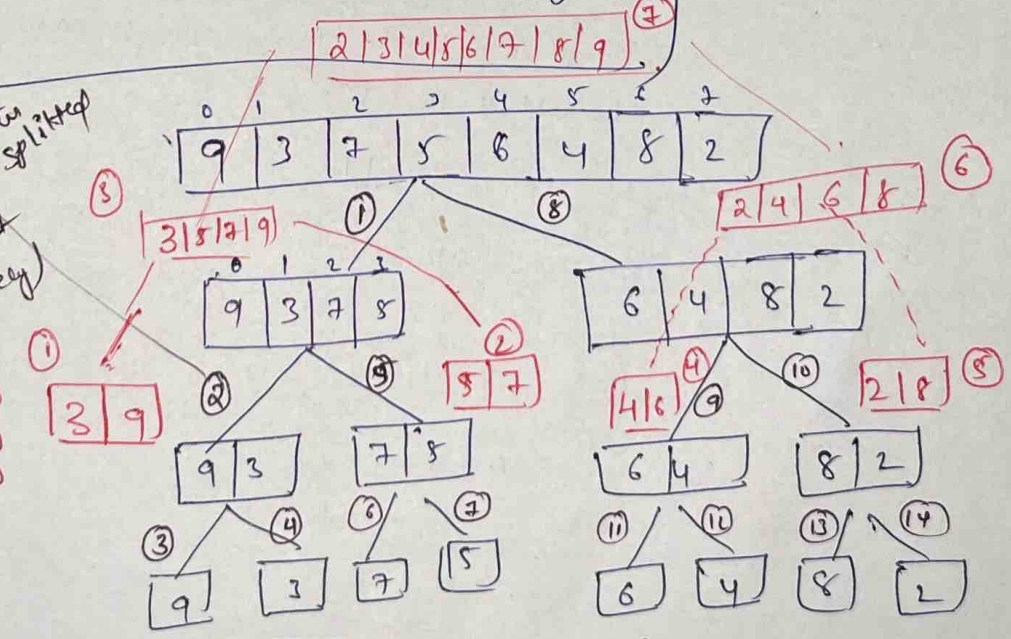
levels

→ for 8 Elements → 3 levels ⇒ $8 = 2^3$ ⇒ $\log(8)$

→ for n Elements → $\log(n)$ - levels

So total time = $n(\log n)$

This is how the single Array splitted i.e left → Right (Recursively)



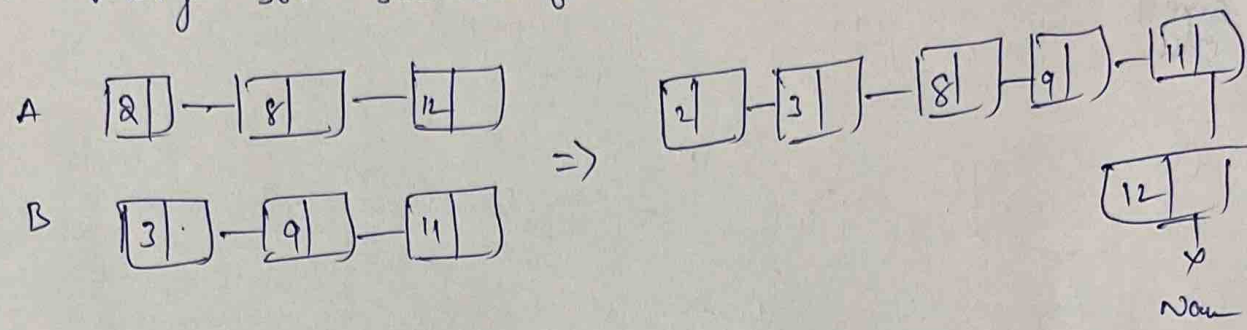
Sorted At this pass

Post-order Traversal Merging

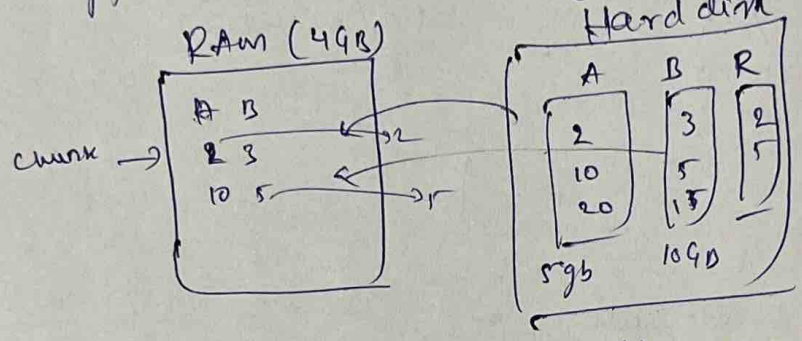
Time complexity $n(\log n)$

Merge sort Advantages

- Supports large size List (using external merge) (No other Algorithm supports.)
- Merge sort suitable for linked list



→ Supports external sorting.



- Say we have two big files A & B in hard disk and we have to merge them files. and store it file (R)
  - To Merge we have to get files into Ram. but the Ram size is smaller than files, so we need to get them into chunks and merge & store in Result File (R)

→ merge sort is stable

- Say we duplicates → then order of duplicates will be maintained.

1st 2nd

8, 6, 4, 3, 8, 5 9

3 4 5 6 8 8 9

1st 2nd

drawbacks of Mergesort

1) Extra space (Not inplace sort) $\Rightarrow (n)$

↳ Inplace Means → sorting should be happened with in the Array.

→ But Merge sort need Extra space for ~~sort~~ storing the Merged result

↳ Extra space is required only in ~~place~~ case of Array not in case of linked list.

↳ so not much suitable for Arrays

2) No - Small - Problem defined

↳ we know in Merge sort, if the Problem is large we break it into Small problems and when it is Small if there is a single element, it is small and we don't do anything there, so it is not having small Problem.

↳ Becoz ~~of~~ it is divide and Conquer and recursive it is Observered that for small size List it is Slower compared to other sorts

↳ Observation $n \leq 15 \Rightarrow$ Merge sort slower

n ↓ no. of Elements → as it wastes lot of time in recursion.

→ So what we do when the list is smaller we route to insertion sort, ~~we~~ ~~but why insert~~

↳ But why Insertion Sort?

↳ Becoz like Merge sort Insertion sort is also stable, ~~As~~ bubble sort is also stable we can use it right? ~~but~~ ~~Insertion~~ Merge is suitable for Linked List and Insertion is suitable for Linked List

3) Recursive

↳ All recursive will use "stack" in the memory, max size of the stack depends on height of the Tracing tree ($\log(n)$)

↳ so space complexity

$$\boxed{n + \log(n)}$$

Bucket Sort Algorithm

* Bucket sort algorithm divides the unsorted Array into several groups termed buckets.

* Each buckets sorted by using any of the suitable algorithms (or) recursively applying the same bucket algorithm.

Ex:-

0	1	2	3	4	5	6	7
12	9	37	29	41	35	21	22

Range of each bucket = (max - min) / n (n = no. of buckets

range = 47 - 9 / 5 = 7.6 ≃ 8

RADIX SORT

* Radix sort is the linear algorithm that is used for integers