

Handwritten Notes

Data Structure & Algorithm

By Siddharth Singh

C & C++ Concepts

- 1) Arrays
- 2) Structure
- 3) Pointers
- 4) Reference
- 5) Parameter
- 6) Classes
- 7) Constructor
- 8) Templates

→ Array

collection of similar data.

Variables

```
int main()
```

```
{ int A[5] = {2,3,5,7,9}; }
```

```
int B[5] = {7,8,9,10,11};
```

```
for(i=0;i<5;
```

```
{
```

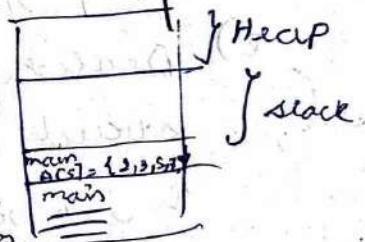
```
cout << A[i];
```

```
}
```

```
}
```

printing
array

This is stored in Main memory as code
sector



→ Structure

Collection of dissimilar data by one name
defined by user

Defining:

```
struct Rectangle;
```

```
{ int length; → 2 byte (assume) length  
int breadth; → 2 byte (assume)  
};
```

∴ This structure will consume 4 byte.

How to declare variable of this structure,

②

```
int main()
{
```

```
    struct Rectangle a;
```

```
    struct Rectangle r={10,5};
```

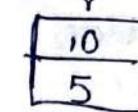
How to access?

(.) operator is used,

```
printf("%d", r.length*r.breadth);
```

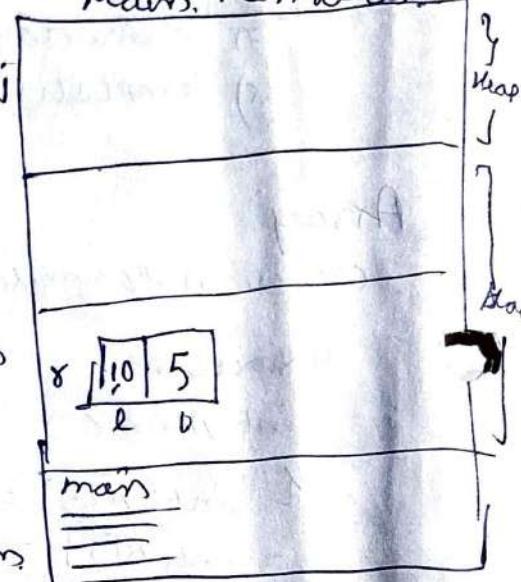


length // Declaration
+ address



// Declaration + init.
address

Mains. Main() { }



Other examples of Structure:

1) Complex No

at 16

∴ complex no is defined by two vars
variables.

struct Complex

```
{ int real; → 2 bytes
    int img; → 2 bytes
}; → 4 bytes.
```

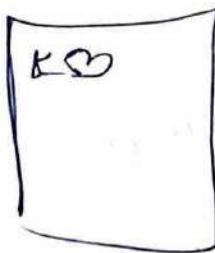
2) Dealing with student information.

struct Student

```
{ int roll; → 2 bytes
    char name[25]; → 25 bytes
    char dept[10]; → 10 bytes
    char address[50]; → 50 bytes
}; → 87 bytes
```

struct Student s;
s.name="John";

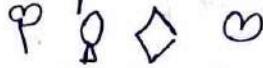
③ Card:



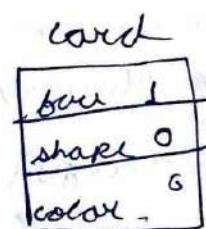
③

face - 1, 2, ..., 10, J, Q, K

shape - 0, 1, 2, 3



color - 0, 1
↓
place



struct Card

{
int face;
int shape;
int color; →
} → 264B
6 myte

int main () {

struct card c;

c.face = 1;

c.shape = 0;

c.color = 0;

10 other ways to initialize

struct card c{1,0,0};

To get 52 cards -

int main.

{

struct card deck[52] = {{1,0,0}, {2,0,0}, ...}

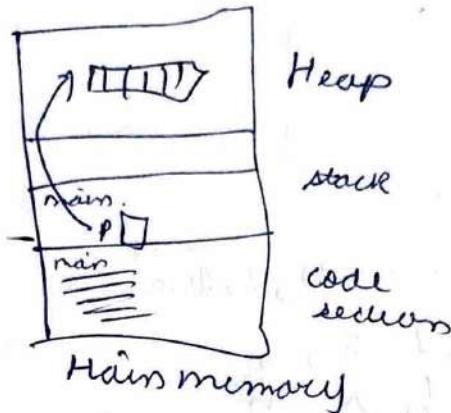
{1,1,0}, {2,1,0},

// way of structuring

①

Pointers:

C
P
V



Program can access code sections & stack of Main Memory. It cannot access Heap memory.
 ∴ To access Heap Memory → Pointer is needed

(Heaps are external to programs.)

∴ A pointer might be accessing monitor files, internet & other external things.

Why are Pointer Useful:

- 1) Accessing Heaps
- 2) " Resources
- 3) Parameter passing.

```
int main()
```

{

```
    int a = 10
```

```
    int *p;
```

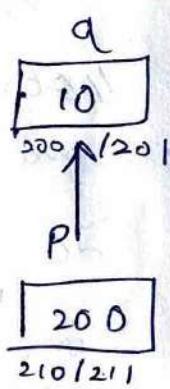
```
    p = &a;
```

```
    printf("%d", a); → 10
```

```
    printf("%d", *p); // Reference is called as.
```

Pointers has 4 bytes in 32-bit.

8 bytes in 64-bit.



5

When you will declare variable it will be in stack.

Accessing Heap memory by Pointer

```
#include < stdlib.h >
```

```
int main()
```

```
{ int *p;
```

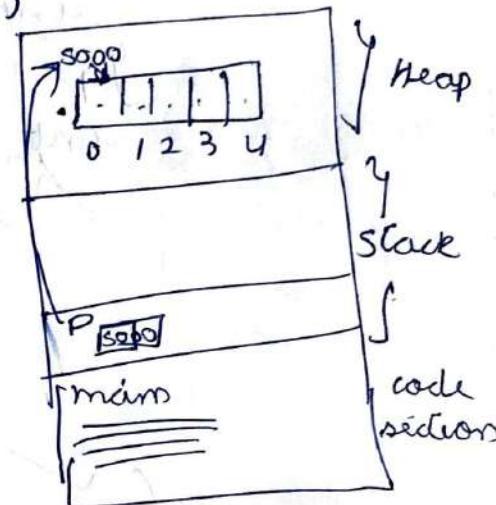
```
p = (int*)malloc(5 * sizeof(int));
```

\downarrow
size

case

This is dynamic
~~void~~ void *

create memory in Heap



If we use .con do some thing as

```
p = new int [5];
```

Reference: (Can't name variable)

```
int main()
```

```
{
```

```
int a=10;
```

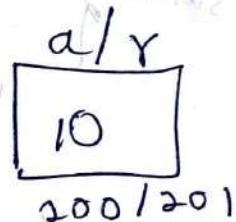
```
int &r=a;
```

```
cout << a ; -> 10
```

```
r++;
```

```
cout << r ; -> 11
```

```
cout << a ; -> 11
```



Use of Reference

→ use for parameters passing

→ To write small functions

⑥

Pointer to Structure

Points same size area of int.

Accessing structure using Pointer.

struct Rectangle

```

    {
        int length; 264th.
        int breadth; → 264th
    }

```

y;

int main()

struct Rectangle r = { 25, y};

struct Rectangle *p = &r;

r.length = 15;

some of ✓ (* p).length = 20; // Here * p.length = 20
~~will be & wrong~~

✓ p → length = 20; " accessibility of
~~is more than *~~

→ To create dynamically variable of struct
 area - (in Heap)

struct Rectangle

{ int length;

 int breadth;

};

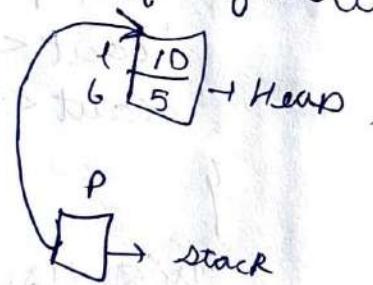
int main()

{ struct Rectangle *p;

p = (struct Rectangle *) malloc (size of (struct
 Rectangle));

p → length = 10;

p → breadth = 5;

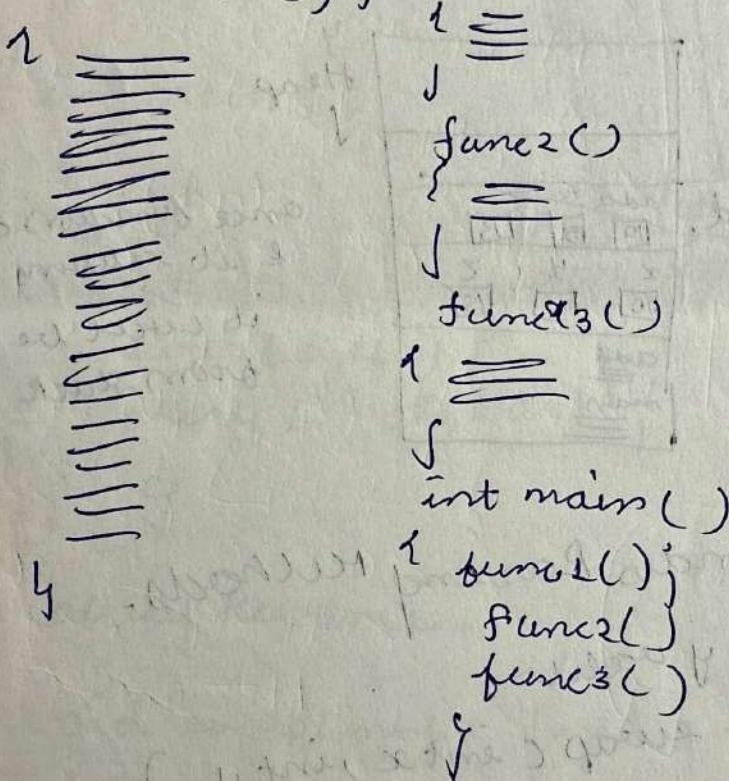


Functions -

- 1) ~~What are Functions~~
- 2) Parameter Passing,
 - 1) Pass by Value
 - 2) Pass by Reference (in C++)
 - 3) Pass by Address

~~Grouping Instructions → Function~~

Monolithic Programming Modular / Procedural Programming
 int main() functions



(8)

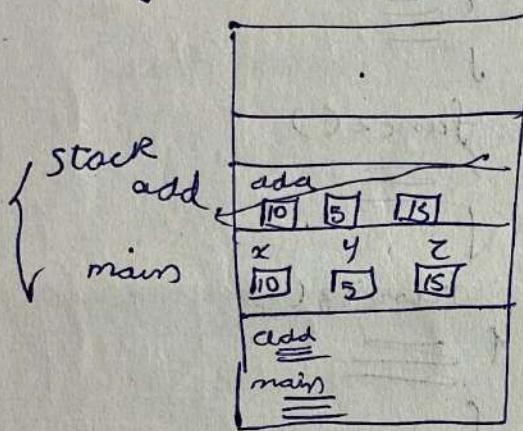
int add (int a, int b) → Prototype
Signature of Functions

```
{ int c
  .c = a + b;
  returns(c); }
```

formal parameters,
declarations

int main()
{ int x, y, z;
 x = 10;
 y = 5;
 z = add(x, y); } → Actual Parameters.

printf("sum is %d", z);



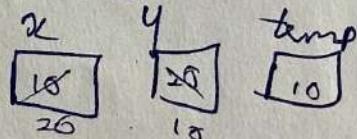
once function add is called
& its return value is
it will be then deleted
from stack

Parameters Passing Methods.

(a) By value

void swap (int x, int y)

```
{ int temp;
  temp = x;
  x = y;
  y = temp; }
```



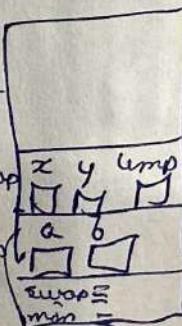
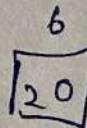
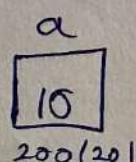
Formal & parameters
are swapped.

int main()

```
{ int a, b;
  a = 10;
  b = 20;
```

swap(a, b);

printf("a.d %d b.d %d", a, b);



200/201

202/203

(8)

No change in Actual Parameters
In call by value only Formal Parameters are modified.

II Call by Address

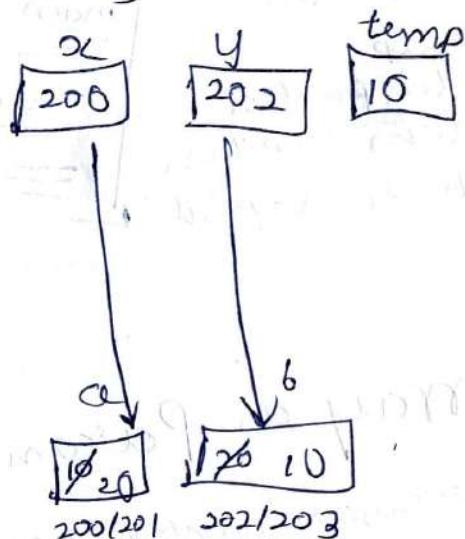
Actual Parameters will also be modified.

void swap(int & x; int & y)

```
1 int temp ;  
temp = &x;  
&x = &y;  
&y = temp;
```

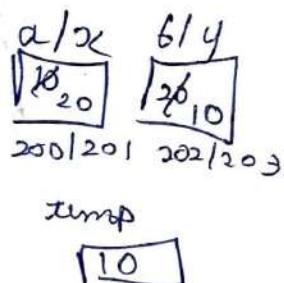
2 int main()

```
{ int a,b ;  
a=10  
b=20  
swap(&a,&b)  
printf("%d,%d",a,b);}
```



Call by Reference

void swap(int & x, int & y)
// code as I

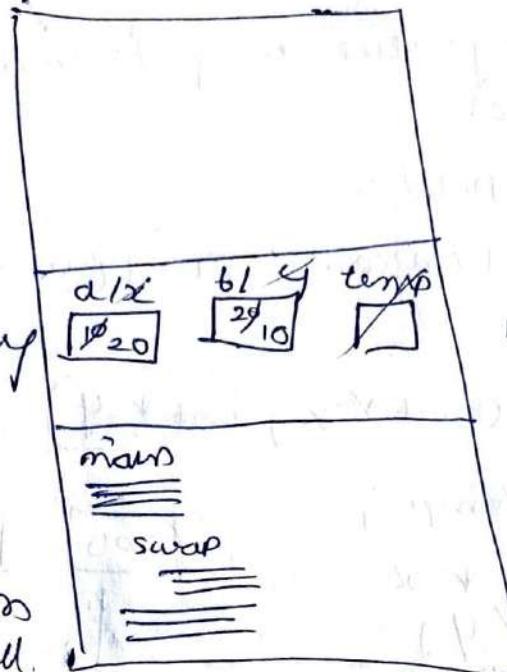


* Here machine code is Non-separable while source code is procedural.

10

Here

temp variable
will be formed
in main memory
because
machine code
of swap
will be pasted
in main function
at the time of call.



main.

when swap
function will
end its task
so x, y & temp
will be removed

Array as Parameter

arrays cannot be passed by value to all,
can be passed only by address

void fun(int * A[] , int n)
 ^ Points to array,

1

~~A[0] = 23;~~ int i;
for (i=0; i < n; i++)
 y points ("y.d", A[i]);

int main ()

{ int a[5] = {2, 4, 6, 8, 10};
 fun (A, 5)

g



0	1	2	3	4
2	4	6	8	10

(7)

* can also be used [] denotes that it is compulsory
array

```

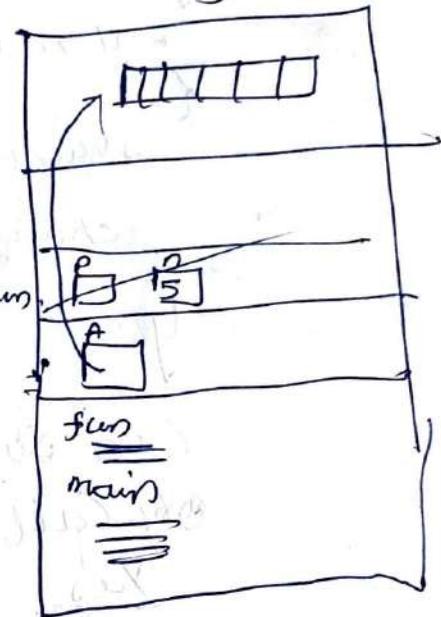
int C I funo( int n )
{
    int * p
    P( int * ) malloc( n * size of int )
    returns ( p );
}
    
```

↓

```

int main( )
{
    int * A;
    A = funo(5);
}
    
```

↓

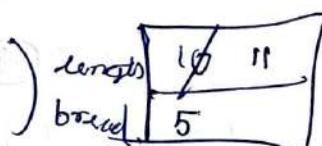


Q7 Structure as Parameters

call by value

```

int area( struct Rectangle r1 )
{
    r1.length++ ;
    return( r1.length * r1.breadth );
}
    
```

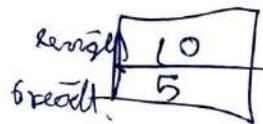


int main()

```

{
    struct Rectangle r = {10, 5} ;
    printf("%d", area(r));
}
    
```

r

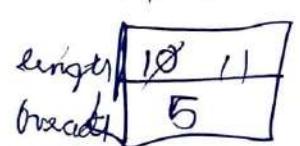


call by Reference

```

int area( struct Rectangle & r1 )
{
    Rest same .
}
    
```

r, r1



(12)

void changelength(struct Rectangle *p, int &l)

{

$p \rightarrow \text{length} = l;$

int main()

struct Rectangle r = {10, 5}

changelength(&r, 20);

Y



10	20	length
5	6	breadth

Can Structure Having Array can be Passed
① Call by Value?

Yes

① → Struct List
int A[5];
int n;

② void fun(struct Test t1)

$t_1.A[0] = 10$

$t_2.A[1] = 9$

Y

A	10	9	t ₁
t ₂	2	4	6
n	5	10	11

③ → int main()

{

struct Test t = {1, 2, 4, 6, 8}

fun(t)

j

2	4	6	8	10	→ A
5	7	9	11	13	

t

i.e. array is copied as an array in t,

Structures & Functions

struct Rectangle

```
{ int length();  
int breadth;
```

};

void initialize (struct Rectangle *r, int l, int b)

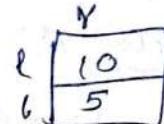
call by address

{

r->length = l

r->breadth = b

y

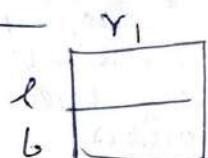


call by value

int area (struct Rectangle r)

```
{ returns r.length * r.breadth;
```

y



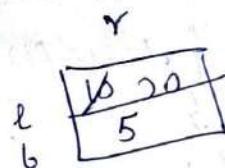
call by address

void changelength (struct Rectangle *r, int l)

{

r->length = l

y



int main()

{

struct Rectangle r

int * a;

initialize (&r, 10, 5) ;

a = area(r);

changelength (&r, 20)

returns 0

}

This is the Best style of coding in C language.

Now transforming this code into ~~or~~ classes and Constructors in C++

(14)

Class & Constructor

In class datatypes and functions all are part of class

1.

class Rectangle

{ private :

int length;

int breadth;

public :

Rectangle (int l, int b)

name
of class

{ length = l ;

breadth = b ;

✓ functions not
constructors
void initialize (int l, int b)

OR

{ length = l
breadth = b

constructor will be used
for initializing
at the time
of the class
creation

it has same
name as that
of class

int area ()

{ returns length * breadth

y

void changelength (int l)

{ length = l ;

};

int main ()

{

 Rectangle r(10,5) // object

 r.initialize (10,5); // function of object

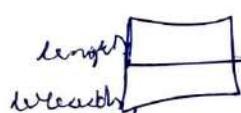
 r.area();

 r.changelength (20);

y

one we create object we get

r



initialize ()
area ()
changelength ()

#include <iostream>

using namespace std;

class Rectangle

{ private:

int length;

int breadth;

public:

constructor [Rectangle() { length = breadth = 1; }]

Rectangle(int l, int b) ;

Facilitator

[int area();]

uses datamember
of class

[int perimeter();]

getters [int getLength();] { returns length; }]

mutates or setters [void setLength(int l) { length = l; }]

destructor ~Rectangle();

};

// Now we define functions outside the class

Rectangle:: Rectangle(int l, int b)

{ length = l;

breadth = b;

};

int Rectangle:: area()

{ returns length * breadth; }

int Rectangle:: Perimeter()

{ returns 2 * (length + breadth); }

Rectangle:: ~Rectangle()

};

};

This definition of function outside the
class is called scope of resolution

(16)

• int main()

```
{
    Rectangle r(10, 5);
    cout << r.area();
    cout << r.perimeter();
    r.setLength(20);
    cout << r.getLength();
```

y

Template Class

To make the class generic, we use templates
ex:

template <class T>

class Arithmetic

{
private:

T ~~int~~ a;

T ~~int~~ b;

public

Arithmetic (~~int~~ a, ~~int~~ b)

T ~~int~~ add();

T ~~int~~ sub();

y;

template <class T>

Arithmetic<T> :: Arithmetic (~~int~~ a, ~~int~~ b)

this → a = a; // This is due to some variable ^{name} _{was}
since it is a pointer, we have to use this.

this → b = b;

y

template <class T>

T ~~int~~ Arithmetic<T>:: add()

{
T ~~int~~ c

c = a + b

return c;

y template <class T>

I template has scope
from { to }

```

Tint Arithmetar<T>:: sub()
{
    T int c
    c=a-
    cout<<a<<c;
}

```

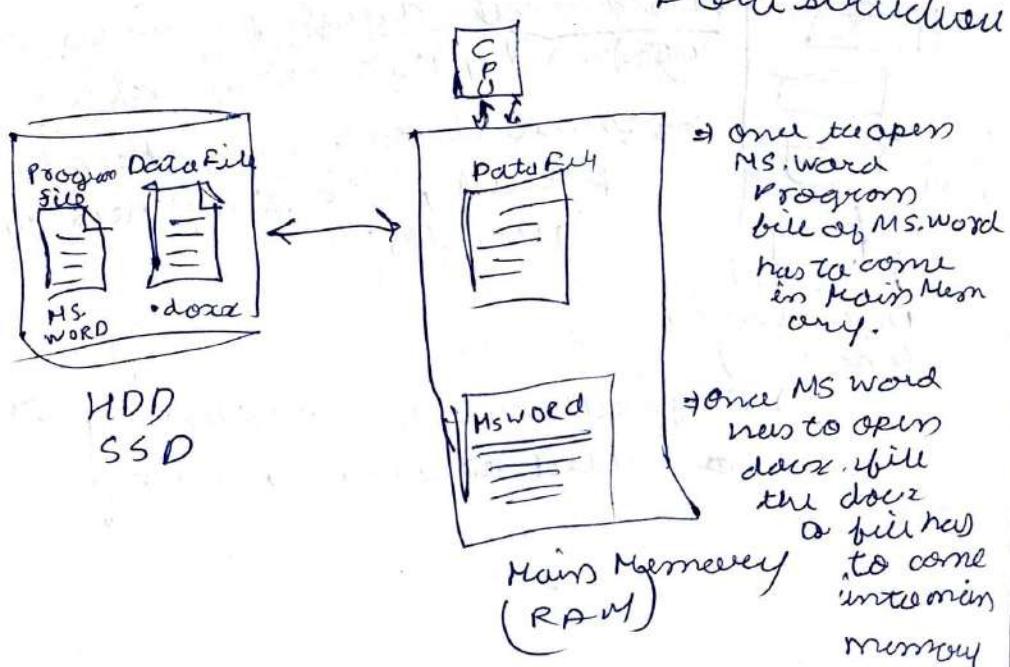
```

int main() {
    Arithmetar<int> ar(10,5);
    cout<<ar.add();
    Arithmetar<float> ar(1.5,1.2);
    cout<<ar.add();
    return 0;
}

```

Data structures

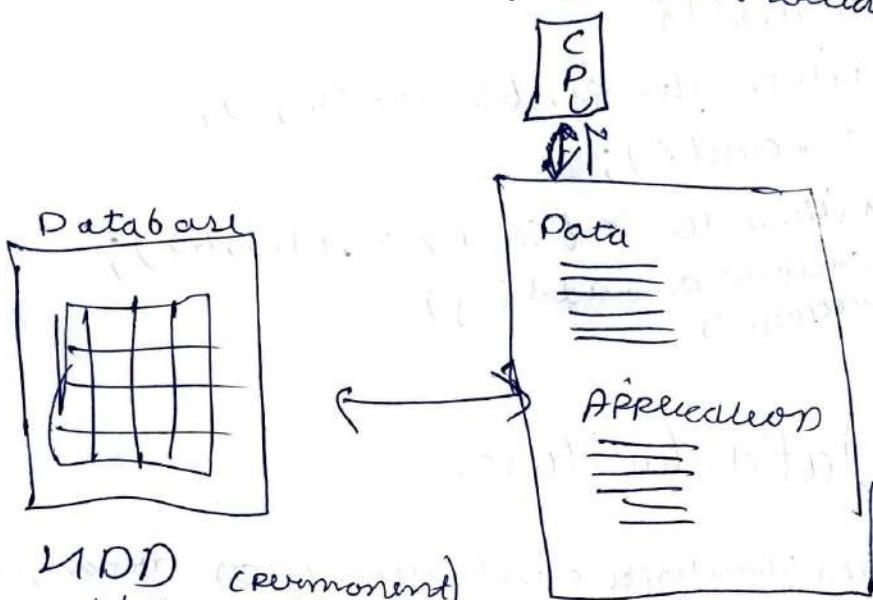
- ⇒ Data structures can be defined as arrangements of collections of data items so that they can utilize efficiently, operations on that data can be done efficiently.
- ⇒ So during execution of program new the program will manage data inside the main memory and perform operations is Data Structure



- ⇒ Data structures are formed in main memory during the execution time of program
- ⇒ Arrangement of data's in main memory is called datastructure

Databases:

when data is stored or collected.



HDD (common)
storage storage

arrangement of data
in some relations

model in HDD

storage set that

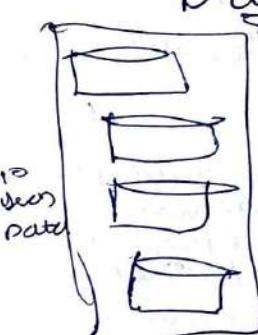
is easily accessed by application

arrangement of data in main
memory is Data structure

Data warehouse

Historical data's previous data's (not in use)
is kept in access of disk

⇒ These data's are helpful for making
polices, steering new trend, analysing
growth



Data Warehouses
(warehouse)

⇒ Algorithms for analysing these data's are
called data mining algorithms

Big Data

huge Data accumulating in Internet.

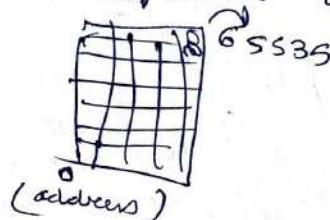
Study of storing & analyzing this large data is called Big Data.

Stack vs Heap Memory

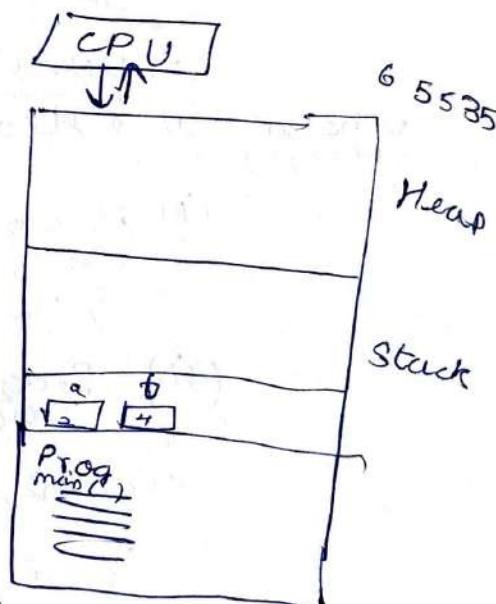
→ Topics we discuss

- 1) About the Main Memory
- 2) How Programs use memory
- 3) Static allocation
- 4) Dynamic allocation

* 8GB or 4GB RAM are divided into parts called segments of 64KB



```
void main()
{
    int a - 264KB
    float b - 464KB
```



* This is static allocation
" how much memory to be allocated to a & b is decided at the time of compilation

* Partition of memory that is given to the function is called activation part of that function



Static Memory Allocation in Stack

void fun2 (int i)

{ int a

\equiv

g

void fun2 ()

{ int x;

fun2(); } (b)

\equiv (I)

fun2
(b)

g

(a) fun2
creation

main

void main()

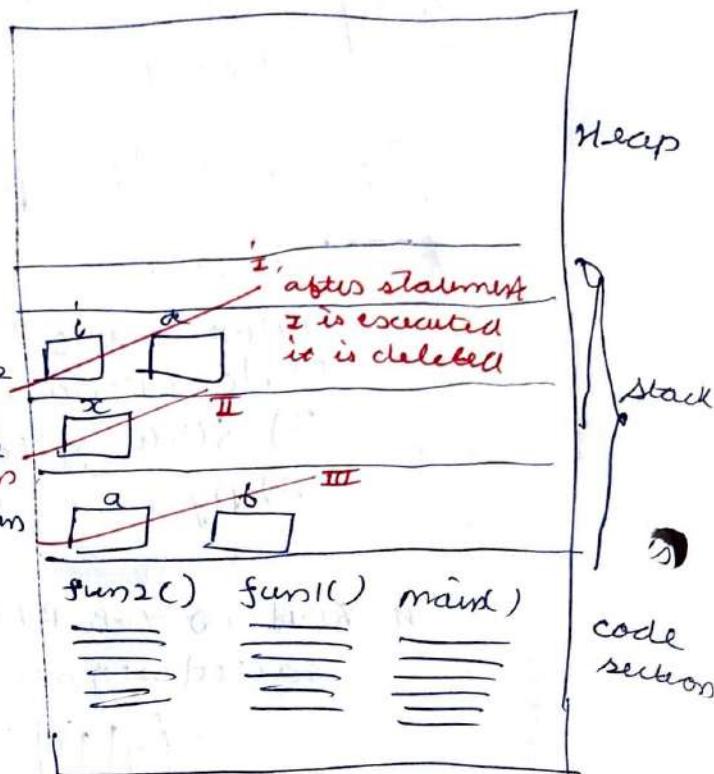
{

int a;
int b;

\equiv

fun1(); } (a)

\equiv
g
III



∴ On stack activation records are created and deleted after call. is over automatically

Dynamic Allocation

What is Heap?

(i) ~~Heaps~~ Randomly arranged over each obj
(unorganized memory)

(ii) Heaps should be used as a resource when required take the memory & when you don't require release the memory
(iii) Programs cannot access heap memory directly.

void main()

{ int * p; byte

C++

C

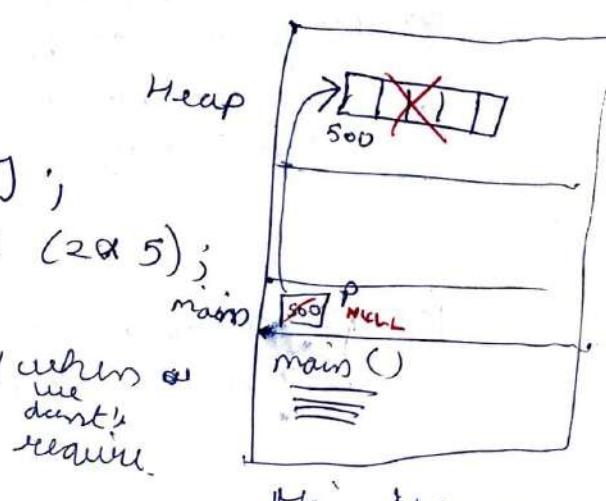
p = new arr[5];

p = (int *) malloc (2 * 5);

\equiv

delete [] p; // when we

p = NULL;



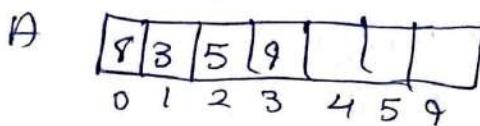
Main Memory

(2) Types of Data Structures

- 1) Physical DS
- 2) Logical DS

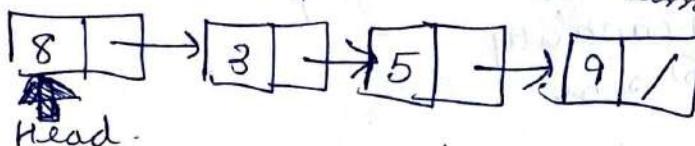
Physical DS

- 1) Array



- contiguous allocation
- fixed size (static)
- can be created in stack or heap
- we use this when we know maximum no. of elements

- 2) Linked List



- variable length / size
- always created in heap
- head can be created in stack

Logical DS:

linear [1) Stack LIFO (Last in First out)
2) Queue FIFO (First in First Out)

non linear [3) Trees

ear ← [4) Graph

Tabular [5) Hash Table

These DS are used in algorithms & for implementing these DS we need array or link list. as combinations of array & link list.

→ ADT (Abstract Datatype)

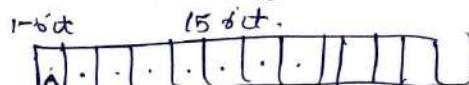
what is ADT?

Datatype is divided in two parts

(i) Representation of Data

(ii) Operations on Data.

representation of Data in datatype



string 2bytes

(22)

Operations allowed

+ - * / % ++ --

some operations

→ int, float are primitive data type.

→ ADT is related to OOPS

→ Abstract means ~~not~~ hiding internal details

ADT

→ List

→ 8, 3, 9, 4, 6, 10, 12
Index 0 1 2 3 4 5 6

Data representation of Data

→ space for storing elements.

- 2) capacity
- 3) size

1) Array
2) linked list

operations on list:

add(x)
remove()
search(key)

Operations on data

When object is created in OOPS we did not know the & how is data represented, it is Abstract datatype.

Some Operations on List

- add(element) / append(element)
- add(index, element) // adding element at given index.
 ↳ insert(index, ele)
- remove(index)
- set(index, ele) // replace(index, element)
- get(index)
- search(key) // contains(key)
- sort() // to sort list

Time and Space Complexity.

ex :-

A $\boxed{2|5|9|6|4|3|5|8|7|2}$ space complexity $O(n)$
 n elements.

\Rightarrow Suppose we are adding all elements then we have to go through all n elements \therefore time complexity will be n i.e $O(n)$

\Rightarrow If we are comparing each element with next \therefore it will take i loops

$O(n^2)$ comparing each element with next.

it will look like $f(i) = \sum_{j=0}^{n-1} f(j)$

$\sum_{j=0}^{n-1} f(j) = \sum_{j=0}^{n-1} j$

$\sum_{j=0}^{n-1} j = \frac{n(n-1)}{2}$

\therefore

\Rightarrow If we are doing program such as

then $(n-1) + (n-2) + (n-3) + \dots + 1$

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

$\boxed{2|3|4|5|7|8|9|}$

2 to take value compare
 3 to take value compare
 4 to take value compare -

: Degree of polynomial : 2

$$\underline{\underline{O(n^2)}}$$

\Rightarrow for ($i = n$; $i > 1$; $i = i/2$)

$\sum_{i=1}^n$

\downarrow
SR

$i = n$

while ($i > 1$)

$\sum_{i=1}^n$
 $i = i/2;$

$\log n$

if n is initially
 then i is
 always expo-
 nentially decreasing
 to 1.

ex : performing operations on
 middle iteration

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

24)

ex 2

example 2:

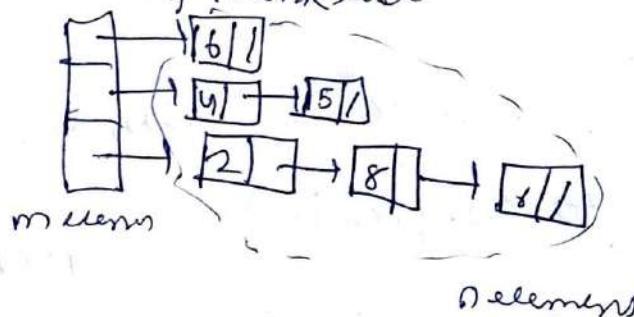
	1	2	3
0	8	3	5
1	7	6	9
2	6	5	8
3	10	4	2

 $n \times n$ If we make program of adding all elements it will be $O(n^2)$

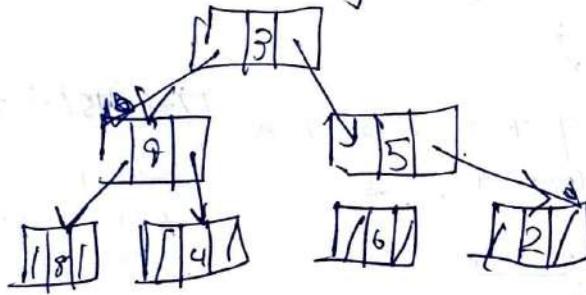
example 3:

Space complexity ~~is~~ $O(n^2)$

array + linked list

 $\therefore O(m+n)$

example 4: Binary tree



along height if we are processing.
 $\log_2 n$

along Processing all elements one by one
 $O(n)$

Space comp

example 4:

Space complexity : $2n$ i.e $O(n)$

Time & Space Complexity

⇒ void swap(^{int, int} x, y)

{ int t;

$t = x;$ — 1 unit

$x = y;$ — 1 unit

$y = t;$ — 1 unit

$$\underline{\quad}$$

$$f(n) = 3n^0$$

$$\therefore O(1)$$

=

int sum(^{int} A[], ^{int} n)

{ int s, i, j;

$s = 0;$ — 1 unit

for ($i=0, i < n; i++$) \uparrow

{ $s = s + A[i];$ \uparrow

return s; \downarrow

summing. $2n+3$
 $O(n)$

⇒ void ~~*~~ ^{int} add(^{int} n)

{ int i, j;

for ($i=0; i < n; i++$) $\rightarrow n+1$

{ for ($j=0; j < n; j++$) $\rightarrow n(n+1)$

$C[i][j] = A[i][j] + B[i][j]$ $\rightarrow n \times n$

$$f(n) = 2n^2 + 2n + 1 \\ \Rightarrow O(n^2)$$

⇒ fun1() $\Rightarrow O(n)$

{ fun2(); $\rightarrow n$ (not n^2)

fun2() $\neq O(n)$

{ for ($i=0; i < n; i++$)

=

Recursion:

- 1) What is Recursion.
- 2) Example of Recursion.
- 3) Tracing Recursion.

Stack Used in Recursion.

Recursion: If a function is calling itself

Type fun (param)

1. if (<base condition>) ↗

2. ↓
3. sum (param);

↓

example:
⇒ void fun1(int n)

1. if ($n > 0$)
2. printf("%d", n);
 fun1($n - 1$);

void main()
1. int x = 3;

 sum(x);

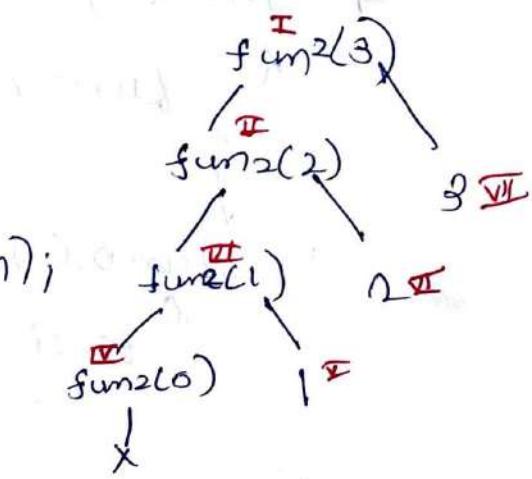
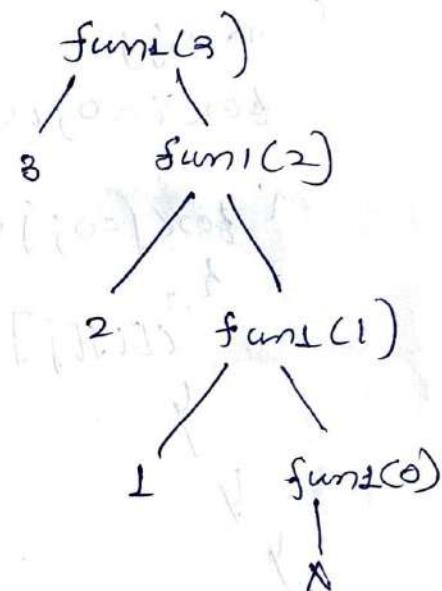
⇒ void fun2(int n)

1. if ($n > 0$)

2. fun2($n - 1$);
 printf("%d", n);

void main()

1. int x = 3;
 sum2(x);



Generalizing Recursion

`void fun(int n)`

{
 if ($n > 0$)

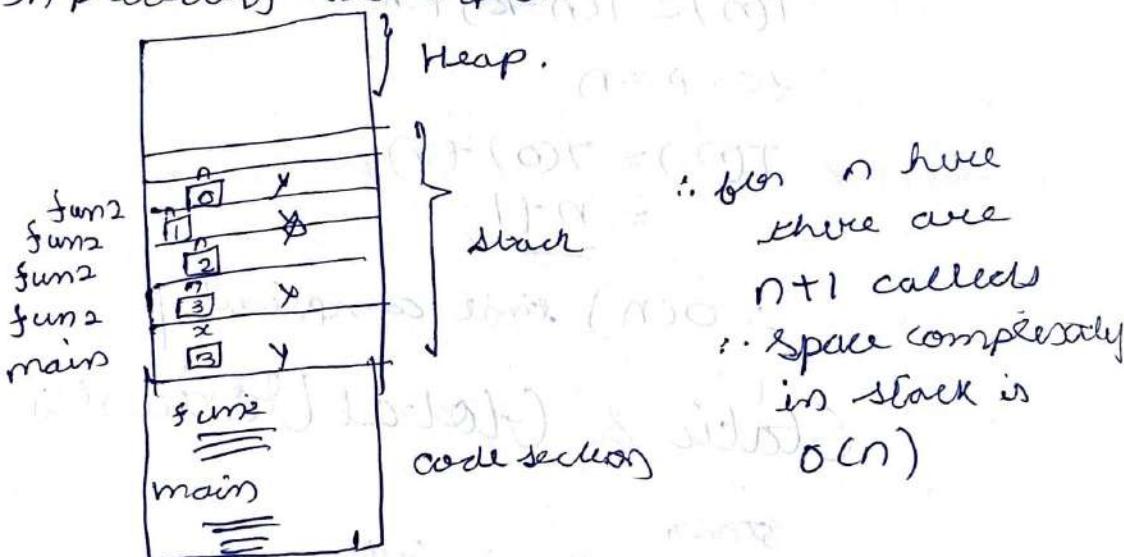
{

- phases of recursions.
- Ascending phase →
 1. Calling Time
 2. $\text{fun}(n-1) * 2$
 - Descending Phase →
 3. Returning time

diff b/w loop recursion is loop has only Ascending Phase.

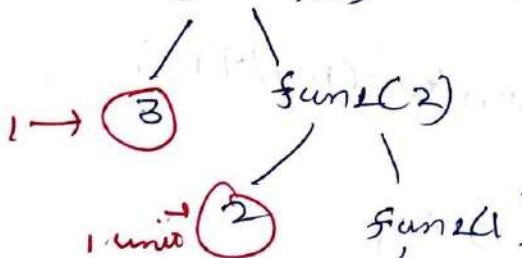
How Recursion uses Stack?

In previous example,



Time Complexity of Recursions

`fun(3)`



a. 3 units line

$\therefore \text{for } n \text{ will } 1 \rightarrow \textcircled{1} \text{ fun}(0)$

There will be n time complexity $\textcircled{1}$
 $\therefore O(n)$

`void fun(int n)`

1. $\{ \text{if } (n > 0)$
 $\{ \text{print}(n); \text{fun}(n-1) \}$

}

By Recurrence Relation

$$T(n) = \begin{cases} 1 & n=0 \\ T(n-1) + 1 & n > 0 \end{cases}$$

void fun1(int n) {
 if (n > 0) {
 cout << "calling fun1(" << n << ",);
 T(n-1);
 }
}

$$T(0) = T(0-1) + 2 \approx T(0-1) + 1$$

$$\begin{aligned} T(n) &= T(n-2) + 1 + 1 \\ &\geq T(n-2) + 2 \end{aligned}$$

$$\therefore T(n) = T(n-3) + 3$$

$$\therefore T(n) \leq T(n-k) + k$$

for $k=n$

$$\begin{aligned} T(n) &= T(0) + n \\ &= \underline{\underline{n+1}} \end{aligned}$$

$\therefore O(n)$ time complexity

Static & Global Variables in Recursion

Static

without static variable

int fun1(int n)

{ if (n > 0)

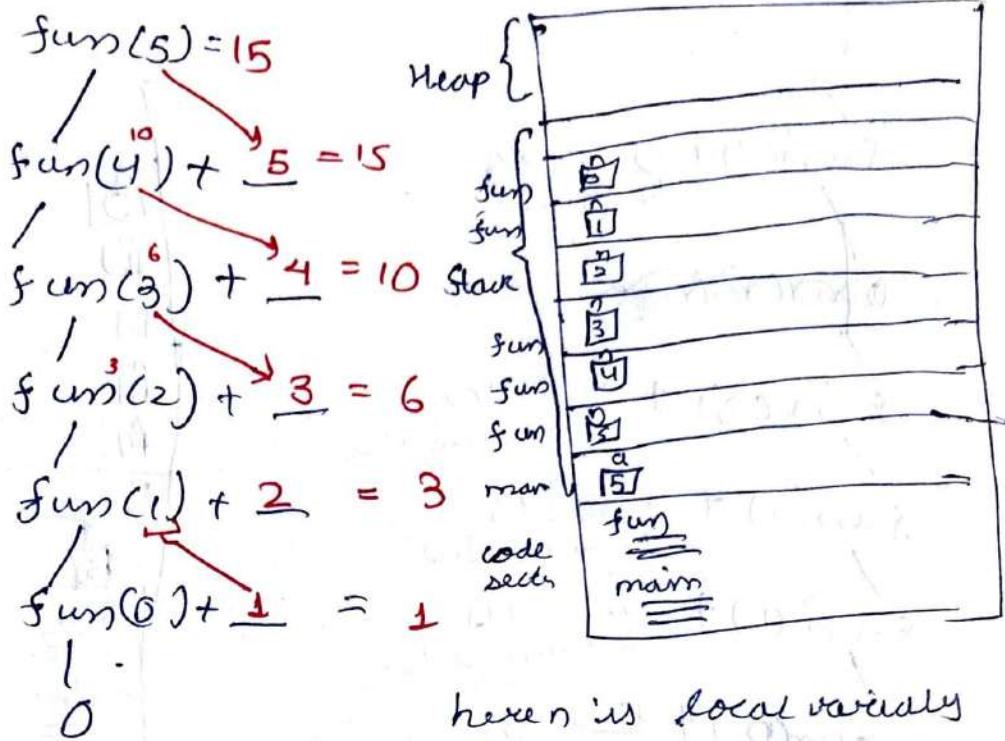
{ cout << fun1(n-1) + n;

}

return 0;

int main()

{ int a = 5;
 cout << fun1(a);
 return 0; }



Now we are creating ~~local~~ static variable

```

int
int fun(int n)
{
    static int x=0
    if (n>0)
        x++;
    returns fun(n-1)+x;
}
return 0;

```

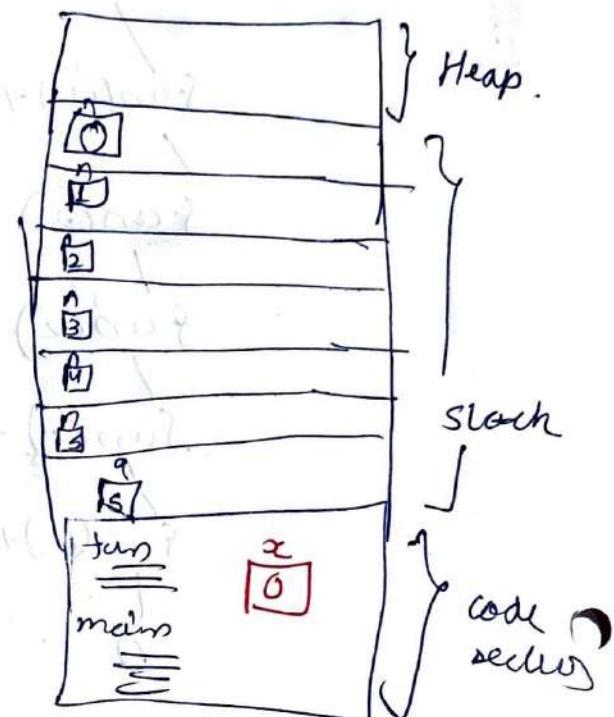
// same as above except

- ⇒ State variable are created in Code sections in sections in code sections for global and static variables
- ⇒ State variable are not created every time the function is called it is created only at one time in ~~work~~ when loading of program.

80

~~XX845~~ Don't show static variable in tracing tree

$$\begin{aligned}
 &\text{fun}(0) = \\
 &\quad \text{fun}(4) + \frac{5}{4} = 25 \quad \text{value of } x \\
 &\quad \downarrow \\
 &\quad \text{fun}(3) + \frac{5}{4} = 20 \quad \text{value of } x \\
 &\quad \downarrow \\
 &\quad \text{fun}(2) + \frac{5}{4} = 15 \quad \text{value of } x \\
 &\quad \downarrow \\
 &\quad \text{fun}(1) + \frac{5}{4} = 10 \quad \text{value of } x \\
 &\quad \downarrow \\
 &\quad \text{fun}(0) + \frac{5}{4} = 5 \quad \text{value of } x
 \end{aligned}$$



Types of Recursion:

- 1) Tail Recursion.
- 2) Head Recursion.
- 3) Leaf Recursion.
- 4) Indirect Recursion.
- 5) Nested Recursion.

→ Tail Recursion

If a recursive fun is calling itself and the recursive call is the last statement in a function.

e.g.:

```
void fun(int n)
```

```

    if (n > 0)
    {
        cout << "Id" << n;
        fun(n - 1); // Last statement
    }

```

Y

every thing is performed on calling time
no task done at returning call.

Tail Recursion vs Loop

Tail recursion can be easily converted into loop

`void fun(int n)`

{ if ($n > 0$)

{ printf("%d", n);
fun(n - 1); } }

conversion

↓
—————

$\underline{\text{fun(3) Time } O(n)}$

`void fun(int n)`

{ while ($n > 0$)
{ printf("%d", n);
 $n--$; } }

↓
—————

$\underline{\text{fun(3) Time : } O(n)}$

Ques: Now which one is Efficient?

Time complexity: $O(n)$
Space complexity

$O(n)$ " n activation records created
in stack

Time complexity: $O(n)$
Space complexity

$O(1)$ ✓ only one activation record
Better 😊

CONCLUSION: ∴ It is better to convert Tail recursion into loop

Head Recursion

- ⇒ Function has to do task on returning no task is done on calling time: T
- ⇒ First statement of function is recursive call.

`void fun(int n)`

{ if ($n > 0$)

{ fun(n - 1); }

=====

↓

`void fun(int n)`

{ int i = 1;

while ($i \leq n$)

{ printf("%d", i); }

↓

They cannot be easily converted into loop as it is converted into loop.

Linear
Recursions

$\text{fun}(n)$

{
if($n > 0$)

{

\equiv

$\rightarrow \text{fun}(n-1);$

\equiv

}
y

Tree Recursions.

$\text{fun}(n)$

{
 $y(n=0)$

{

$\rightarrow \underline{\text{fun}(n-1)};$

$\rightarrow \underline{\text{fun}(n-1)}$

\equiv

y
y

i.e. function
call's itself
more than one
times

Tree Recursions

void fun(n)

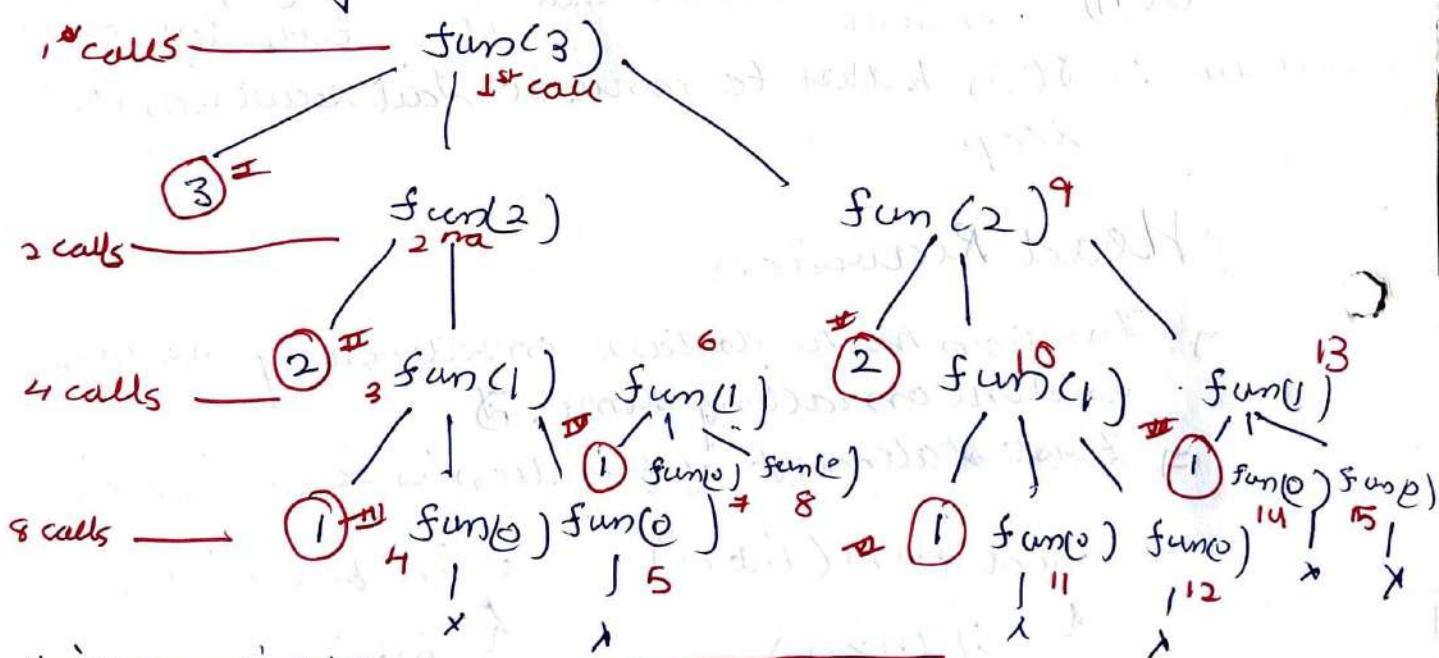
{
 $y(n=0)$

{
 $\text{print}("d", n)$
 $\text{fun}(n-1);$
 $\text{fun}(n-1);$

y

y

Tracing:



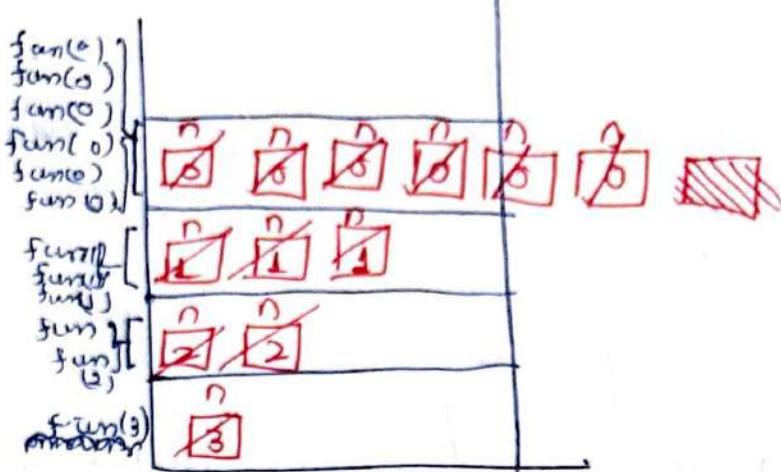
$$2^0 + 2^1 + 2^2 + 2^3 = \text{sum of GP} = 2^{n+1} - 1$$

∴ Generally, $2^{n+1} - 1$

∴ Time Complexity : $\underline{\underline{O(2^n)}}$

Space Complexity: $\frac{4}{3}^{\text{height}} = n + 1 = \underline{\underline{O(n)}}$

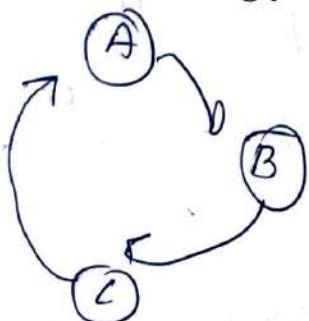
maximum call
in stack



Stack

Maximum 4 calls are exec at a time in stack & Total calls are 15

Indirect Recursion



call B & B call C & C call A again
 \therefore Circular Pattern

```

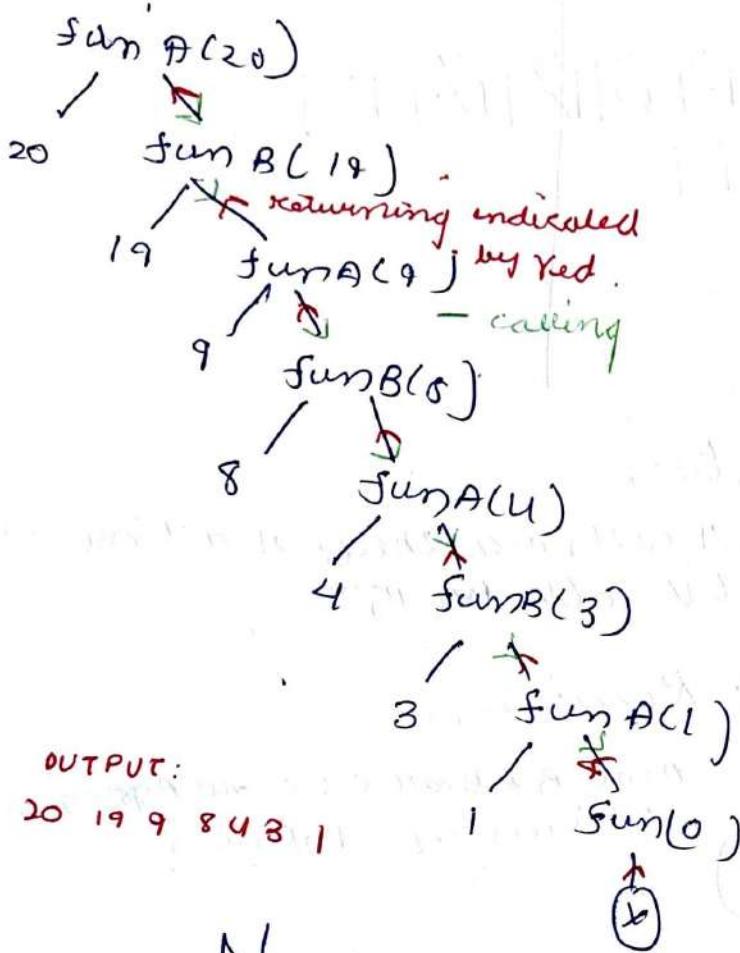
void A(int n)
{
    if (n == 0)
        return;
    else {
        cout << "A(" << n << ")";
        B(n - 1);
    }
}

void B(int n)
{
    if (n == 0)
        return;
    else {
        cout << "B(" << n << ")";
        A(n - 1);
    }
}

```

34

Indirect Recursion



```

void funA(int n)
{
    if (n > 0)
        cout << "d", n,
        funB(n-1);
    else
        void funB(int n)
{
    if (n > 1)
        cout << "d", n,
        funA(n/2);
    else
        funA(20);
}
  
```

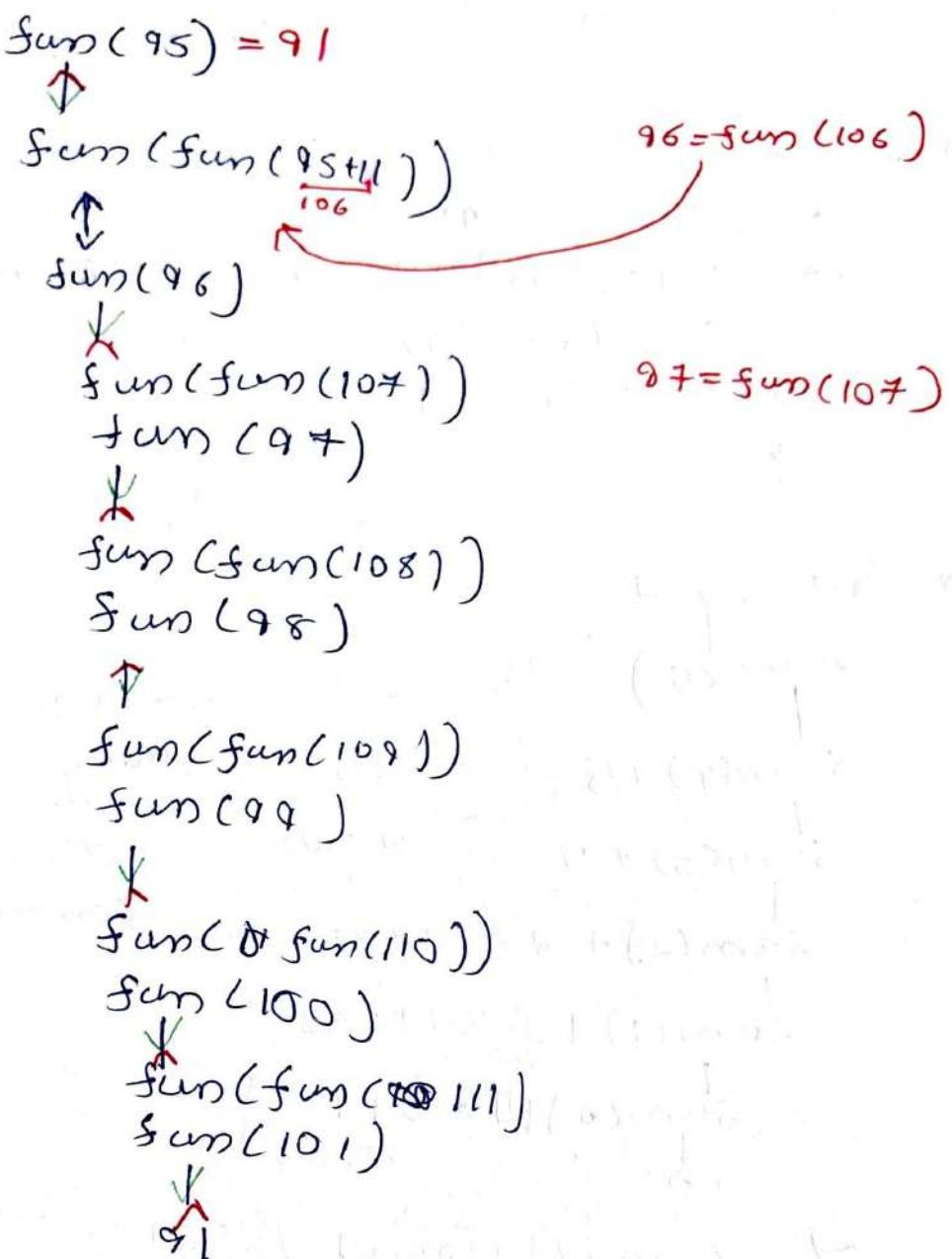
Nested Recursion

Recursion inside recursion

```

int fun(int n)
{
    if (n > 100)
        return n - 10;
    else
        return fun(fun(n + 1));
}
  
```

Tracing:
`fun(5)`:



⇒ Sum of Natural Number's Using Recursion.

$$1+2+3+\dots+n$$

$$\text{sum}(n) = (1+2+3+4+\dots+(n-1))+n$$

$$\text{sum}(n) = \text{sum}(n-1) + n$$

$$\text{sum}(n) = \begin{cases} 0 & n=0 \\ \text{sum}(n-1) + n & n>0 \end{cases}$$

⇒ int. sum(int n)

if ($n == 0$)
return 0;

else
return sum(n-1) + n;

But recursion is costlier.
since it uses more time.

Space complexity: $O(1)$

Time complexity: $O(n)$

Q6

function II

```

int sum (int n)
{
    returns n * (n+1)/2;
}

```

Time complexity
 $O(1)$
Best Method

III + int sum (int n)

loop

```

int i, s=0; — ①
for (i=1; i<=n; i++) — n+1
    s=s+i; — n
return s; — 1
}

```

Time complexity
 $O(n)$

Tracing I

$$\text{sum}(5) = 15$$

$$\text{sum}(4)+5 = 10+5 = 15$$

$$\text{sum}(3)+4 = 6+4 = 10$$

$$\text{sum}(2)+3 = 3+3 = 6$$

$$\text{sum}(1)+2 = 1+2 = 3$$

$$\text{sum}(0)+1 = 0+1$$

Total value of 5

there are $n+1$

calls

\therefore Time complexity $= O(n)$

Factorial using Recursion:

$$n! = 1 \times 2 \times 3 \times \dots \times n$$

$$5! = 1 \times 2 \times 3 \times 4 \times 5 = 120$$

$$\text{fact}(n) = 1 \times 2 \times 3 \times \dots \times (n-1) \times n$$

$$\text{fact}(n) = \text{fact}(n-1) \times n$$

$$\text{fact}(n) = \begin{cases} 1 & n=0 \\ \text{return fact}(n-1) \times n & n>0 \end{cases}$$

int fact (int n)

if ($n==0$)

return 1;

else return fact(n-1) * n;

}

37

If we take $n > n$, then the function will get into recursive conditions and call the recursion in infinite.

But it will terminate at some point due to stack overflow.

Power using Recursion

$$2^5 = \underbrace{2 * 2 * 2}_{4 \text{ times}} * 2$$

$$m^n = m * m * m * \dots \text{ for } n \text{ terms}$$

$$\text{pow}(m, n) = \underbrace{(m * m * m * \dots * m)}_{\text{Repeating } (n-1) \text{ times}} * m$$

$$\text{pow}(m, n) = \text{pow}(m, n-1) * m$$

$$\text{pow}(m, n) = \begin{cases} 1 & n=0 \\ \text{pow}(m, n-1) * m & n>0 \end{cases}$$

```
int pow(int m, int n)
{ if (n == 0)
```

executes 1

returns $\text{pow}(m, n-1) * m$

}

$\text{pow}(2, 8)$

$$\text{pow}(2, 8) * 2 = 2^9$$

$$\text{pow}(2, 7) * 2 = 2^8$$

$$\text{pow}(2, 6) * 2 = 2^7$$

$$\text{pow}(2, 5) * 2 = 2^6$$

$$\text{pow}(2, 4) * 2 = 2^5$$

$$\text{pow}(2, 3) * 2 = 2^4$$

$$\text{pow}(2, 2) * 2 = 2^3$$

$$\text{pow}(2, 1) * 2 = 2^2$$

$$\text{pow}(2, 0) * 2 = 2^1$$

* multiple
calls
are made.

Space complexity $O(n)$
 $\because n$ activations, recursion

Time complexity $O(n)$
 $\because n$ calls are made.

③ More efficient way to write Poem fun
(Reduce)

int paul(int m, int n)

$\{i_4(n=0)\}$

* returns i

if ($n \neq 0$)

returns $\text{pow}(m*m, n/2)$

else

scissors myron (m+n, 6-11/2)

$$\text{row}(2, 4) = 2^4$$

$$2 \nmid \text{row}(2^2, 4) = 2 \nmid 2^8 = 2^9$$

6 multiplications
is taken

faster fun

$$\text{pw}(2^4, 2) = 28$$

$$\text{row } \underbrace{(2^8, 1)}_{1} = 2^8$$

$$2^8 * \text{paro}(2^{16}, 0) = 2^8 * 1 = 28$$

Taylor series using revisions.

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots + n \text{ terms.}$$

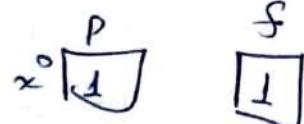
$$\text{sum}(n-1) + n \quad \text{sum}(n) = 1 + 2 + 3 + \dots + n \Rightarrow \text{adding done at}$$

$$\text{faktor(n) \<} \text{faktoren} = 1 \times 2 \times 3 \times \dots \times n$$

rechnung
some

$$\text{row}(x, n) \rightarrow \text{row}(x, n) = x * x * \dots * x \text{ (n times)}$$

$$e(x, n) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!}$$



$$e(x, 3) = \frac{P}{f} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!}$$

$$P = P * x$$

$$f = f * n$$

$$e(x, 2) = \frac{P}{f} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!}$$

$$e(x, 2) = \frac{P}{f} = 1 + x + \frac{x^2}{2}$$

$$e(x, 0) = \frac{P}{f} = 1 + x$$

~~Temporary~~

```

1 int e (int x, int n)
{
    static int P = 1, f = 1;
    int r;
    if (n == 0)
        return 1;
    else
    {
        e = e(x, n - 1);
        P = P * x;
        f = f * n;
        return e * r + P / f;
    }
}

```

Taylor Series Using Horner's Rule.

$$e^x = 1 + \frac{x}{1} + \frac{x^2}{2!} + \frac{x^3}{3!}$$

$$\frac{\partial x}{1 \times 2} \quad \frac{\partial x \cdot x}{1 \times 2 \times 3} \quad \frac{3 \text{ multi}}{3 \text{ multi}}$$

$$2 + 4$$

$$6 + 8$$

$$2[1 + 2 + 3 + 4 \dots]$$

$$2 \frac{(n(n+1))}{2}$$

$$= n(n+1)$$

Time complexity: $O(n^2)$

multiplications are required

40

$$1 + \frac{x}{1} + \frac{x^2}{1+2} + \frac{x^3}{1+2+3} + \frac{x^4}{1+2+3+4}$$

$$1 + \frac{x}{1} \left[1 + \frac{x}{2} + \frac{x^2}{2+3} + \frac{x^3}{2+3+4} \right]$$

$$1 + \frac{x}{1} \left[1 + \frac{x}{2} \left[1 + \frac{x}{3} + \frac{x^2}{3+4} \right] \right]$$

$$1 + \frac{x}{1} \left[1 + \frac{x}{2} \left[1 + \frac{x}{3} \left[1 + \frac{x}{4} \right] \right] \right]$$

\therefore 4 multiplications,
 $\therefore O(n)$
 Time complexity

```
int f(int x, int n)
{
    int s = 1;
    for ( ; n > 0; n--)
    {
        s = 1 + x/n * s;
    }
    return s;
}
```

```
int f(int x, int n)
{
    static int s = 1;
    if (n == 0)
        return s;
    s = 1 + x/n * s;
    return f(x, n - 1);
}
Time complexity
= O(n)
```

MORAL: By taking common we can reduce no of multiplication from quadratic to linear

Fibonacci Series Using Recursion

$f(n)$	0	1	1	2	3	5	8	13
n	0	1	2	3	4	5	6	7

$$f(n) = \begin{cases} 0 & n=0 \\ 1 & n=1 \\ f(n-2) + f(n-1) & n>1 \end{cases}$$

```
int fib(int n)
{
    if (n <= 1)
        return n;
    return fib(n-2) + fib(n-1);
}
```

Fibonacci seising loop

41

`int sum(int n)`

int $t_0 = 0, t_1 = 1, s, i; -1$

if ($n \leq 2$) return $n - 1$

for ($i = 2$; $i \leq n$; $i++$) — n

Time complexity
 $O(n)$

$$s = t_0 + t_1 \quad - n-1$$

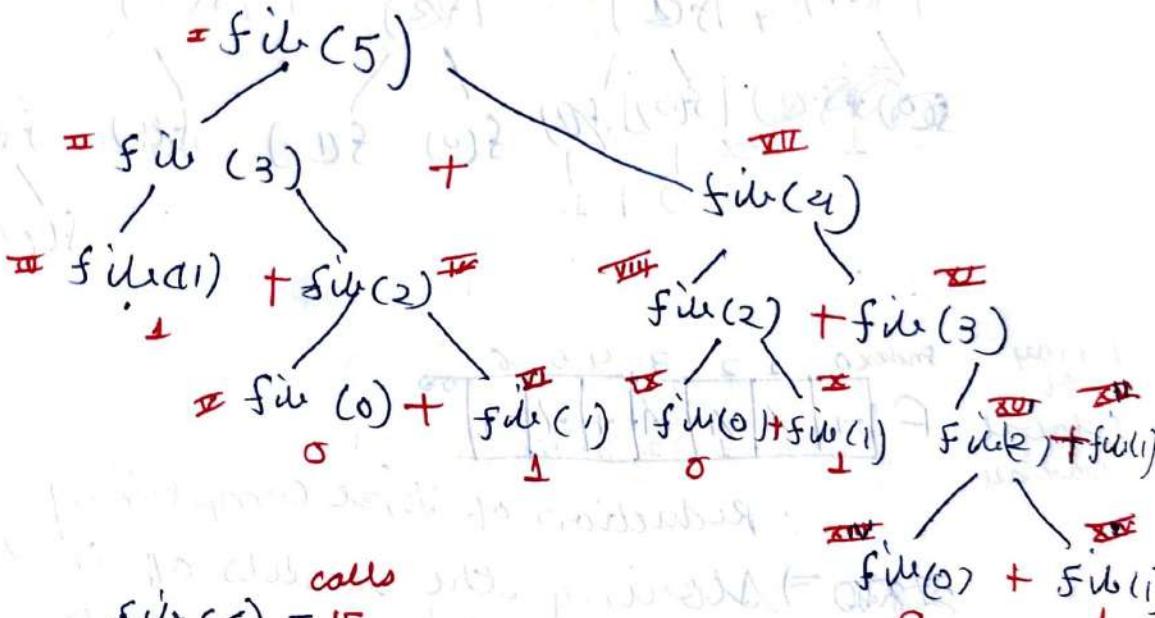
$$t_0 \in t_{1,j} - n-1$$

$$t_1 = s_1 - n-1$$

return 5;

Training Revision of Falconari

order of calling



~~sub(5)~~ - 15

$\sin(u) = ?$

$$\sin(3) - 5$$

Now we cannot predict time complexity
some will assume

even, ~~if~~ $\text{for } n \geq 2 \text{ find } n-1$

∴ for each $\text{fib}(n)$ there will be
2 calls from $\text{fib}(n-1)$ to $\text{fib}(n)$

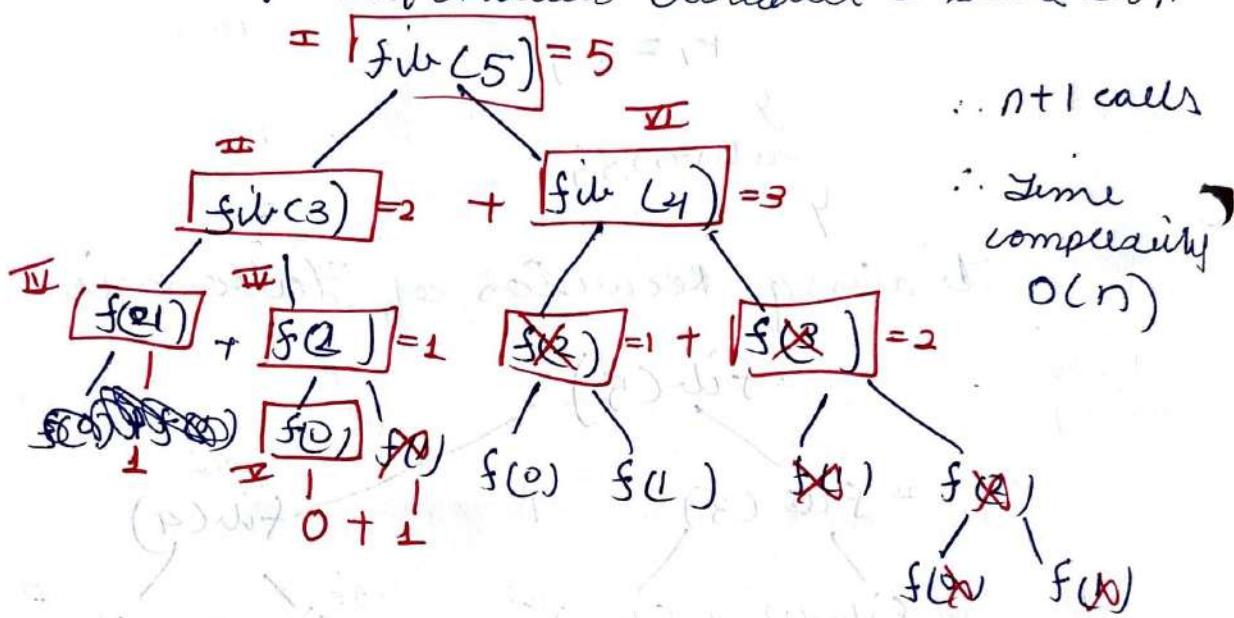
There will be ~~ways~~ $2 \times 2 \times 2 \dots n$ ways
 $= 2^n$ calls.

Now to make it faster we analyse
that $\text{fib}(3)$, $\text{fib}(2)$, $\text{fib}(1)$, $\text{fib}(0)$ are used
multiple times increasing.
So a recursive function calling itself multiple
times for some values are called Excessive Recursion.

So this func for Fibonacci is ex. of "Excessive Fx"

Method to Prevent :

To retain value of repeated element we store
in global or static variable & store them



Array of global variable	Index 0	1	2	3	4	5	6
F	0	1	1	1	2	3	5

∴ Reduction of Time complexity

~~Storing~~ → Storing the results of the functions
call so that they can be utilized again
when we need a same call or avoiding
excessive call. This approach is called
as a Memorization

\therefore Memoization Function

int fib(int n)

{ if ($n \leq 1$)

$f[n] = n;$

 return n;

 else

 { if ($f[n-2] == -1$)

$f[n-2] = fib(n-2);$

 if ($f[n-1] == -1$)

$f[n-1] = fib(n-1);$

 return $f[n-2] + f[n-1];$

 }

}

$\cap C_r$ using Recursion:

$$C_r = \frac{n!}{r!(n-r)!}$$

int C (int n, int r)

{ int t₁, t₂, t₃;

 t₁ = fact(n);

 t₂ = fact(r);

 t₃ = fact(n-r);

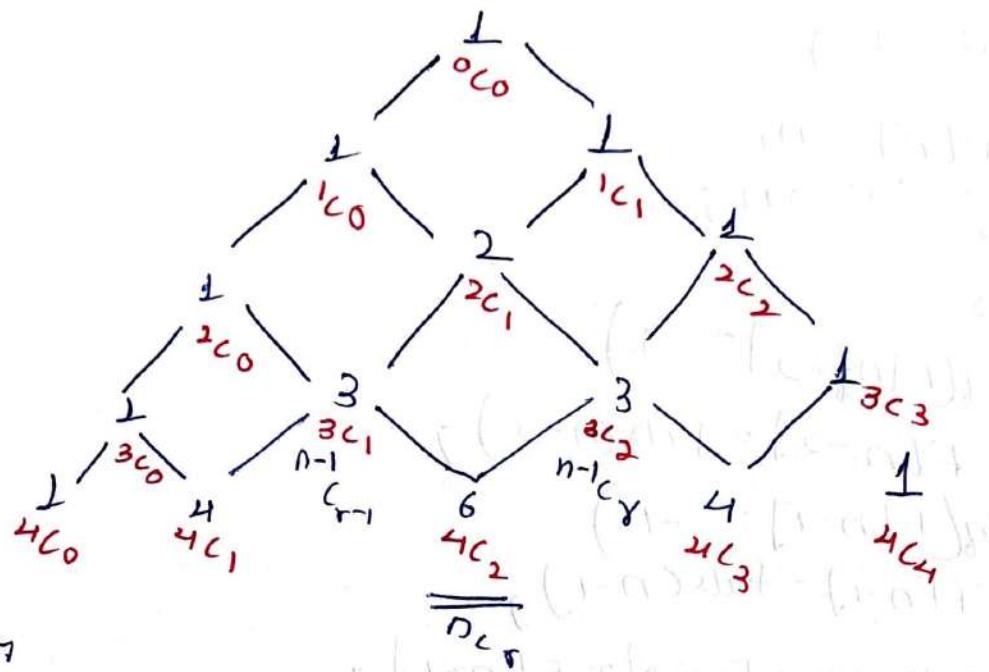
 ① — returns $t_1 / t_2 * t_3$

$\frac{1}{3n+1} \therefore O(n)$

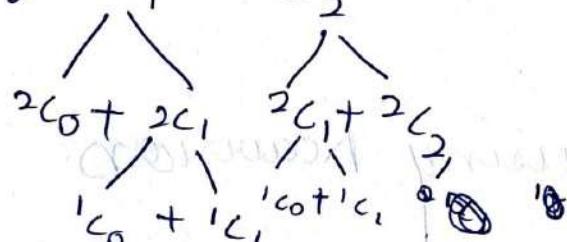
Now to calculate ~~C_r~~ C_r using recursive function

44

Pascal's Triangle.



To calculate ${}^4C_2 = {}^3C_1 + {}^3C_2$



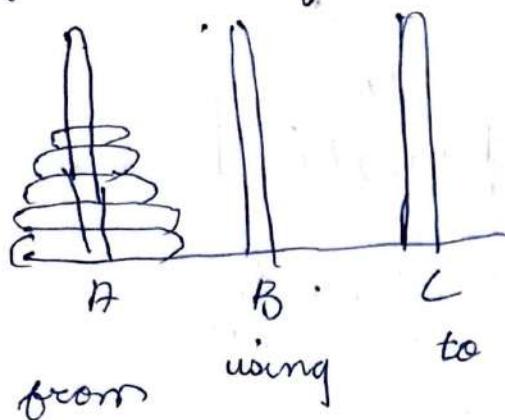
Recursive Functions:

```

int c(int n, int r)
{
    if (r == 0 || n == r)
        return 1;
    else
        return c(n-1, r-1) + c(n-1, r);
}

```

Tower of Hanoi Problem



- $\text{TOH}(1, A, B, C)$

→ Move Disk from A to C using
B



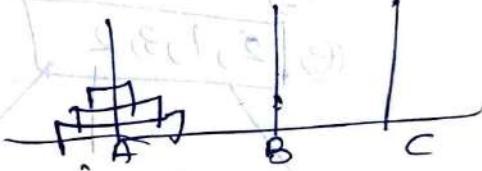
- $\text{TOH}(2, A, B, C)$

1. $\text{TOH}(1, A, C, B)$

2. Move Disk from A to C using B

3. $\text{TOH}(1, B, A, C)$

L_A is used as intermediate although same order.



- $\text{TOH}(n, A, B, C)$

1. $\text{TOH}(n-1, A, C, B)$

2) Move Disk from A to C using B

- 3) $\text{TOH}(1, B, A, C)$

n-1

void $\text{TOH}(\text{int } n, \text{int } A, \text{int } B, \text{int } C)$

from using to

if ($n > 0$)

{ $\text{TOH}(n-1, A, C, B)$; }

print ("Move " + n + " from " + A + " to " + C);

$\text{TOH}(n-1, B, A, C);$

}

$\text{TOH}(3, 1, 2, 3)$

n A B C

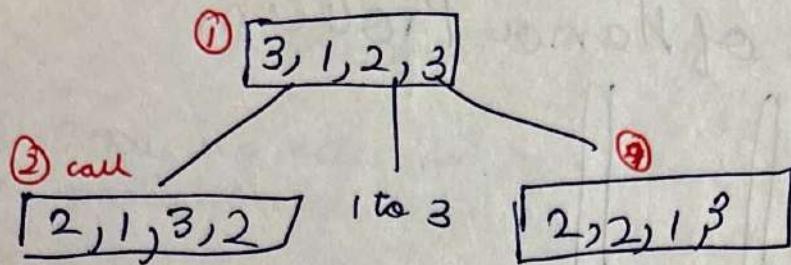


abb a bba bbb cc a
 1st part IV 2nd part

a
b
b

c
c
a

$$c[0] = 1$$

$$c[1] = 2$$

$$c[0] = 1 - 1 = 0$$

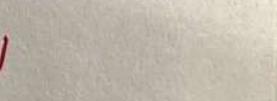
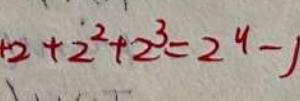
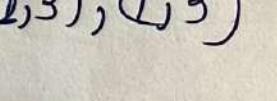
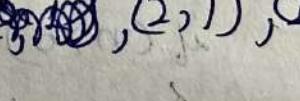
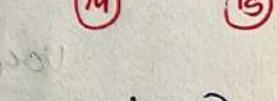
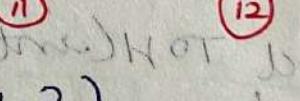
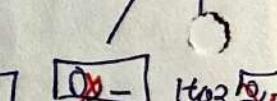
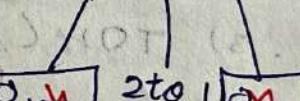
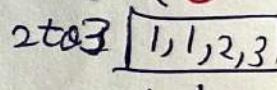
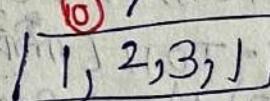
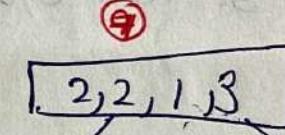
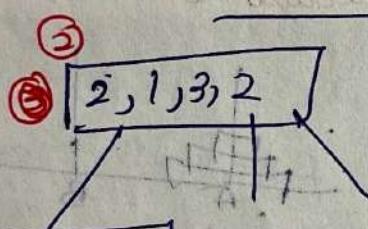
$$c[2] = -2$$

Now for k = 0 to 25

$$\text{if } c[k] > 0$$

$$\text{count} = \text{count} + c[k]$$

$$\text{count} = 0 + 2$$



output : (1, 3), (1, 2), (3, 2), (1, 1), (2, 1), (2, 3), (4, 3)

Time complexity: ~~O(n!)~~ ~~O(2^n)~~

$$n = 3 \Rightarrow 15 \quad 1 + 2 + 2^2 + 2^3 = 2^4 - 1$$

$$n = 2 \Rightarrow 4 \quad 1 + 2 + 2^2 = 2^3 - 1$$

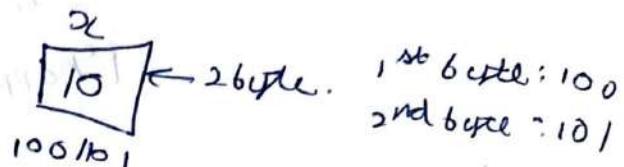
$$1 + 2 + 2^2 + \dots + 2^n = 2^{n+1} - 1$$

∴ Time complexity : $O(2^n)$

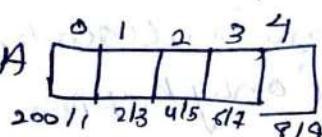
ARRAYS:

Array is a collection of similar data elements grouped under one name.

Scalar variable \rightarrow `int x=10;`



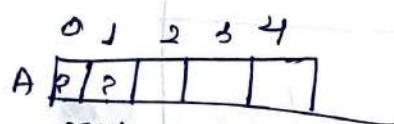
vector \rightarrow `int A[5];`



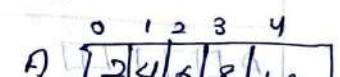
for 5 ints 10 blocks of memory is allocated.
All five ints have same name 'A'. we can differentiate 5 ints with their indices.

Declarations

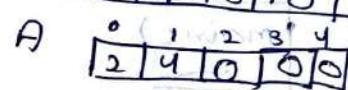
① `int A[5];`



② `int A[5]={2,4,6,8,10};`

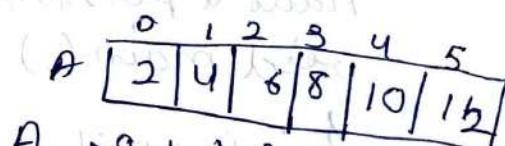


③ `int A[5]={2,4};`

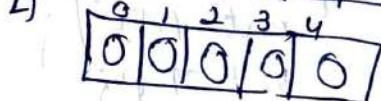


one initialization is done
other remaining elements
should be zero.

④ `int A[]={2,4,6,8,10,12};`



⑤ `int A[5]={0};`



Printing

① `for(i=0; i<4; i++)`
 { printing ("%.d", A[i]) }
 ↴

② `printf("%d", A[2]);`

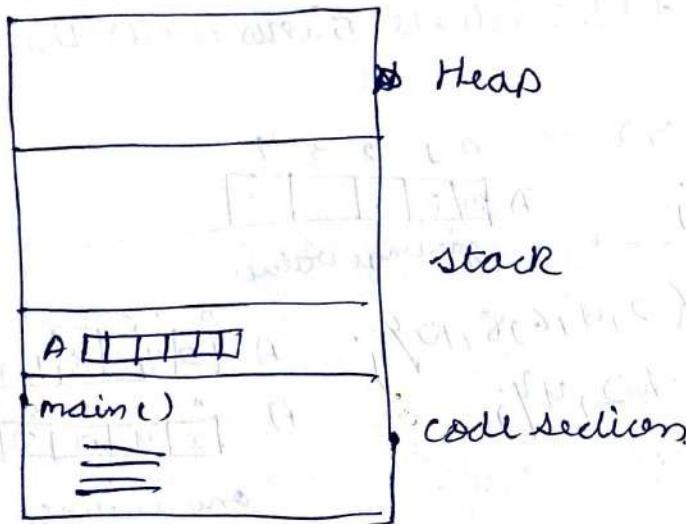
⑥ By pointers : `printf("%d", *A+2);`

Static & Dynamic Array

`void main()`

{ `int A[5];` → size and memory is allocated at compilation time only
[happens in C & C++]

`int n;` → size is allocated at runtime
`cin >> n;` [only happens in C++]
`int B[n];` array of `n` size will be created
in the stack.



for accessing anything from heap we must have a pointer

`void main()`

{
`int A[5];`
`int *P;`
`P = new int[5];`

`new` is an operator in C++ for

`P = (int*) malloc(5 * sizeof(int));`

If the unused memory is not released then it will cause memory leakage. (The memory that is not required & will be in program),

C++ : `delete [] p;` } → deallocation
 C : `free(p)`

→ we cannot change the size of array once allocated.

In heap array, it can be resized using one technique but stack array cannot be resized at all.

Method to indirectly increase size of Array.
`main()`

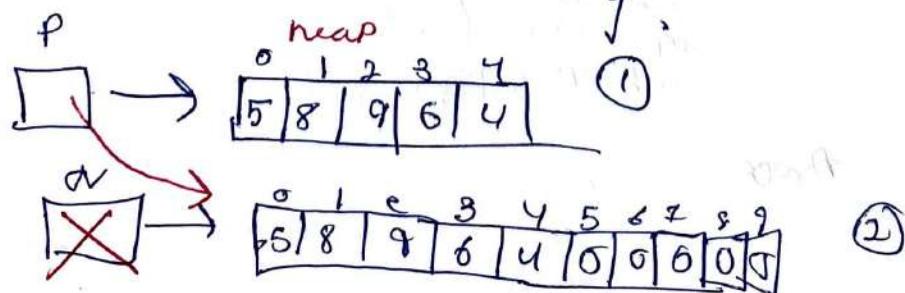
```
{ int *p = new int[5]
  int *q = new int[10];
  for(i=0; i<5; i++)
    q[i] = p[i]; }
```

`delete [] p;` → array ① is deleted.

`p = q` → p is also now point to array ②.

`q = NULL` → Now q is not pointing anything.

only p is pointing to a new sized array.



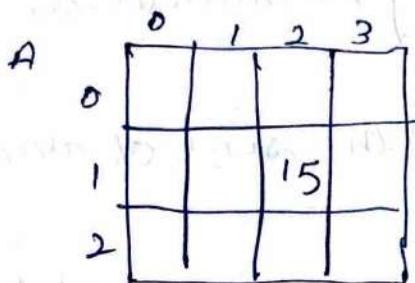
50

2D Array

3 row 4 columns

- $$\textcircled{1} \quad \text{int } A[3][4] = \{ \{1, 3, 4, 2\}, \{2, 4, 6, 8\}, \{5, 3, 7, 4\} \};$$

→ it is all created in stack because no new operators is used.



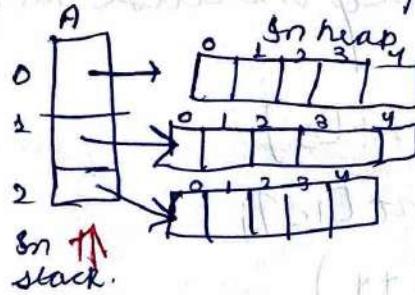
$$A \cap B = 15;$$

Compiler treats it like single dimensional array, arranges as A [] is compiler
But compiles let us use this 1D array as 2D arr

- ② ~~Compiler-located~~
int *A[3];

Partially in heap
partially in stack

it is ~~an~~ away of int sources

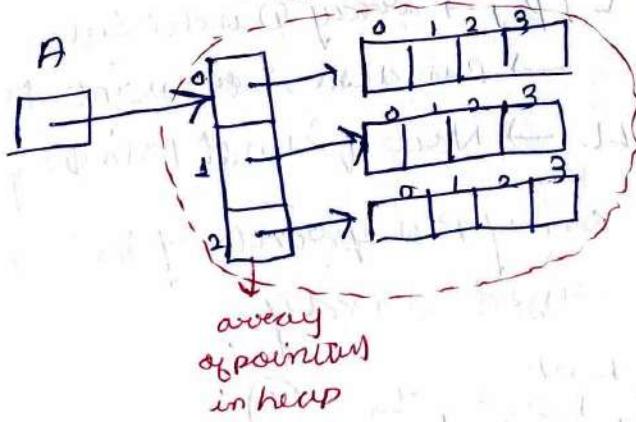


```
for(i=0;i<3;i++)
    A[i] = new int[15];
```

- ③ int * A
it is double pointer

Array full in
heap

on head



$A = \text{new int}^*[3]$
 $\& A[i] = \text{new int}[4];$

51

Array Representation by Compiler

`int A[5] = {3, 5, 8, 4, 2}`

A	0	1	2	3	4
	3	5	8	4	2
L ₀	20011	20213	20415	20617	20819

$$A[3] = 10$$

$$\text{Addr}(A[3]) = 200 + 3 \times 2 = 206$$

$$\text{Addr}(A[3]) = L_0 + 3 \times 2$$

$$\boxed{\text{Addr}(A[i]) = L_0 + i \times w} \rightarrow \text{size of data type}$$

relative formula

since it
is selected

to base address of array

base Address index

Q In any language index starts from 1 this
and language

`int A[1..5]`

A	1	2	3	4	5
	3	5	8	4	2
L ₀	20011	21313	41615	61717	819

$$A[3] = 10$$

$$\text{Addr}(A[3]) = 200 + (3-1) \times 2 = 204$$

$$\boxed{\text{Addr}(A[i]) = L_0 + (i-1) \times w}$$

In C++, it is strictly starting from 0
since it starts with other indices like

if there are three operations +, -, & *
for storing variable in that array

we have to compute address of every index
n times... n extra operations of '-' has to
done... time consuming

(52)

Row Major Formula for 2D Array

Representations of 2D into 1D mapping occupy
single domain
There are two types of mapping for it.

- (i) Row-major mapping
- (ii) Column-major mapping

Row-major mapping

	0	1	2	3	4	5	6	7	8	9	10	11
A												
200	11	213	4	6	8	10	12	14	16	18	20	22

	0	1	2	3	
A	0	a_{00}	a_{01}	a_{02}	a_{03}
	1	a_{10}	a_{11}	a_{12}	a_{13}
	2	a_{20}	a_{21}	a_{22}	a_{23}

	0	1	2	3	4	5	6	7	8	9	10	11	
A													
200	11	a_{00}	a_{01}	a_{02}	a_{03}	a_{10}	a_{11}	a_{12}	a_{13}	a_{20}	a_{21}	a_{22}	a_{23}

row 0 row 1

$$\text{Add}[A[1][2]] = 200 + [1*4 + 2]*2 \\ = 200 + 6*2 = 212$$

$$\text{Add}[A[2][3]] = 200 + [2*4 + 3]*2 = 222$$

$$\text{Add}[A[i][j]] = L_0 + [i*n + j]*w$$

C/C++ follow Row Major Mapping formula (53)
if index are starting from 1

$$\text{Ans } A[1..3][1..4]$$

$$\text{Addr}[A[i][j]] = L_0 + [(i-1)*n + (j-1)] * w$$

Column Major Representation

A	1	2	3	4	5	6	7	8	9	10	11
	a_{00}	a_{10}	a_{20}	a_{01}	a_{11}	a_{21}	a_{02}	a_{12}	a_{22}	a_{03}	a_{13}
	200	2	4	6	8	10	12	14	16	18	20
	col 0	col 1	col 2	col 3							

Ans $A[m][n]$

$$\text{Addr}[A[i][j]] = 200 + (2*3 + j)*2 = 214$$

$$[\text{Addr}[A[i][j]] = L_0 + (j*m + i)*w]$$

Formula for nD Arrays

Arrays in Compiler. Type: $A \in [d_1][d_2][d_3] \dots [d_n]$

Row major

$$\begin{aligned} \text{Addr}[A[i_1][i_2][i_3][i_4]] &= L_0 + [i_1 * d_1 * d_2 * d_3 * d_4 + i_2 * d_2 * d_3 * d_4 \\ &\quad + i_3 * d_3 * d_4 + i_4] * w \\ &= L_0 + \sum_{p=1}^n [i_p * \prod_{q=p+1}^n d_q] * w \end{aligned}$$

Column Major

$$\begin{aligned} \text{Addr}[A[i_1][i_2][i_3][i_4]] &= L_0 + [i_4 * d_1 * d_2 * d_3 + i_3 * d_1 * d_2 \\ &\quad + i_2 * d_1 + i_1] * w \end{aligned}$$

~~$\sum_{p=1}^n [i_p * \prod_{q=p+1}^n d_q]$~~

(54)

For you - Horner's Formula :

$$L_0 + \underbrace{i_1 * d_2 * d_3 * d_4}_{3 \text{ operations}} + \underbrace{i_2 * d_3 * d_4}_{2 \text{ operations}} + i_3 * d_4 + i_4$$

1 operation

$$4D \rightarrow 3+2+1$$

$$5D \rightarrow 4+3+2+1$$

$$nD \rightarrow (n-1) + (n-2) + \dots + 1 = \frac{n(n-1)}{2}$$

$$\therefore \underline{\mathcal{O}(n^2)}$$

TOO MUCH TIME CONSUMING

Any other way to do this ?

Yes, by Using HORNERS RULE

$$i_4 + i_3 * d_4 + i_2 * d_3 * d_4 + i_1 * d_2 * d_3 * d_4$$

$$i_4 + d_4 * [i_3 + i_2 * d_3 + i_1 * d_2]$$

$$i_4 + d_4 * [i_3 + d_3 * [i_2 + i_1 * d_2]]$$

3D + 3

4D → 4 operations

5D → 5 nD + 0

$$\therefore \underline{\mathcal{O}(n)}$$

This method of taking common is called HORN

- E.R.S Rule -

Array ADT

(5)

Data

- 1) Array size
- 2) size
- 3) Length (No. of elements)

{ almost all
computers provide
this basic array
datatype}

Operations

- 1) display ()
- 2) Add (x) / Append ()
- 3) Insert (index, x)
- 4) Delete (index, x)
- 5) search (x)
- 6) Get (index)
- 7) set (index, x)
- 8) Max () / Min ()
- 9) Reverse ()
- 10) shift () / rotate ()

Representations of Data

Two ways to declare

in compilation

① statically in stack.

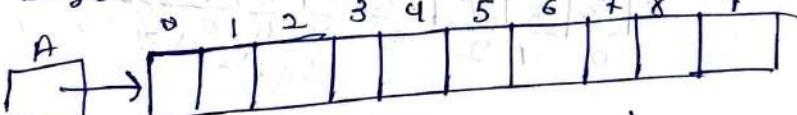
int A[10];

② dynamically in heap

int *A;
A = new int [size]

in runtime

size = 10



Length = 0 (nothing in array)

(56)

Operations in this Array:

#include <stdio.h>

struct Array

{ int A[20];

int size;

int length;

}; void display (struct Array arr)

display

```
{ int i;
  printf ("Elements are \n");
  for (i=0; i<arr.length; i++)
    printf ("%d", arr.A[i]); }
```

}

+ Add(x) / Append(x)

A[length] = x;

length + 1;

$\frac{1}{\text{length}}$

$\frac{1}{\text{length}}$

$f(n) = 2$

$O(n^0) = O(1)$

= Insert (4, 15)

size = 10

length = 7

index = 5

value = 15

length = 8

size = 11

length = 9

size = 12

length = 10

size = 13

length = 11

size = 14

length = 12

size = 15

length = 13

size = 16

length = 14

size = 17

length = 15

size = 18

length = 16

size = 19

length = 17

size = 20

length = 18

size = 21

length = 19

size = 22

length = 20

size = 23

length = 21

size = 24

length = 22

size = 25

length = 23

size = 26

length = 24

size = 27

length = 25

size = 28

length = 26

size = 29

length = 27

size = 30

length = 28

size = 31

length = 29

size = 32

length = 30

size = 33

length = 31

size = 34

length = 32

size = 35

length = 33

size = 36

length = 34

size = 37

length = 35

size = 38

length = 36

size = 39

length = 37

size = 40

length = 38

size = 41

length = 39

size = 42

length = 40

size = 43

length = 41

size = 44

length = 42

size = 45

length = 43

size = 46

length = 44

size = 47

length = 45

size = 48

length = 46

size = 49

length = 47

size = 50

length = 48

size = 51

length = 49

size = 52

length = 50

size = 53

length = 51

size = 54

length = 52

size = 55

length = 53

size = 56

length = 54

size = 57

length = 55

size = 58

length = 56

size = 59

length = 57

size = 60

length = 58

size = 61

length = 59

size = 62

length = 60

size = 63

length = 61

size = 64

length = 62

size = 65

length = 63

size = 66

length = 64

size = 67

length = 65

size = 68

length = 66

size = 69

length = 67

size = 70

length = 68

size = 71

length = 69

size = 72

length = 70

size = 73

length = 71

size = 74

length = 72

size = 75

length = 73

size = 76

length = 74

size = 77

length = 75

size = 78

length = 76

size = 79

length = 77

size = 80

length = 78

size = 81

length = 79

size = 82

length = 80

size = 83

length = 81

size = 84

length = 82

size = 85

length = 83

size = 86

length = 84

size = 87

length = 85

size = 88

length = 86

size = 89

length = 87

size = 90

length = 88

size = 91

length = 89

size = 92

length = 90

size = 93

length = 91

size = 94

length = 92

size = 95

length = 93

size = 96

length = 94

size = 97

length = 95

size = 98

length = 96

size = 99

length = 97

size = 100

length = 98

size = 101

length = 99

size = 102

length = 100

size = 103

length = 101

size = 104

length = 102

size = 105

length = 103

size = 106

length = 104

size = 107

length = 105

size = 108

length = 106

size = 109

length = 107

size = 110

length = 108

size = 111

length = 109

size = 112

length = 110

size = 113

length = 111

size = 114

length = 112

size = 115

length = 113

size = 116

length = 114

size = 117

length = 115

size = 118

length = 116

size = 119

length = 117

size = 120

length = 118

size = 121

length = 119

size = 122

length = 120

size = 123

length = 121

size = 124

length = 122

size = 125

length = 123

size = 126

length = 124

size = 127

length = 125

size = 128

length = 126

size = 129

length = 127

size = 130

length = 128

size = 131

length = 129

size = 132

length = 130

size = 133

length = 131

size = 134

length = 132

size = 135

length = 133

size = 136

length = 134

size = 137

length = 135

size = 138

length = 136

size = 139

length = 137

size = 140

length = 138

size = 141

length = 139

for ($i = \text{length} ; i \geq \text{index} ; i--$)
 {
 $A[i] = A[i-1]$ ————— when we have to add at 0 index
 }
 $A[\text{index}] = x ; i--$ to add at index $= \text{length}$
 $\text{length}++$,
 $\therefore O(0)$ to $O(n)$
 ↓
 Best Case (min)
 ↓
 Worst Case (max)

void Append (struct Array *arr, int x)

{
 if ($\text{arr} \rightarrow \text{length} < \text{arr} \rightarrow \text{size}$)
 $\text{arr} \rightarrow A[\text{arr} \rightarrow \text{length} + 1] = x;$

void Insert (struct Array *arr, int index, int x)

{
 int i;
 if ($\text{index} \geq 0 \& \text{index} \leq \text{arr} \rightarrow \text{length}$)
 {
 for ($i = \text{arr} \rightarrow \text{length} ; i \geq \text{index} ; i--$)
 $\text{arr} \rightarrow A[i] = \text{arr} \rightarrow A[i-1]$
 $\text{arr} \rightarrow A[\text{index}] = x;$
 $\text{arr} \rightarrow \text{length}++$;
 }
 }
 }

Reducing froms Array

8	3	7	13	15	6	9	10	
0	1	2	3	4	5	6	7	8

Delete (3)

$x = A[\text{index}] ; i--$

55

```

for(i=index; i<length-1; i++)
{
    A[i] = A[i+1];
}

```

$\text{length}--$; — $\frac{1}{\text{max}=2}$ $\text{max}=n+2$

\Rightarrow It should not be beyond length.

Best Case Time: $O(1)$

worst case Time: $O(n)$

int Delete (struct Array *arr, int index)

{ int x=0;

if(index >= 0 & index < arr->length)

{ x=arr->A[index];

for(i=index; i<arr->length-1; i++)
 arr->A[i]=arr->A[i+1]

arr->length(); — $\text{max}=2$

return x;

return 0;

Search Method:

1) Linear Search

2) Binary Search

Linear Search

A	8	9	4	7	6	3	10	5	14	2
0	1	2	3	4	5	6	7	8	9	

Successful Key = 5

unsuccessful key = 12

Pseudocode for Linear Search

```
for(i=0; i<length; i++)
    {
```

if (key == A[i]) = O(1)

return i; = for successful search

```
y
    return -1;      min: O(1)
                    max: O(n)
```

\Rightarrow for unsuccessful search,
always $O(n)$

$$\text{Average case: } \frac{1+2+3+\dots+n}{n} = \frac{n(n+1)}{2} = \frac{n+1}{2} = O(n)$$

\therefore Average Case: $O(n)$

worst case: $O(n)$

Best case: $O(1)$

\Rightarrow thus Avg & worst case analysis is almost same

Improving Linear Search

To solve when we search for
the same key again.

Methods:

1) Transposition

* moving ahead by 1 step if elem

pseudo code: for $i=0$; $i < \text{length}$; $i++$)

if (key == A[i])

swap (A[i], A[i-1]);

return i-1;

2) Move to front / Head

for(i=0; i < length; i++)

if (key == A[i])

swap (A[i], A[0])

return 0;

to Code

```

Code
int linearSearch(struct Array arr, int key)
{
    int i;
    for(i=0; i<arr.length; i++)
    {
        if(key == arr.A[i])
            return i;
    }
    return -1;
}

```

```

Improving Lines search
void swap (int *x, int *y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}

```

✓
int LinearSearch (const Array arr, int key)
{
 int i;
 for (i = 0; i < arr.length; i++)
 {
 if (key == arr[i])
 return i;
 }
 return -1;
}

swap (&arr + A[i], &arr + A[i-1]);
 Transposition returns i-1
 Move to Head:
 swap (&arr + A[i], &arr + A[0])

Q

Binary Search

for sorted arr.

Size = 15
Length = 15

0	1	2	3	4	5	6	7	8	9	10	"
48	10	15	18	21	24	27	29	33	34	37	
l										h	
12	13	14									
38	41	43									

Key = 18

$$l \quad h \quad \text{mid} = \frac{l+h}{2}$$

index → 0 14 7
 0 6 3
 4 6 5
 4 4 4 found

when key is not present
 then in that case $l > h$
 \therefore we will stop.

```

Algorithm BinSearch (l, h, key)
{
    while (l ≤ h)
        mid = (l + h)/2 ;
        if (key == A[mid])
            return mid ;
        else if (key < A[mid])
            h = mid - 1 ;
        else
            l = mid + 1 ;
}
    
```

62 Recursive Algorithms for Binary Search:

A algorithm RBisSearch(l, h, key)

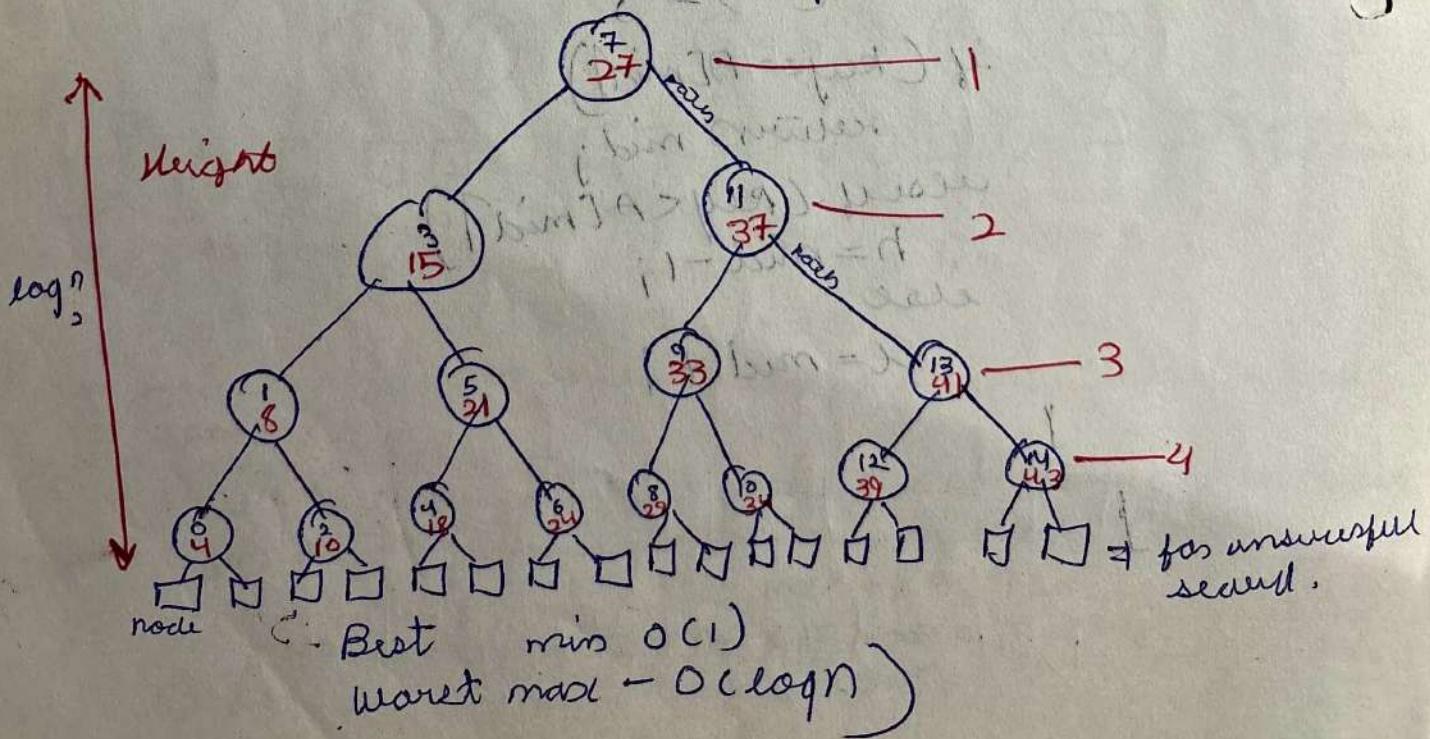
```

{
    if (l == h)
        mid = (l+h)/2;
    if (key == A[mid])
        returns mid;
    else if (key < A[mid])
        returns RBisSearch(l, mid-1, key);
    else
        returns RBisSearch(mid+1, h, key);
}
returns -1;

```

This is 'Tail Recursion'; it does not perform any task while returning call.

Analysis of Binary Search



64

Average Case

$$\sum_{i=1}^{\log n} i \times 2^i + \sum_{i=2}^{\log n} 2 \times 2^i + \sum_{i=3}^{\log n} 3 \times 2^i + \dots + \sum_{i=\log n}^{\log n} n \times 2^i$$

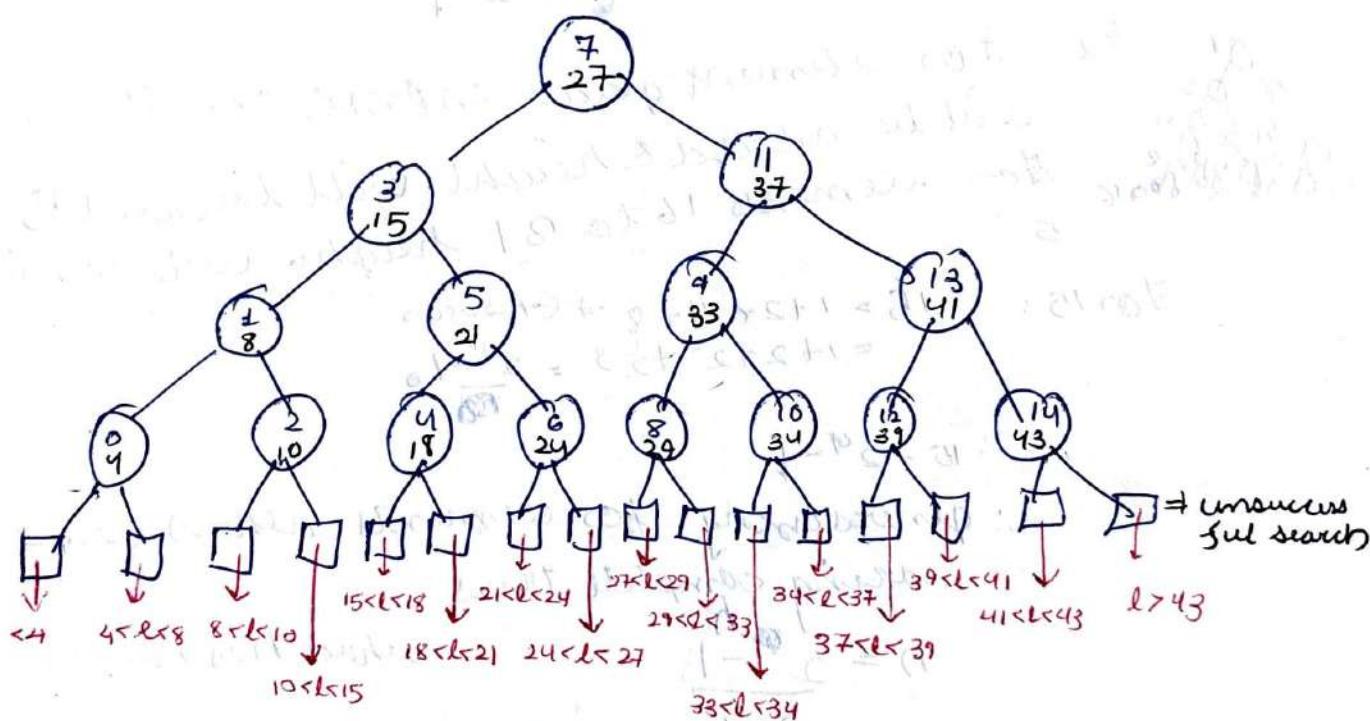
↓ element
↓ paths
↓ pairs
↓ elements
ie 15, 37

$$1 + 1 \times 2^1 + 2 \times 2^2 + 3 \times 2^3$$

$$\sum_{i=1}^{\log n} i \times 2^i \approx \frac{\log n \times 2}{n}$$

$= \underline{\log n}$

approximate ::
we don't need exact
in Average case
because it becomes
typical.

Best case: $O(1)$ Worst case: $\Omega(\log n)$ 

\Rightarrow here l is element which is searched in unsuccessful search operations.

Now if there are three paths & 3 comparisons to reach from root to 1 for key = 8 Comparisons \rightarrow time complexity
 \rightarrow path + 1 = 2 + 1 = 3

No of Comparisons for any Node \geq Path + 1

○ ones are called Internal Nodes

□ ones are called External Nodes representing unsuccessful search.

\therefore To be exact $\lceil \log_2(n+1) \rceil$ - Time Complexity (63)

$$\text{for } 16 \text{ elements} \quad \frac{16}{2} = 8 \quad \therefore 16 = 2^4 \\ \frac{8}{2} = 4 \quad a \log_2 = \log 16 \\ \frac{4}{2} = 2 \quad \text{above} = \log_2 16 \\ \vdots \quad \therefore \log_2$$

Now to find height or time complexity for n elements $\lceil \log_2(n+1) \rceil$ ie for 15

$$\lceil \log_2(16) \rceil = \lceil \frac{4}{2} \rceil = 4$$

This bracket serves
Shanya's Operato

$$\text{For } n=16 \quad \lceil \log_2(17) \rceil = \lceil \frac{4}{2} \rceil = 5$$

ex # as element greater than \leq one leaves will be added & height will become 5
for elements 16 to 31 height will remains 5

$$\text{For } 15: \quad 15 = 1 + 2 + 4 + 8 \rightarrow \text{Gr. terms} \\ = 1 + 2 + 2^2 + 2^3 = \frac{2^4 - 1}{2 - 1}$$

$$\therefore 15 = 2^4 - 1$$

: generalising for elements which are making complete trees

$$n = \frac{2^h - 1}{2 - 1} \quad \text{where } h \text{ is height}$$

$$n = 2^h - 1 \Rightarrow n + 1 = 2^{h+1}$$

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

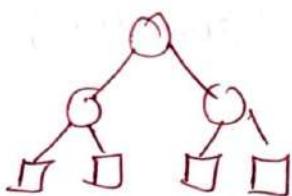
$$\lceil h = \log_2(n+1) \rceil$$

For elements when they make incomplete tree we use shanya operato $n = \lceil \log_2(n+1) \rceil$ ex: 16 will make one tree in 5th height.

- ⇒ Let I denotes sum of the paths of all internal nodes which are representing successful search
 ⇒ E is total paths of all external nodes i.e sum of paths of all external nodes

$$\Rightarrow E = I + 2n \quad \text{away from no of nodes (internal)}$$

ex.



$$\begin{aligned} n &= 3 \\ I &= 1+1=2 \\ E &= 2+2+2+2=8 \end{aligned}$$

$$\begin{aligned} E &= I + 2n \\ 8 &= 2 + 2 \times 3 \end{aligned}$$

$$\Rightarrow E = i + 1$$

no of
external
nodes

$$A_s(n) = 1 + \frac{I}{n}$$

Average
successful
search

$\because \text{path} + 1 = \text{no of comparisons}$
 $= \text{time complexity}$

$$A_u(n) = \frac{E}{n+1}$$

Now $E = (n+1) \log(n+1)$ \Rightarrow since height of tree will be equal to path.

$$\begin{aligned} \therefore A_u(n) &= \frac{(n+1) \log(n+1)}{(n+1)} && \begin{aligned} (n+1) \text{ elements have} \\ \text{path} = \log(n+1) \end{aligned} \\ &= \log(n+1) \approx \log n \end{aligned}$$

$$A_s(n) = 1 + \frac{I}{n} = 1 + \frac{E - 2n}{n} = 1 + \frac{E}{n} - 2$$

$$\therefore \frac{n \log n - 1}{n} \approx \log n$$

1) Get (Index)

if (index $\geq 0 \text{ and } index < lengths$)
return A[index];

2) Set (index)

if (index $\geq 0 \text{ and } index < lengths$)
A[index] = x;

3) Max ()

1 — max = A[0]

n — for (i=1; i<lengths; i++)

1 — if ($A[i] > max$)

n-1 — max = A[i];

}

1 — return max

2n+1 ... O(n)

4) Min ()

min = A[0]

for (i=1; i<lengths; i++)

1 — if ($A[i] < min$)
min = A[i];

}

return min;

5) sum()

(67)

1 - Total = 0;
 n+1 for ($i=0; i < \frac{n}{n+1}$; $i++$)
 n { Total = Total + A[i]; }
 2 - returns Total
 $\overline{2n+3} : O(n)$

But we consider only $n+1$ while calculating time complexity.

→ Recursion. Function of last element

$\sum(A, n) = \begin{cases} 0 & n < 0 \\ \sum(A, n-1) + A[n] & \text{else} \end{cases}$

int sum(A, n)
 {
 if (n < 0)
 return 0;
 else
 return sum(A, n-1) + A[n];
 }

6) Avg()

Total = 0;
 for ($i=0; i < \frac{n}{length}; i++$)

{ Total = Total + A[i]; }
 }
 return Total / length;

(68)

Operations :

- 1) Reverse
- 2) Left shift.
- 3) Left rotate
- 4) Right shift.
- 5) Rightrotate

Reverse:

First Method: Using extra auxiliary array

A	<table border="1"> <tr><td>8</td><td>3</td><td>9</td><td>15</td><td>6</td><td>10</td><td>7</td><td>2</td><td>12</td><td>4</td></tr> </table>	8	3	9	15	6	10	7	2	12	4	i
8	3	9	15	6	10	7	2	12	4			

B	<table border="1"> <tr><td>4</td><td>12</td><td>2</td><td>7</td><td>10</td><td>6</td><td>15</td><td>9</td><td>3</td><td>8</td></tr> </table>	4	12	2	7	10	6	15	9	3	8	j
4	12	2	7	10	6	15	9	3	8			

for ($i = \text{length} - 1; j \geq 0; i \geq 0, j \leftarrow -j, j++$)

$\frac{n+1}{n}$ $\frac{1}{1}$

not concerned $\frac{1}{1}$

$\frac{n}{n}$ $\frac{1}{1}$

$\frac{1}{1}$

$\frac{B[j]}{B[j]} = A[i];$

for ($i = 0; i < \text{length}; i++$)

$\frac{n}{n}$ $\frac{1}{1}$

$\frac{1}{1}$

 $\therefore O(n)$

Second Method : swap

for ($i = 0, j = \text{length} - 1; i < j, i++, j--$)

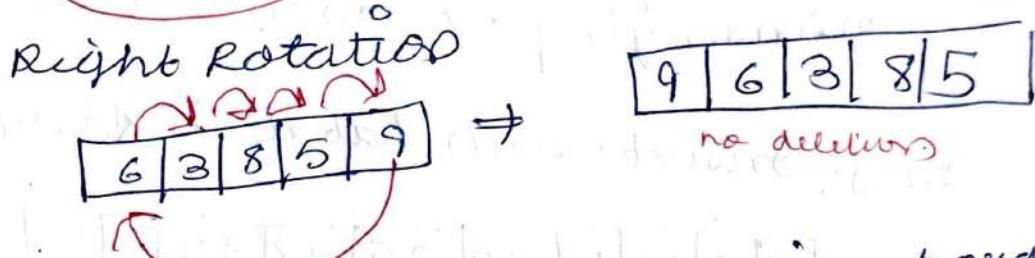
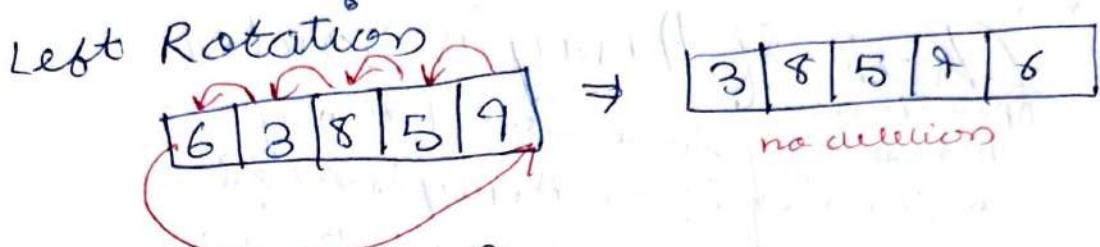
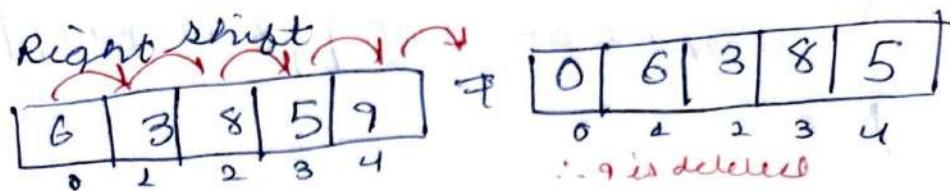
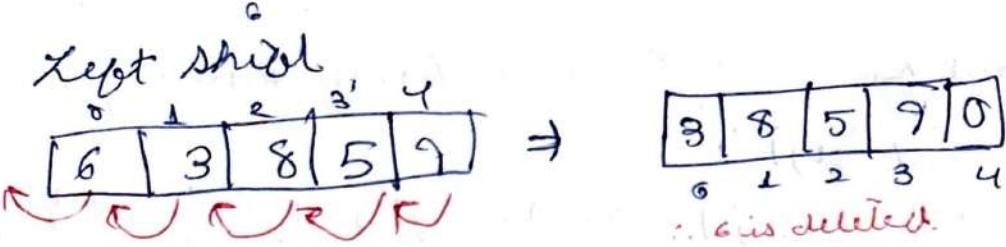
A temp = A[i],

A[i] = A[j],

A[j] = temp;

$\frac{V//}{V//}$ $\frac{1}{1}$

$\frac{\backslash\backslash}{\backslash\backslash}$ $\frac{1}{1}$



shifting is used in LED display boards
to make words scrolling.

Code:

```
void reverse(struct Array *arr)
{
    int *B;
    int i, j;
    B = (int *)malloc((arr->length * sizeof(int)));
    for (l = arr->length - 1; j = 0; i >= 0; i--, j++)
        B[j] = arr->A[i];
    for (i = 0, j = arr->length - 1; i < j; i++)
        arr->A[i] = B[j];
}
```

(70)

void reverse (struct Array *arr)

{ int i, j;

for (i=0; j=arr->lengths-1; i<j; i++, j--)
{ swap(&arr->A[i], &arr->A[j]);

}

↓
Check If Array is Sorted.

Operations:

- 1) Inserting in a sorted Array
- 2) Checking if an Array is sorted.
- 3) Arranging -ve on left side.

Ques. Insert such that it is sorted

A	4	8	13	16	20	25	28	33		
	0	1	2	3	4	5	6	7	8	9

Insert -18

Checking with last element is greater
we will store element next to it.

$$x = 18$$

$$i = \text{lengths} - 1$$

while ($x < A[i]$){ $A[i+1] = A[i];$

$$i = j$$

y

$$A[i+1] = x;$$

Ans How to check whether list is sorted or

Note

Algorithm isSorted(A, n)

for (i=0; i<n-1; i++)

{ if ($A[i] > A[i+1]$)

return false

}

return true;

Time complexity
 $O(n)$ Min: $O(1)$ Max: $O(n)$

71

-ve on Left side.

```

i = 0
j = len(A) - 1
while(i < j)
{
    while(A[i] < 0) { i++; }
    while(A[j] > 0) { j--; }
    if (i < j)
        swap(A[i], A[j]);
}

```

Example:

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

Step 1 a) $i=0$, b) $j=9$

Step 2 a) $i=1$ b) $j=8$ c) Now swap

-6	-4	-8	10	5	-7	-9	12	3	2
----	----	----	----	---	----	----	----	---	---

Step 3 a) $i=3$ b) $j=6$ c) Now Swap

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

Step 4 a) $i=4$ b) $j=5$ c) Now Swap

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

Step 5 a) $i=5$ b) $i=4$ c) Does not fulfill if condition
∴ cannot swap

Step 6: Does not fill while condition ∴ program ends.

72

Merge Arrays

Bendix Operations

Other Binary Operations in Array

- Append
 - Concat
 - Compare
 - Copy

Merging Array can be done only in Sorted Arrays

A	<table border="1"><tr><td>3</td><td>8</td><td>16</td><td>20</td><td>25</td></tr><tr><td>↑ i=0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr></table>	3	8	16	20	25	↑ i=0	1	2	3	4	m	i = 0 $\sqrt{20}$ $k \geq 0$ while ($i < m \& i < n$) { if ($A[i] < B[i]$) $C[k++]$ = $A[i++]$; else $C[k++]$ = $B[i++]$; } gives arrays A and B										
3	8	16	20	25																			
↑ i=0	1	2	3	4																			
B	<table border="1"><tr><td>9</td><td>10</td><td>12</td><td>22</td><td>23</td></tr><tr><td>↑ i=0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr></table>	9	10	12	22	23	↑ i=0	1	2	3	4	n	P = 1										
9	10	12	22	23																			
↑ i=0	1	2	3	4																			
C	<table border="1"><tr><td>3</td><td>4</td><td>8</td><td>10</td><td>12</td><td>16</td><td>20</td><td>22</td><td>23</td><td>25</td></tr><tr><td>↑ k=0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr></table>	3	4	8	10	12	16	20	22	23	25	↑ k=0	1	2	3	4	5	6	7	8	9	g	for (i = 0; i < m; i++) $C[k++]$ = $A[i]$;
3	4	8	10	12	16	20	22	23	25														
↑ k=0	1	2	3	4	5	6	7	8	9														

Here for Terne completely

Copying of elements a
Comparisons of elements

Here copying is taken into

circumstances, many elements
are copied

$\therefore \text{O}(\text{m+n})$

$\therefore \text{Ocm}^2 \text{in}^{-1}$) $\text{for wind}, \rho = \rho_0 (d + \rho_0 h)$

Set operations on Array:

(73)

Union, Intersection & Difference.

Union:

unordered
array

A	<table border="1"> <tr> <td>3</td><td>5</td><td>10</td><td>4</td><td>6</td></tr> </table>	3	5	10	4	6	m
3	5	10	4	6			
	↓ ↓ ↓ ↓ ↓						

B	<table border="1"> <tr> <td>12</td><td>4</td><td>7</td><td>2</td><td>5</td></tr> </table>	12	4	7	2	5	n
12	4	7	2	5			
	↓ ↓ ↓ ↓ ↓						

C	<table border="1"> <tr> <td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr> <tr> <td>3</td><td>5</td><td>10</td><td>4</td><td>8</td><td>12</td><td>7</td><td>2</td><td></td></tr> </table>	0	1	2	3	4	5	6	7	8	3	5	10	4	8	12	7	2		
0	1	2	3	4	5	6	7	8												
3	5	10	4	8	12	7	2													

List all elements of OA
are copied then elements
of B are copied only by
one if they are not
present already.

Time complexity: $m + m \neq n$

$$= O(m+n) \\ = O(n^2)$$

Sorted
Array.

A	<table border="1"> <tr> <td>3</td><td>4</td><td>5</td><td>6</td><td>10</td></tr> </table>	3	4	5	6	10	M
3	4	5	6	10			
	↓ ↓ ↓ ↓ ↓						

B	<table border="1"> <tr> <td>2</td><td>4</td><td>5</td><td>7</td><td>12</td></tr> </table>	2	4	5	7	12	n
2	4	5	7	12			
	↓ ↓ ↓ ↓ ↓						

C	<table border="1"> <tr> <td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>10</td><td>12</td></tr> </table>	2	3	4	5	6	7	10	12	
2	3	4	5	6	7	10	12			
	↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓									

Time By Merging
(similar to merging
diff is just that,
some element will
be copied only
once)

$O(m+n)$

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

$= O(n)$

Intersection:

common elements of both array are copied

A	<table border="1"> <tr> <td>3</td><td>5</td><td>9</td><td>4</td><td>6</td></tr> </table>	3	5	9	4	6	m
3	5	9	4	6			
	↓ ↓ ↓ ↓ ↓						

B	<table border="1"> <tr> <td>12</td><td>4</td><td>7</td><td>2</td><td>5</td></tr> </table>	12	4	7	2	5	n
12	4	7	2	5			
	↓ ↓ ↓ ↓ ↓						

C	<table border="1"> <tr> <td>5</td><td>9</td><td></td><td></td><td></td><td></td><td></td></tr> </table>	5	9						
5	9								
	↓ ↓ ↓ ↓ ↓								

All m elements are searched in n elements
of B if found is copied in C

Time $O(m \times n) = O(mn)$
complexity

Sorted Array

A	3	4	5	6	10	m
---	---	---	---	---	----	---

B	2	4	5	7	12	n
---	---	---	---	---	----	---

C	4	5	7	10	12	15	17	17	17	17	17	17
---	---	---	---	----	----	----	----	----	----	----	----	----

Time Complexity : $O(m+n)$; $O(n)$

Difference

A - B means element of A which are present in B

Unsorted =

A	3	5	10	14	6
---	---	---	----	----	---

B	12	4	7	2	5
---	----	---	---	---	---

C	3	10	6				
---	---	----	---	--	--	--	--

$m \times n$ $O(n^2)$

m

n

All elements of A
are searched in B
if element not found
will be copied
to C

Set Membership Operation

whether an element belongs to the set or not

Same as Searching

Code : (For sorted arrays)

struct Array * Union (struct Array * arr1,

struct Array * arr2)

{

int i, j, k;

i = j = k = 0

struct Array * arr3 = (struct Array *)

malloc(sizeof(struct Array))

(75)

while ($i < \text{arr1.length} \leq j < \text{arr2.length}$)

{ if ($\text{arr1}[i] < \text{arr2}[i]$)

$\text{arr3}[k++] = \text{arr1}[i++]$;

else if ($\text{arr2}[j] < \text{arr1}[i]$)

$\text{arr3}[k++] = \text{arr2}[j++]$;

else

{ $\text{arr3}[k++] = \text{arr1}[i++]$,
 $j++$ }

}

{ for ($j; i < \text{arr1.length}; i++$)
 $\text{arr3}[k++] = \text{arr2}[i]$,

for ($i; j < \text{arr2.length}; i++$)

$\text{arr3}[k++] = \text{arr2}[i]$ }

$\text{arr3.length} = k$

$\text{arr3.size} = 10$

return arr3 ;

}

Some as union diff's.

while ($i < \text{arr1.length} \leq j < \text{arr2.length}$)

{ if ($\text{arr1}[i] < \text{arr2}[j]$)

$i++$

else if ($\text{arr2}[j] < \text{arr1}[i]$)

$j++$

else

{ $\text{arr3}[k++] = \text{arr1}[i++]$ }
 $j++$

}

{ $\text{arr3.length} = k$ }
 $\text{arr3.size} = 10$ }

76 some as unres.

start Difference

while ($i < \text{arr1} \rightarrow \text{length}$ & $j < \text{arr2} \rightarrow \text{length}$)

{ if ($\text{arr1} \rightarrow A[i] < \text{arr2} \rightarrow A[j]$)

$\text{arr3} \rightarrow A[k++]$ = $\text{arr1} \rightarrow A[i++]$

else if ($\text{arr2} \rightarrow A[i] < \text{arr1} \rightarrow A[j]$)

$\text{arr3} \rightarrow A[k++]$ = $\text{arr2} \rightarrow A[i++]$

else

{ $i++$;

{ $j++$;

} } $i < \text{arr1} \rightarrow \text{length}$ & $j < \text{arr2} \rightarrow \text{length}$)

for ($i = 0$; $i < \text{arr1} \rightarrow \text{length}$; $i++$)

$\text{arr3} \rightarrow A[k++]$ = $\text{arr1} \rightarrow A[i]$;

for ($i = 0$; $i < \text{arr2} \rightarrow \text{length}$; $i++$)

$\text{arr3} \rightarrow A[k++]$ = $\text{arr2} \rightarrow A[i]$;

return arr3 ;

Code is C++ Download Resources ^{Q7 Lecture}
for Header Driver for all operations functions
Complete Code

#include <iostream>

using namespace std;

class Array {

private:

int *A;

int size;

int length;

void swap(int *x, int *y)

public:

Array(); // constructor

size = 10;

length = 0;

77

```

A = new int [size];
}
Array (int sz)           // constructor when parameter
{                         is given
    size = sz;
    length = 0;
    A = new int [size];
}
~Array ()                // Destructor
{
    delete [] A;
}
void display ();
void Append (int x);
void Insert (int index, int x);
int Delete (int index);
Array * Enter (Array arr2);
};

void Array :: display ()
{
    int i;
    cout << "n elements are n" << endl;
    for (i = 0; i < length; i++)
        cout << A[i] << " ";
}

void Array :: Append (int x)
{
    if (length >= size)
        A[length + 1] = x;
}

void Array :: Insert (int index, int x)
{
    for (int i = index; i < length; i++)
    A[index] = x;
    length++;
}

void operator Array :: Delete (int index)
{
    for (int i = index + 1; i < length; i++)
    A[index] = 0;
    length--;
}

```

array* array :: Enter (Array arr2)

{ int i, j, k

i = j = k = 0

(some part)

array arr3 = new Array (length + arr2.length)

while (i < length && j < arr2.length)

{ if (arr2[i] < arr2[A[j]])

i++;

else if (arr2[A[j]] < arr2[i])

j++;

else if (arr2[i] == arr2[A[j]])

{ arr3[A[k + e]] = arr2[i]

i++

y (some condition) draw line

y (some condition) draw line

y (some part) write & write

arr3.length = k

return arr3;

y { "A" is greater than B

int main()

{ // some code

y

(some part) write & write

(some part) write & write

i = 0 + arr2.length

some part and write & write

some part and write & write

1	2	3	4	5	6	8	9	10	11	12	
0	1	2	3	4	5	6	7	8	9	10	

1) Single Missing Element in Sorted Array.

case I when elements start from 1 as above

$$\text{sum} = 0,$$

$\text{for } i=0; i < n; i++ \}$

{ $\text{sum} = \text{sum} + A[i]$ }

$$k = n * (n+1) / 2$$

$$\text{missing} = s - \text{sum};$$

print "d", missing)

case II when element starting does not is not 1

case II

6	7	8	9	10	11	13	14	15	16	17
0	1	2	3	4	5	x	7	8	9	10

$$l = 6$$

$$h = 17$$

$$n = 11$$

$$\text{diff} = l - 0$$

$\text{for } i=0; i < n; i++ \}$

{ if ($A[i] - i \neq \text{diff}$)

if ($A[i] - i \neq \text{missing element}$) it will

{ print
break;

}

| time complexity : O(n)
(worst case)

2) More than One Missing Element

6	7	8	9	11	12	15	16	17	18	19
0	1	2	3	4	5	6	7	8	9	10
6	6	6	6	7	7	9	9	9	9	9

e.g.:

Missing element

$$6 + 4 = 10$$

$$7 + 6 = 13 \quad 8 + 6 = 14$$

(80)

$$diff = l - 0;$$

for ($i=0; i < n; i+t$)

 if ($A[i]-i \neq diff$)

 while ($diff < A[i]-i$)

 print ("Y. d*(n)", i+diff);
 $diff + t;$

negligible time with
respect to n complexity
is linear.

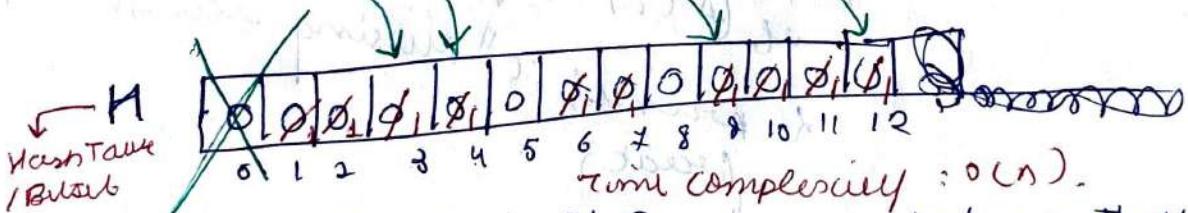
Time complexity $\in O(n)$

Unsorted Array:

How to Find Missing Elements.

A	<table border="1"> <tr> <td>3</td><td>7</td><td>4</td><td>9</td><td>12</td><td>6</td><td>1</td><td>11</td><td>2</td><td>10</td> </tr> <tr> <td>0</td><td>2</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td> </tr> </table>	3	7	4	9	12	6	1	11	2	10	0	2	2	3	4	5	6	7	8	9
3	7	4	9	12	6	1	11	2	10												
0	2	2	3	4	5	6	7	8	9												

$$\begin{aligned} i &= 1 \\ h &= 12 \\ n &= 10 \end{aligned}$$



This H: Hash Table ~~is O(n)~~ Because whatever that element is we are going to that same index and accessing it. (SIMPLE FORM OF HASH)
 REMEMBER: HASH TABLE TAKES CONSTANT TIME

In H now from index 0 to 12 those elements are missing on which element at that index = 0

NOTE: HASH TAKES LARGER SPACE SINCE SIZE OF HASH TABLE HAS TO BE EQUAL TO LARGEST NO IN Array A. Hashing is best solution whenever we are searching where there is no space restriction

$\text{for } (i=0; i < n; i++)$

scanning through A

1 $H[A[i]]++;$ $\rightarrow n$

scanning through H

2 $\text{for } (i=0; i < h; i++)$

3 if ($H[i] == 0$) $\text{printf}("ydn", i); \rightarrow n$

$\rightarrow 2n$

Time Complexity: $\underline{\underline{o(n)}}$

Finding Duplicate Elements

1) Finding Dup. Elems. in Sorted Array

A	3	6	8	8	10	12	15	15	15	20
	0	1	2	3	4	5	6	7	8	9

$n=10$

lastDuplicate = 0;

$\text{for } (i=0; i < n; i++)$

1 if ($A[i] == A[i+1]$) $\& A[i] != \text{lastDuplicate}$

2 $\text{print}("ydn", A[i]);$

lastDuplicate = $A[i];$

}

Ques: How to count How many times
Duplicate elements is repeating
if is keeping to count.

$\text{for } (i=0; i < n-1; i++)$

1 if ($A[i] == A[i+1]$)

2 $j = i+1;$

⑫

while ($A[i] == A[j+1]$) $\downarrow i++;$

printf ("%.d is appearing %d times", $A[i]$, $j-i$)

y

y

$O(n)$: Time complexity

method 2: By Hash Table.

A	3	6	8	8	10	12	15	15	15	26
	0	1	2	3	4	5	6	7	8	9

11

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

∴ Incrementing takes constant time

Accessing all elements of A takes n time

∴ Accessing element of A to count: k_n

$\therefore O(n)$ (as k_n is constant for n elements)

for ($i=0$; $i < n$; $i++$)

{ $H[A[i]]++$ } $\Rightarrow n$

for ($i=0$; $i <= A[9]$; $i++$)

D [if ($H[i] > 1$)]

{ printf ("%.d appears %d times", i , $H[i]$) }

∴ Time complexity: $n + n = \underline{\underline{2n}}$

Finding Duplicates in Unsorted.

Method I

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

$n=10$

```
for(i=0; i<n-1; i++)
```

```
{ count = 1;
  if(A[i] == j = 1)
```

```
{ for(j = i+1; j < n; j++)
```

```
{ if(A[i] == A[j])
```

```
{ count++;
  A[j] = -1;
```

almost n times

```
} if(count > 1) cout << "y.d" << "d", A[i], count);
```

∴ Time complexity: $\underline{\underline{O(n^2)}}$

Method II: Using Hash Table.

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

$n=10$

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

$\underline{\underline{O(n)}}$

Same logic as in previous Hash Table programs.

84

Ques: If a list of elements are given then we have to find out a pair of elements that is two elements such that their sum is equal to a given no.

Find a pair with sum k. ($a+b=k$)

Method 1

A	6	3	8	10	16	7	5	2	9	14	$a+b=10$
	0	1	2	3	4	5	6	7	8	9	

there are no duplicate elements
if there are duplicates remove duplicates
and then perform these operations

$\text{for}(i=0; i < n-1; i++)$

{ $\text{for}(j=i+1; j < n; j++)$

 { if ($A[i] + A[j] == k$)

 printf("y. d + %d = %d", A[i], A[j]);

4

{

$n-1 + n-2 + \dots + 1$

$\therefore \frac{(n-1)n}{2}, \therefore \underline{\underline{O(n^2)}}$

Method 2: Hashing

A	6	3	8	10	16	7	5	2	9	14
	0	1	2	3	4	5	6	7	8	9

0	0	0	9	0	0	9	0	9	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

: $O(n)$

(85)

for ($i=0$; $i < n$; $i++$)

{

 if ($A[i] \leq k$) { ~~exited~~ $y[n[k - A[i]]] = 0$ { prints " $y.d + y.d = y.d$ ", $A[1], A[1], k - A[1]$
 $, k$); $y[n[A[i]]] +=$ $\underline{O(n)}$: Time complexity

Finding Pair of Elements in Sorted Array.

A	$\begin{array}{ c c c c c c c c c } \hline 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ \hline 1 & 3 & 4 & 5 & 6 & 8 & 9 & 10 & 12 & 14 \\ \hline \end{array}$	$\{a+b=16\}$
-----	--	--------------

 $i = 0$; $j = n - 1$ while ($i < j$) { if ($A[i] + A[j] == k$) { printd " $y.d + y.d = y.d$ ", $A[1], A[1], 1, 1$, $i++$ $j--$ y else if ($A[i] + A[j] < k$) { $i++$ y

else

 $j--$ y time complexity : i is scanning some part & j is scanning other part: $O(n)$:- It will be $O(n)$

(86)

Finding MAX and MIN in a Single Scan

No use ... \Rightarrow same logic of max & min.

($i=1, j=n$)

(with loop invariant)

$\text{Max}(b) = b[i] + b[i+1] \dots b[n]$

($i=1$)

$\Rightarrow i=1, j=n$

(loop invariant condition)

loop invariant will be found here.

($i=1$)

Max	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]	[16]	[17]	[18]	[19]	[20]	[21]	[22]	[23]	[24]	[25]	[26]	[27]	[28]	[29]	[30]	[31]	[32]	[33]	[34]	[35]	[36]	[37]	[38]	[39]	[40]	[41]	[42]	[43]	[44]	[45]	[46]	[47]	[48]	[49]	[50]	[51]	[52]	[53]	[54]	[55]	[56]	[57]	[58]	[59]	[60]	[61]	[62]	[63]	[64]	[65]	[66]	[67]	[68]	[69]	[70]	[71]	[72]	[73]	[74]	[75]	[76]	[77]	[78]	[79]	[80]	[81]	[82]	[83]	[84]	[85]	[86]	[87]	[88]	[89]	[90]	[91]	[92]	[93]	[94]	[95]	[96]	[97]	[98]	[99]	[100]
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	-------

$$Max = i; i = 1$$

($i=1, j=n$)

($i=1, j=n$)

($i+1$)

($i+1$)

($i+1, j=n$)

($i+1$)

($i+1$)

($i+1$)

($i+1$)

Strings:

- 1) Character Set / ASCII Codes
- 2) Character Array
- 3) String
- 4) Creating a String

ASCII codes

For every character we define numeric value

Unicodes : <small>for various national languages</small>		
<small>we define numerical values called <u>character of</u> <u>unicodes</u></small>		
A = 65	a = 97	0 - 48
B = 66	b = 98	1 - 49
C = 67	,	2 - 50
;	!	!
Z = 90	Z = 122	9 - 57

Enter ↲ : 10 Space : 13 Esc : 27

ASCII CODES ARE STARTING FROM 0 to 127

Unicodes But 7 bits cannot be stored in memory. i.e. 8 bits are stored for characters

Take 2 bytes = 16 bits \Rightarrow It can be represented in form of Hexadecimal.

∴ Unicodes are represented in form of Hexadecimal

\therefore Hexadecimal can be represented in

4 bits : $16 \text{ bits} = 4 \times 4 \text{ bits}$

\therefore It can be represented as 4

Hexadecimal
as CO 3A

char temp; \rightarrow 16416

temp = 'A';

~~temp = 'AB';~~

~~temp = A;~~

~~temp = "A";~~

~~temp = "AB";~~

How to print?

\Rightarrow printf("%c", temp); // A

\Rightarrow printf("%d", temp); // 65

\Rightarrow cout < temp; // A

temp
A

A actually is memory
65 is stored

Array of characters

\Rightarrow char x[5];

\Rightarrow char x[5] = { 'A', 'B', 'C', 'D', 'E' };

\Rightarrow char x[] = { 'A', 'B', 'C', 'D', 'E' };

\Rightarrow char x[5] = { 65, 66, 67, 68, 69 };

char x[5] = { A, B, C }

x	A	B	C	D	E
	0	1	2	3	4

what are strings

Array of character char name[] i.e. { 'J', 'o', 'h', 'n', '\0' }

name	J	o	h	n	0	0	0	0	0
	0	1	2	3	4	5	6	7	8

size : 10

String:

This is to show end of string

name	J	o	h	n	0	0	0	0	0
	0	1	2	3	4	5	6	7	8

10 String delimiter
end of string class
NULL caract.
String Terminator

In memory, length of string is known by Null character.

`char name[10] = { 'J', 'o', 'h', 'n', ' ', 'l', 'o' };`
Now this is String

Methods of Creating Strings

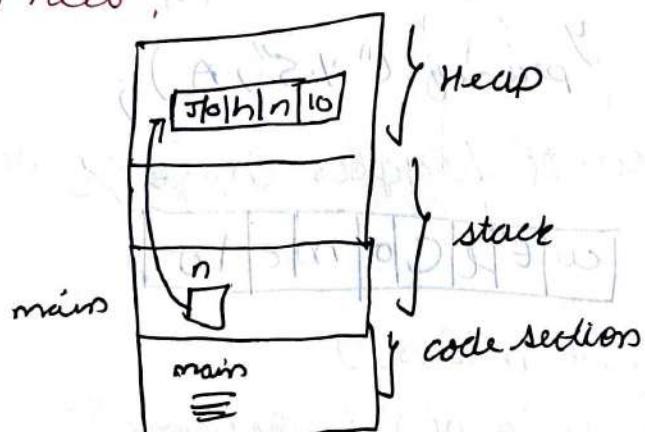
`char name[10] = { 'J', 'o', 'h', 'n', ' ', 'l', 'o' };`

`char name[] = { 'J', 'o', 'h', 'n', ' ', 'l', 'o' };`

`char name[] = "John";`

~~int~~ ~~*~~ `char *name = "Johns";`

Automatically created in heap
though we have not created malloc or new.



`printf("%s", name);`

`scanf("%s", name);` // will input

To take input of multiple words; single word

→ `gets(name);` would take input until we hit enter.

→ Finding Length of String

`s [w | e | l | c | o | m | e | / | 0]`

`int main()`

`char *s = "welcome";`

`int i; for (i=0; s[i] != '\0'; i++)`

`return i;`

(40)

Changing Case of String

$A - 65$	$a = 97$	$97 - 65 = 32$
$B - 66$	$b = 98$	$98 - 66 = 32$
$:$	$:$	
$Z - 90$	$z = 122$	$122 - 90 = 32$

A

W	E	L	U	C	O	M	E	10
---	---	---	---	---	---	---	---	----

From upper case to lower case
int main()

```

    {
        char s[ ] = "WELCOME";
        int i;
        for(i=0; s[i] != '\0'; i++)
            {
                A[i] = s[i] + 32;
            }
        printf("%s", A);
    }

```

How to Toggle Case of String

A

w	E	L	C	O	m	c	10
---	---	---	---	---	---	---	----

int main()

```

    {
        char A[ ] = "welcome";
        int i;
        for(i=0; A[i] != '\0'; i++)
        {
            if ((A[i] >= 65 & A[i] < 90))
                A[i] += 32;
            else if ((A[i] >= 'a' & A[i] <= 'z'))
                A[i] -= 32;
        }
        printf("%s", A);
    }

```

printf("%s", A);

5

Counting Vowels & Consonants.

A H0w l4bke l4y0u l0

int main()

{ char A[] = "How are you";

int i, vcount=0, consonant=0;

for(i=0, A[i] != '0'; i++)

{ if (A[i] == 'a') || A[i] == 'e'

|| A[i] == 'A' || A[i] == 'E' ...)

consonant++;

else if ((A[i] >= 65 & A[i] <= 90) ||

(A[i] >= 97 & A[i] <= 122)

vcount++;

}

How to Count Words in String

no of words = no of spaces + 1

int main()

{ char A[] = "How are you";

int i, word=0;

for(i=0; A[i] != '0'; i++)

{ if (A[i] == ' ')

word++;

5

printf ("%d", word);

Here this condition says that every word have only one space in between

(42)

For care like

name	take		lulco								
0	1	2	3	4	5	c	7	8	9	10	11

white spaces

int main()

{ char A[] = "How are u";

int i, word = 1;

for(i=0; A[i] != '\0'; i++)

{ if(A[i] == '-' & A[i-1] == '_')

word++;

printf("%d", word);

Validall a String

for ans as valid password or not

Here if string has any special characters
then it will be invalid

name [A|n|i?|3|2|1|1|0]

int valid(char *name)

{ int i;

for(i=0; name[i] != '\0'; i++)

{ if((name[i] >= 65 & name[i] <= 90) &&

i[name[i] >= 91 & name[i] <= 122) &&

{ name[i] >= 48 & name[i] <= 57)}

return 0;

} return 1; // denote valid

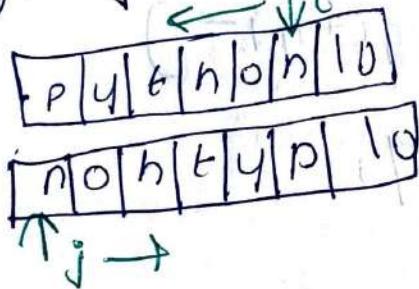
```

int main()
{
    char phone = "Anil321";
    if (isValidName())
        { printf ("Valid String");
    }
    else
        { printf ("Invalid String");
    }
}

```

Reversing a String

Method by using other character array



```

int main()
{
    char A[7] = "python";
    char B[7];
    int i;
    for(i=0; A[i] != '\0'; i++)
    {
        i = i - 1;
        for(j=0; i >= 0; i--, j++)
        {
            B[j] = A[i];
        }
        B[i] = '\0';
        printf("%s", B);
    }
}

```

Q4) Method 2: Swap

~~NOTE~~ In C/C++ compiler, strings are not mutable
∴ If we declare string by using pointers of
char & then it will be immutable ∴ to
perform mutable operations declare
strings as array of character

int main()

{ char A[] = "python"; }

char t;

int i, j;

for (j = 0; A[j] != '\0'; j++)

{

} ; ~j-1;

for (i = 0; i < j; i++, j--)

{ t = A[i];

A[i] = A[j];

A[j] = t;

}

printf("y.s%", A);

}

How to compare two strings:

A

P	a	i	n	t	e	r	t	o
i	1	2	3	4	5	6	7	

P	a	i	n	t	i	n	g	b
i	1	2	3	4	5	6	7	8

int main ()

{ char A[] = "Painter";

char B[] = "Painting";

int i, j;

for (i = 0; j = 0; A[i] != '0' & B[j] != '0'; i++, j++)

{ if (A[i] == B[i])

break;

if (A[i] == B[i])

printf ("equal");

else if (A[i] < B[i])

printf ("smaller");

else

printf ("greater");

}

Here this algorithm is Case Sensitive.

ie for Painter & Painter → Painter is will

be lesser than painter.

→ high ASCII value

46

Palindrome

A	<table border="1"> <tr> <td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td></td></tr> <tr> <td>m</td><td>a</td><td>d</td><td>a</td><td>m</td><td>10</td></tr> </table>	0	1	2	3	4		m	a	d	a	m	10
0	1	2	3	4									
m	a	d	a	m	10								

B	<table border="1"> <tr> <td>m</td><td>a</td><td>d</td><td>a</td><td>m</td><td>10</td></tr> </table>	m	a	d	a	m	10	Reverse of A
m	a	d	a	m	10			

Method I:

① Reverse

② Compare $i=0, j=4$

Method 2: $i=0 \quad j=4$ $i+1 \quad j-1$ $i < j$
 $i+1 \quad j-1$ $i \neq j \quad A[i] \neq A[j]$
points ("Not Palindrome")
break;

Finding Duplicate in a String

Methods :

- 1) compare with other elements
- 2) using HashTable as counter
- 3) using Bits.

Method I :

A	<table border="1"> <tr> <td>f</td><td>i</td><td>n</td><td>d</td><td>i</td><td>n</td><td>g</td><td>10</td></tr> <tr> <td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr> </table>	f	i	n	d	i	n	g	10	0	1	2	3	4	5	6	7
f	i	n	d	i	n	g	10										
0	1	2	3	4	5	6	7										

Same as array

Method II :

A	<table border="1"> <tr> <td>102</td><td>105</td><td>110</td><td>100</td><td>105</td><td>103</td><td>100</td><td>10</td></tr> <tr> <td>f</td><td>i</td><td>n</td><td>d</td><td>i</td><td>n</td><td>g</td><td>10</td></tr> <tr> <td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr> </table>	102	105	110	100	105	103	100	10	f	i	n	d	i	n	g	10	0	1	2	3	4	5	6	7
102	105	110	100	105	103	100	10																		
f	i	n	d	i	n	g	10																		
0	1	2	3	4	5	6	7																		

ASCII : 97-122

		1	1	2		2																			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

$$f = 102 - 97 = 5$$

$$i = 105 - 97 = 8$$

$$n = 110 - 97 = 13$$

$$d = 100 - 97 = 3$$

$$g = 108 - 97 = 6$$

int main() {
 char dg = "finding";
 int H[26], i;
 for (i=0; dg[i] != '\0'; i++) H[dg[i] - 'A']++;
 for (i=0; i < 26; i++) {
 if (H[i] == 1) {
 cout << dg[i];
 break;
 }
 }
}

{ char dg = "finding";
int H[26], i;

for (i=0; dg[i] != '\0'; i++) H[dg[i] - 'A']++;

H[dg[i] - 'A']++ = 1;

dg[i] = 102 - 97 = 5
H[5]++ = 1

cout << dg[i];

for (i=0; i < 26; i++) {
 if (H[i] == 1) {
 cout << dg[i];
 break;
 }
}

dg[i] = 102 - 97 = 5
H[5]++ = 1

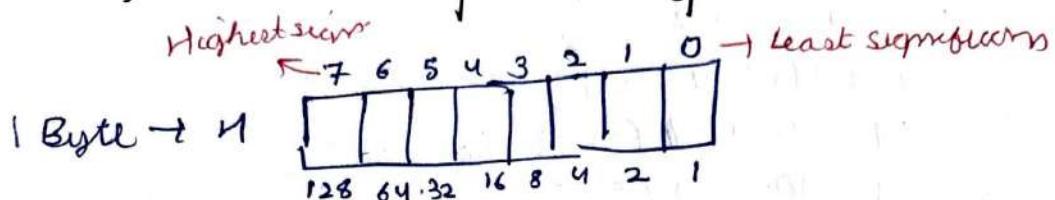
Time Complexity $O(n)$: $n+n$

Method 3: Bitwise
Also useful for duplicates or int. but most
space efficient & for strings

100 98

Bitwise Operation

- 1) Left shift \ll
- 2) Bits ORing (Merging)
- 3) Bits ANDing (Masking)



~~Step 1: $H = 18 \dots 8 + 2 \dots$~~ ~~00001010~~
~~↓~~
~~↓~~

~~Step 2: $H = 20 \dots 16 + 4 \dots$~~ ~~00010100~~

⇒ Left operation

~~00~~
 $H = 1$

$H = 1 \ll 5$

~~00~~
 $H = 32 \dots 5 \times 2^5$ multipliers

+ And operation

$a = 10 =$	1010	$b = 6 =$	0110
$a = 10 =$	1010	$b = 6 =$	0110
$a \& b =$	0010	$= 2$	

OR operation

$a = 10 : 1010$

$b = 6 : 0110$

$a \vee b = 1110 = 14$

$a \vee b$	1	$1 = 1$
1	1	$0 = 0$
1	0	$1 = 0$
0	1	$0 = 0$
0	0	$0 = 0$

Masiney:

To find that it is 0 or 1 on that particular bit.

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

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

Suppose now we have to find at position 2 whether it is 0 or 1

thus

$$a = a \ll 2$$

0	0	0	1	0	1	0	0
---	---	---	---	---	---	---	---

$$a \& H$$

$$\text{if } (a \& H == 0)$$

then "OFF"

else

"ON"

Merging

To give 1 on a particular bit we use merging

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

a	0	0	0	1	0	0	1	
---	---	---	---	---	---	---	---	--

$$a = 1$$

$$a = a \ll 2$$

$$H = a \& H$$

H	0	0	0	0	1	0	0	
---	---	---	---	---	---	---	---	--

H	0	0	0	1	1	0	0	
---	---	---	---	---	---	---	---	--

H	0	0	0	1	1	0	0	
---	---	---	---	---	---	---	---	--

Now we will use all these three operators for finding duplicates in a string.

100

A

102	105	110	108	105	110	103
finding 10						

Now it should be of 26 bit. But: 26 bits is not possible to allocate in memory

\therefore we take 32 bit = 4 byte (long int)

it stores in H according to index
 $i = \text{index} - 102 + 97$

0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 1 0 0 0 0 1	0 1 1 0 1 0 0 0 0
31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0

now for f: $102 - 97 = 5$

\therefore at index 5 there is zero value. \therefore we will perform merging

But if there is a present ~~zero~~ (we have to see masking) then it ~~is~~ is ~~zero~~ duplicates in a string

int main()

{ char A[] = "finding";
 long int H = 0, x = 0;

for (i=0; A[i] != '\0'; i++)

{

x = x << A[i] - 97;



if (x & H > 0)

masking

{ printf("%c is Duplicate, A[%d]\n", A[i], i); }

else

H = x | H; // Merging

y

y

Checking if 2 strings are Anagrams

A	decimal	10
	1 0 0 1 0 1 9 9 1 0 5 1 0 7 9 7 1 0 8	
B	medical	10

ie the strings have same form
ed by same character as
other string.

Step I: we have to see whether they have same size or not.

Step II: ~~Method I:~~

Method I:
Checking each element of A in B if it is found
replace it by 0 in B & if it is not found in B
then they are not Anagrams.
But Time Complexity $n \times D = n^2$

Method II: By Hash Table

We will create Hash Table by incrementing by
one whenever found in A & would Decr-
ement by 1 whenever found in B

1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

- while scanning B

Time complexity: n for scanning of A
 n for B

we are not fully scanning it but $\leftarrow n \approx n$ for scanning H

$$= 2n$$

\therefore

$O(n)$

int main()

{ char A[] = "decimal" ;

char B[] = "medical" ;

int i ; H[26] = {0} ;

for (i=0 ; A[i] != '\0' ; i++)

{ H[A[i]]++ ; }

102

for ($i=0$; $B[i] \neq '10'$; $i++$)

 ↳ $U[A[i]-97] = 1$

 if ($U[A[i]-97] < 0$)

 ↳ print ("notAnagram");
 break;

y

 ↳ if ($B[i] == '10'$)

 ↳ print ("Anagram");

y

[If both strings don't have any duplicates
then we can check by bits also whether they
are Anagrams or not.]

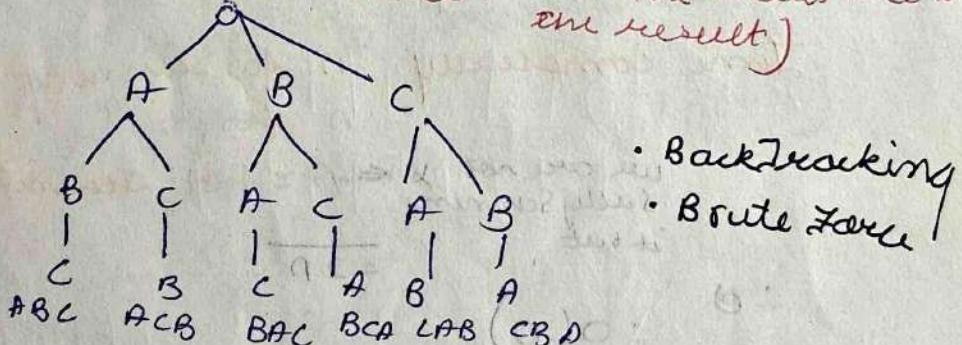
Permutation of a String.

s | A | B | C | 10

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

- 1) ABC
- 2) ACB
- 3) BAC
- 4) BCA
- 5) CAB
- 6) CBA

State Space Tree (where the leaves are showing the result)



- Backtracking
- Brute Force

Brute Force: Finding out all the permutations
or with any method.

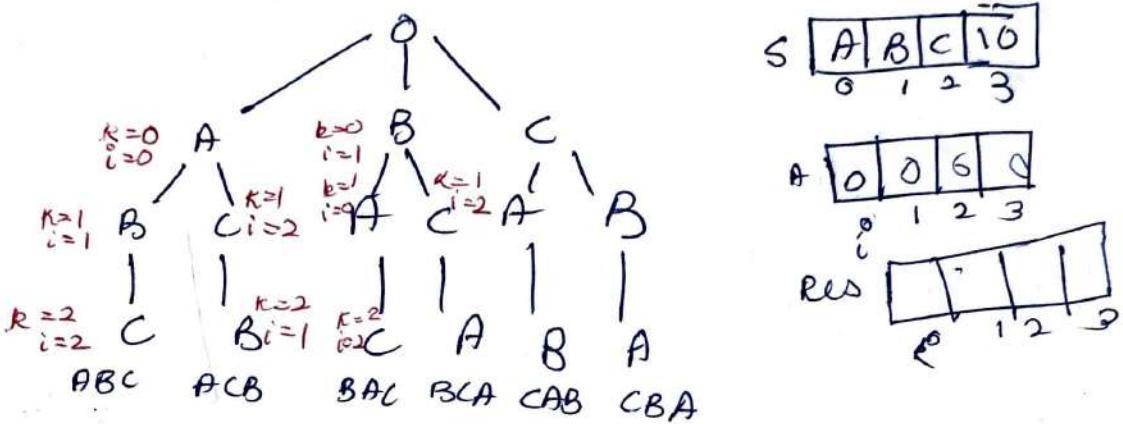
Backtracking: In this we have gone to A then B then C
and then C → B & B → A & then A → C : we are taking
a back track.

- Backtracking.
- Brute Force
- Recursion

⇒ If procedure is backtracking and finding out everything then that procedure is called Brute force

⇒ Backtracking is implemented by Recursion

Method I:



void perm (char S[], int k)

```

    static int A[10]={0};  

    static char Res[10];  

    int b;  

    if (S[k]=='\0')  

    { Res[k]='\0'; print(Res); }  

    else  

    { for (i=0; S[i]!='\0'; i++)  

        if (A[i]==0)  

        { res[k]=S[i];  

          A[i]=1;  

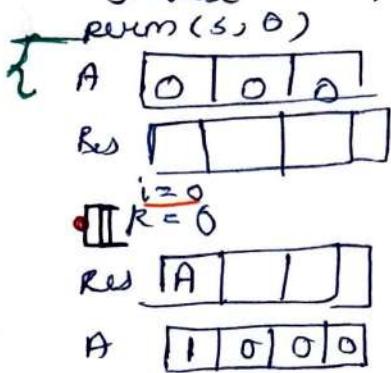
          perm(S,k+1);  

          A[i]=0; }
    }
  
```

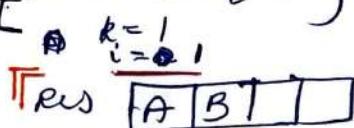
}

Method I Tracing

~~Case 2 with i > j
Index = n + 1~~



perm($S, 1$)



A [1 | 1 | 0 | 0]

perm($S, 2$)

k=2
i=2

Res [A | B | C |]

A [1 | 1 | 1 | 0]

perm($S, 3$)

Res [A | B | C | D]

\Rightarrow PS(ABC)

A [1 | 1 | 0 | 0]

i=3

Done

A [1 | 0 | 0 | 0]

i=2

Res [A | C | D | 0]

A [1 | 0 | 1 | 0]

perm($S, 2$)

i=1

Res [A | C | B | 0]

A [1 | 1 | 1 | 0]

perm($S, 3$)

Res [A | D | C | 0]

\Rightarrow PS(ACB)

A [1 | 0 | 1 | 0]

i=2

i=3

A [1 | 0 | 0 | 0]
i=3

A [0 | 0 | 0 | 0 | 0] $\because A[0] = 0$
i=1

i=1; k=0
 Res [B | F | A | 10]

A [0 | 1 | 0 | 0]

perm($S, 1$)

i=0
k=1

Res [B | A | B | 10]

A [1 | 1 | 0 | 0]

perm($S, 2$)

k=2

Res [B | A | C | 10]

A [1 | 1 | 1 | 0]

perm($S, 3$)

Res [B | A | C | 10]

\Rightarrow PS(CBA)

A [1 | 1 | 0 | 0]

i=3

A [0 | 1 | 0 | 0]

i=1

i=2

Res [B | C | C | 10]

A [0 | 0 | 1 | 0]

perm($S, 2$)

i=0
k=2

Res [B | C | A | 10]

A [1 | 1 | 1 | 0]

perm($S, 3$)

Res [B | C | A | 10]

\Rightarrow PS(BCA)

A [0 | 1 | 1 | 0]

i=1
i=2
i=3

1046

A 01010101 3

i=2
000000

01010101

01010101

01010101

01010101

01010101

01010101

01010101

01010101

01010101

01010101

01010101

01010101

01010101

01010101

01010101

01010101

01010101

01010101

01010101

01010101

01010101

01010101

01010101

01010101

01010101

01010101

01010101

01010101

01010101

01010101

MATRIX:

Special Matrices

(105)

- 1) Diagonal Matrix
- 2) Lower Triangular Matrix
- 3) Upper Triangular Matrix
- 4) Symmetric Matrix
- 5) Skewdiagonal Matrix
- 6) Band Matrix
- 7) "Sparse"
- 8) Sparse Matrix

Diagonal Matrix

$$M = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 7 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 & 0 \\ 0 & 0 & 0 & 9 & 0 \\ 0 & 0 & 0 & 0 & 6 \end{bmatrix}$$

$$M[i, j] = 0 \text{ if } i \neq j.$$

5x5

$$A = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 \\ 0 & 3 & 0 & 0 & 0 \\ 1 & 0 & 7 & 0 & 0 \\ 2 & 0 & 0 & 4 & 0 \\ 3 & 0 & 0 & 0 & 9 \\ 4 & 0 & 0 & 0 & 6 \end{bmatrix}$$

5x5

$$25 \times 1 = 25 \text{ bytes}$$

2D Array

Now ~~5x5~~ it is taking 50 bytes
 & it is waste to store 0
 because addition, ~~x~~ will result in zero.
 ∴ we only store diagonal elements
 ∴ 10 bytes are required for that.

Representing diagonal Mat. in 1D array

$$A [\underline{3} \ | \ \underline{7} \ | \ \underline{4} \ | \ \underline{9} \ | \ \underline{6}]$$

$$\quad \quad \quad \begin{matrix} 0 & 1 & 2 & 3 & 4 \end{matrix}$$

Now to create Diagonal Matrix in 1D array -

108 You colour

```
int A[5];
void set(int A[], int i, int j, int x)
{
    if (i == j)
    {
        A[i - 1] = x;
    }
    else
        int get (int A[], int i, int j)
    {
        if (i == j)
            return A[i - 1];
        else
            return 0;
    }
}
```

Diagonal Matrix

#include <stdio.h>

Struct Matrix

```
{ int A[5];
    int n;
}
```

```
void Set (struct Matrix m, int i, int j, int x)
```

```
{ if (i == j)
    m.A[i - 1] = x;
}
```

```
int Get (struct Matrix m, int i, int j)
```

```
{ if (i == j)
    return m.A[i - 1];
    else
        return 0;
}
```

```
void display (struct Matrix M)
```

107

```

int i, j;
for (i=0; i<m.n; i++)
{
    for (j=0; j<m.n; j++)
        if (i==j)
            *prinif ("y. d"), m.A[i]);
        else
            prinif ("0");
        prinif ("\\n");
}

```

b)

```

int main()
{
    start Matrix m,
    m.n = 4;
    set (m, 1, 1, 5), set (m, 2, 2, 8), set (m, 3, 3, 9),
    set (m, 4, 4, 12);
    prinif ("y & n", get (m, 2, 2));
    Resplay (m);
    return 0;
}

```

In C++:

```

class Diagonal
{
private:
    int n; //Dimensions
    int *A;
public:
    Diagonal (int n)
    {
        this->n = n;
        A = new int [n];
        void set (int i, int j, int x);
        int get (int i, int j);
    }
}

```

```

void display();
~Diagonal() { delete[] A; }

void Diagonal::set(int i, int j, int x) {
    if (i == j)
        A[i - 1] = x;
}

int Diagonal::get(int i, int j) {
    if (i == j) return A[j];
    else return 0;
}

void Diagonal::display() {
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            if (i == j) cout << A[i - 1];
            else cout << "0";
        }
        cout << endl;
    }
}

```

Lower Triangular Matrix

$$M = \begin{bmatrix} a_{11} & 0 & 0 \\ a_{21} & a_{22} & 0 \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

3×3

$$M[i][j] = 0 \text{ if } i < j$$

$$M[i][j] = \text{non zero if } i \geq j$$

$$\text{Nonzero} = 1 + 2 + 3$$

$$\text{for n size} = 1 + 2 + 3 + \dots + \frac{n(n+1)}{2}$$

$$\text{Zero} = n^2 - \frac{n(n+1)}{2} = \frac{n(n-1)}{2}$$

109

Representing in 1D array

⇒ Row Major

a_{11}	a_{21}	a_{22}	a_{31}	a_{32}	a_{33}
0	1	2	3	4	5
row 1	row 2		row 3		

$$\text{Index}(A[2][2]) = [1+1] = 2$$

$$\text{Index}(A[3][3]) = [1+2] + 2 = 5$$

$$\text{Index}(A[i][j]) = \left[\frac{i(i-1)}{2} \right] + j - 1$$

⇒ Column Major

a_{11}	a_{21}	a_{31}	a_{41}	a_{51}	a_{22}	a_{32}	a_{42}	a_{52}	a_{33}	a_{43}	a_{53}	a_{44}	a_{54}	a_{55}
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
col 1	col 2		col 3		col 4		col 5		col 6		col 7		col 8	

$$\text{Index}(A[4][4]) = [5+4+3] + 0 = 12$$

$$\text{Index}(A[5][4]) = [5+4+3] + 1 = 13$$

$$\text{Index}(A[5][3]) = [5+4] + 2 = 11$$

$$\text{Index}(A[i][j]) = [n+n-1+n-2+\dots+n-(j-2)] + (i-1)$$

$$\left[\frac{n(n+1)}{2} - \frac{(n-j+1)(n-j+2)}{2} \right] + (i-1)$$

$$\left[\frac{n(n+1)}{2} - \frac{(n-(i-1))(n-(i-2))}{2} \right] + (i-1)$$

$$\left[\frac{n}{2} - \frac{n(j-2)}{2} - \frac{n(j-1)}{2} - \frac{(j-1)(j-2)}{2} \right] + (i-1)$$

$$\left[n(j-1) - \frac{(i-1)(j-2)}{2} \right] + (i-1)$$

110 #include <stdio.h>
 #include <stdlib.h>
 struct Matrix
 { int *A;
 int n;
 } y;
 void Set(struct Matrix *m, int i, int j, int x)
 { if (i >= j)
 m->A[m.n*(j-1)+(i-1)] = x
 }
 int Get(struct Matrix m, int i, int j)
 { if (i >= j)
 return m.A[m.n*(j-1)+(i-1)];
 else
 return 0; }
 void Display(struct Matrix m)
 { int i, j;
 for (i = 1; i < m.n; i++)
 { for (j = 1; j < m.n; j++)
 { if (i >= j)
 printf("%d", m.A[m.n*(j-1)+(i-1)-(j-2)*(i-1)-(i-1)*(i-1)]);
 else
 printf("0"); }
 printf("\n"); }
 }
 int main()
 { struct Matrix m;
 int i, j, x;
 printf("Enter dimensions "); }

scanf("%d", &m.n);

m.A = (int*) malloc(m.n * (m.n + 1) / 2 * sizeof(int));

printf("Enter all elements ");

for (i = 1; i <= m.n; i++)

{ for (j = 1; j <= m.n; j++)

{ scanf("%d", &x);

set(&m, i, j), x);

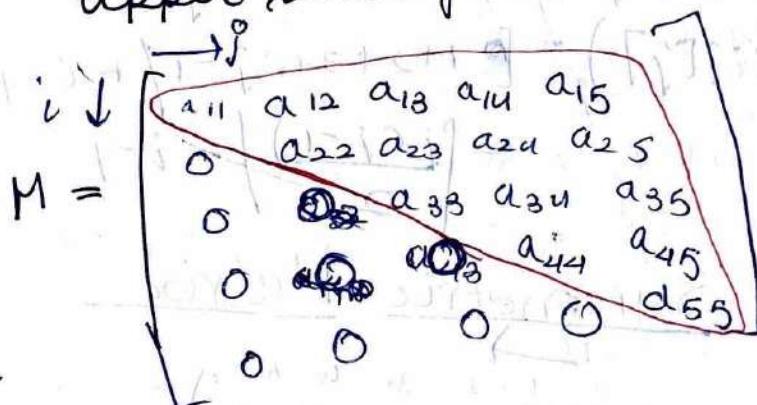
}

} printf("\n\n");

display(m);

return 0;

Upper Triangle Matrix



$$M[i][j] = 0 \quad \text{if } i > j$$

$$M[i][j] \neq 0 \quad \text{if } i \leq j.$$

$$\text{non-zero} = 5 + 4 + 3 + 2 + 1$$

$$= n + n - 1 + \dots + 1 = \frac{n(n+1)}{2}$$

$$\text{zero} = \frac{n(n-1)}{2}$$

(12)

Row-Majors :

a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	a_{21}	a_{22}	a_{23}	a_{24}	a_{25}	a_{31}	a_{32}	a_{33}	a_{34}	a_{35}	a_{41}	a_{42}	a_{43}	a_{44}	a_{45}	a_{51}
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	10	11	12	13	14	
row 1				row 2				row 3				row 4				row 5				

$$\text{Index}(A[4][5]) = [5+4+3] + 1 = 13$$

$$\begin{aligned} \text{Index}(A[i][j]) &= [n+n-1+\dots+n-(i-2)] + (j-i) \\ &= \left[\frac{(i-1)n - (i-2)(i-1)}{2} \right] + (j-i) \end{aligned}$$

Column-Majors :

a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	a_{21}	a_{22}	a_{23}	a_{24}	a_{25}	a_{31}	a_{32}	a_{33}	a_{34}	a_{35}	a_{41}	a_{42}	a_{43}	a_{44}	a_{45}
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	10	11	12	13	14
column 1				column 2				column 3				column 4				column 5			

$$\text{Index}(A[4][5]) = (1+2+3+4) + 3$$

$$\begin{aligned} \text{Index}(A[i][j]) &= [1+2+3+\dots+j-1] + i-1 \\ &= \left[\frac{(j-1)(j-1)}{2} \right] + i-1 \end{aligned}$$

Symmetric Matrix

i →				
↓ j				
1	2	3	4	5
M = 1	2	2	2	2
2	3	3	3	3
3	2	4	4	4
4	2	3	5	5
5	2	3	4	6

5×5

Since know lower triangular matrix we can create symmetric matrix.

$$\text{if } M[i][j] = M[j][i]$$

∴ we will store in 1D array by storing lower as upper part of symmetric matrix

Tri-Diagonal

$$M = \begin{bmatrix} & & i & & \\ i \downarrow & \begin{matrix} 1 & 2 & 3 & 4 & 5 \\ a_{11} & a_{12} & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 0 \\ 0 & a_{32} & a_{33} & a_{34} & 0 \\ 0 & 0 & a_{43} & a_{44} & a_{45} \\ 0 & 0 & 0 & a_{54} & a_{55} \end{matrix} & & \\ & & 5 \times 5 & \end{bmatrix}$$

Main diagonal $\approx i - j = 0$

Lower diagonal $\approx i - j = 1$

Upper diagonal $\approx i - j = 1$

$$\therefore |i-j| \leq 1$$

$M[i][j] = \text{non-zero if } |i-j| \leq 1$

$M[i][j] = 0 \text{ if } |i-j| > 1$

$$\text{Total elements: } 5 + 4 + 4 = n + n-1 + n-1 \\ = 3n - 2$$

a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	a_{21}	a_{22}	a_{23}	a_{24}	a_{25}	a_{31}	a_{32}	a_{33}	a_{34}	a_{35}
0	1	2	3	4	5	6	7	8	9	10	11	12		

Lower Diag. *Main Diag.* *Upper Diag.*

index $(A[i][j])$

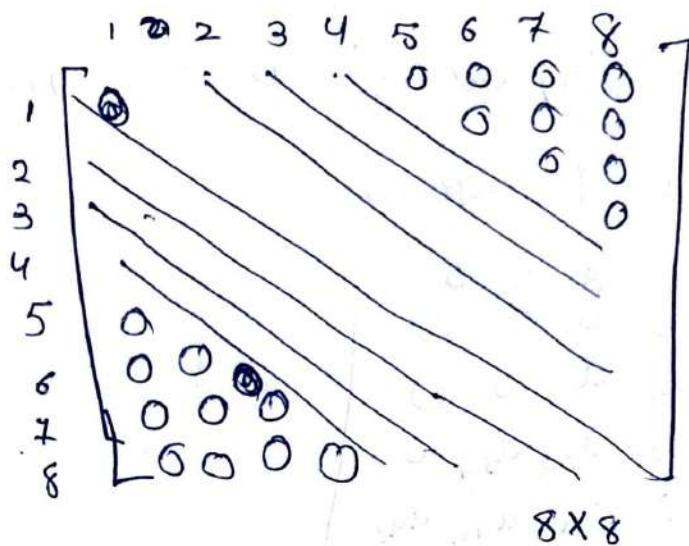
Case 1: if $i-j=1$

$$\text{index} = i-2$$

Case 2: if $i-j=0$ index $= n-1+i-1$

Case 3: if $i-j=-1$ index $= 2n-1+i-1$

119



Toepility Matrix:

	1	2	3	4	5
1	2	3	4	5	6
2	7	2	3	4	5
3	8	7	2	3	4
4	9	8	7	2	3
5	10	9	8	7	2

5×5

$j-i=2$
 $j-i=1$
 $j-i=0$

$i-j=3$
 $i-j=2$
 $i-j=1$

$$M[i][j] = M[i-(j)-1]$$

Q1 Need to store whole matrix only, 1st column

Q2 1st row of matrix can work

elements : $n+n-1$

A	2	3	4	5	6	7	8	9	10
	row					col			

Index $(A[i][j])$

case 1 if $i < j$ element belongs to off \leftrightarrow

index $= i - j$

case 2: if $i \geq j$

index $= n + i - j - 1$

Sparse Matrix

	1	2	3	4	5	6	7	8	9
1	0	0	0	0	0	0	3	0	0
2	0	0	8	0	0	10	0	0	0
3	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0
6	0	0	0	3	0	0	0	0	0
7	0	0	0	6	0	0	0	0	0
8	0	9	0	0	5	0	0	0	0

$$8 \times 9 = 72 \text{ elements}$$

Matrix in which more no. of zeros are present.
i.e. here 8 non zeros other values are zero
To store in less amount of memory
we will only store non zero value

Two representations (for storing in 1D array)

- 1) Coordinate List / 3-column Rep.
- 2) Compressed Sparse Row

* 3 column Rep

row <small>8 total</small>	columns <small>9 total</small>	elements	
		<small>8 (non zero)</small>	<small>8 (non zero)</small>
1	8		3
2	3		8
2	6		10
4	1		4
6	3		2
7	4		6
8	2		9
8	5		5

$$(6 + 8 + 8) \text{ elements} = \underline{\underline{24 \text{ elements}}}$$

Compressed Sparse Row

$$A = [3, 8, 10, 4, 2, 6, 9, 5]$$

array of nonzero elements

$$IA = [0, 1, 3, 3, 4, 4, 5, 6, 7, 8]$$

array sum of column elements in i^{th} row.

$$JA = [8, 3, 6, 1, 3, 4, 2, 5]$$

array of columns in which element are present

$8 + 9 + 8 = 25$ elements.

Addition of Square Matrices

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 0 & 0 & 0 & 6 & 0 & 0 \\ 0 & 7 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$A = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 5 & 1 & 2 & 3 & 3 & 5 \\ 4 & 2 & 2 & 4 & 1 & 1 \\ 6 & 7 & 2 & 5 & 4 & 4 \\ 5 & 6 & 7 & 1 & 5 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 5 & 0 \\ 0 & 0 & 2 & 0 & 0 & 7 \\ 0 & 6 & 0 & 9 & 0 & 0 \\ 8 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$B = \begin{bmatrix} 5 & 2 & 2 & 3 & 3 & 4 & 5 \\ 6 & 2 & 5 & 3 & 6 & 4 & 1 \\ 3 & 5 & 2 & 7 & 9 & 8 & 0 \end{bmatrix}$$

$$A+B=C = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 0 & 0 & 0 & 6 & 0 & 0 \\ 0 & 10 & 0 & 0 & 5 & 0 \\ 0 & 2 & 2 & 5 & 0 & 7 \\ 0 & 0 & 0 & 9 & 0 & 0 \\ 12 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$C = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 5 & 1 & 2 & 2 & 3 & 3 & 3 & 3 & 4 & 1 \\ 6 & 4 & 2 & 5 & 2 & 8 & 4 & 6 & 4 & 1 \\ 9 & 6 & 10 & 5 & 2 & 2 & 5 & 7 & 8 & 12 \end{bmatrix}$$

∴ here it is known: we know result
other wise we have to make $6+5 = 11$ element
∴ there can be almost 11 diff elements

Array Representations

↑ Let's code to Create Sparse Matrix.

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
struct Element
```

```
{ int i;  
  int j;
```

For row, column &
that element

```
  int x;
```

```
  { ("0") present  
  g;
```

```
struct Sparse
```

```
{ int m;
```

```
  int n;
```

```
  int num;
```

struct Element *ele; // for dynamically
allocate arrays of
size num

```
y;
```

```
void create(struct Sparse *S)
```

```
{ int i;
```

```
  printf("Enter dimensions");
```

```
  scanf("%d%d", &S->m, &S->n);
```

```
  printf("Number of non zero");
```

```
  scanf("%d", &S->num);
```

```
S->ele = (struct Element *) malloc
```

(S->num * size of (struct Element));

```
for(i=0; i< S->num; i++)
```

```
  printf("Enter non zero elements");
```

```
  scanf("%d%d%d", &S->ele[i].i,
```

(S->ele[i].j), &S->ele[i].x);

```
i++;
```

j++;

118

void display (struct Sparse)

```

    {
        int i, j, k = 0;
        for (i = 0; i < s.m; i++)
            for (j = 0; j < s.n; j++)
                if (i == s.ele[k].i & & j == s.ele[k].j)
                    printf ("%d", s.ele[k].x);
                else
                    printf ("0");
        printf ("\n");
    }

```

(a) $i \in [0, m]$
j $\in [0, n]$

struct Sparse *add (struct Sparse *S1, struct Sparse *S2)

(a) struct Sparse *sum;

int i, j, k;

{ "now i=j=k=0"; } $i \in [0, m]$ $j \in [0, n]$ $k \in [0, m+n]$

{ "now i=k+j=n"; } $i \in [0, m]$ $j \in [0, n]$ $k \in [0, m+n]$

return (new Node *) malloc (sizeof (struct Sparse));

sum = (struct Sparse *) malloc (sizeof (struct Sparse));

sum->ele = (struct Element *) malloc (sizeof (struct Element));

sum->ele = (struct Element *) malloc (sizeof (struct Element));

sum->ele = (struct Element *) malloc (sizeof (struct Element));

sum->ele = (struct Element *) malloc (sizeof (struct Element));

sum->ele = (struct Element *) malloc (sizeof (struct Element));

while (i < s1->num & & j < s2->num)

& if (s1->ele[i].j < s2->ele[j].i)

sum->ele[k++].j = s1->ele[i++].j;

else if ($s_1 \rightarrow \text{ele}[i].i > s_2 \rightarrow \text{ele}[j].i$)

sum $\rightarrow \text{ele}[k+1] = s_2 \rightarrow \text{ele}[j+1];$

else

if ($s_1 \rightarrow \text{ele}[i].j < s_2 \rightarrow \text{ele}[j].j$)

sum $\rightarrow \text{ele}[k+1] = s_1 \rightarrow \text{ele}[i+1];$

else if ($s_1 \rightarrow \text{ele}[i].i > s_2 \rightarrow \text{ele}[j].i$)

sum $\rightarrow \text{ele}[k+1] = s_2 \rightarrow \text{ele}[j+1];$

else

else

sum $\rightarrow \text{ele}[k] = s_1 \rightarrow \text{ele}[i];$

sum $\rightarrow \text{ele}[k+1].x = s_1 \rightarrow \text{ele}[i+1].x + s_2 \rightarrow \text{ele}[j+1].x;$

sum $\rightarrow \text{ele}[k+1] = s_1 \rightarrow \text{ele}[i];$

for (; $i < s_1 \rightarrow \text{num}; i++$) sum $\rightarrow \text{ele}[k+1] = s_1 \rightarrow \text{ele}[i];$

for (; $j < s_2 \rightarrow \text{num}; j++$) sum $\rightarrow \text{ele}[k+1] = s_2 \rightarrow \text{ele}[j];$

sum $\rightarrow m = s_1 \rightarrow m;$

sum $\rightarrow n = s_1 \rightarrow n;$

sum $\rightarrow num = k;$

return sum;

int main()

struct Sparse $s_1, s_2, *s_3;$

create(&s1)

create(&s2);

s3 = add(&s1, &s2);

print("First Matrix (n)");

display(s1);

print("Second Matrix (n)");

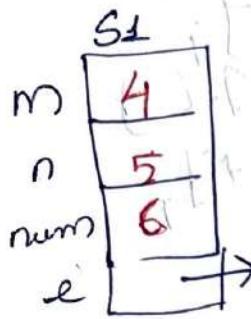
display(s2);

print("Sum Matrix (n)");

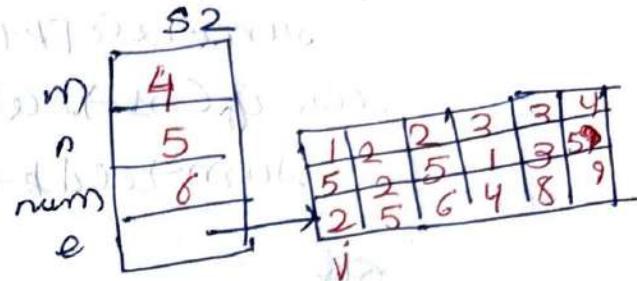
display(*s3); return 0; }

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

0	0	0	0	2
0	5	0	0	6
4	0	8	0	0
0	0	0	0	9



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



add (struct Sparse *S1, struct Sparse *S2)

{ struct Sparse *sum; // we are really
// pointers so that
// if (S1+m) = S2+n, |S1+m| != S2+n
// can return
// returning j // checking whether
// dimensions are
// equal or not.

sum = new struct Sparse;

sum->m = S1+m; sum->n = S2+n;

sum->e = new Element [S1+num+S2+num];

// adding elements.

while (i < S1+num & j < S2+num)

{ if (S1+e[i].i < S2+e[j].i)

 sum->e[k++].i = S1+e[i].i;

 else if (S1+e[i].i > S2+e[j].i)

 sum->e[k++].i = S2+e[j].i;

i & both
S1 & S2 are
same

else

{ if (S1+e[i].i < S2+e[j].i) sum->e[k++].i = S1+e[i].i;

 else if (S1+e[i].i > S2+e[j].i)

 sum->e[k++].i = S2+e[j].i;

Both i & j
are same
for both
S1 and S2

else sum->e[e] = S1+e[i].i;

{ sum->e[k++].i = S2+e[j].i; + S1+e[i].i.x }

}

y.

Steps:

- First rows are checked in S1 & S2
- If rows are equal columns are checked
 - The one which is lesser is then stored in sum $\rightarrow e[k,j]$ and if columns are equal both elements are added & then stored in sum $\rightarrow e[k+1,j]$.

Polynomial Representation

univariate polynomial

$$p(x) = 3x^5 + 2x^4 + 5x^2 + 2x + 7$$

1) Polynomial Representations

2) Evaluation of Polynomial

3) Addition of Two polynomials

$$p(x) = \cancel{3x^5} + \underline{2x^4} + \underline{5x^2} + \cancel{2x} + 7$$

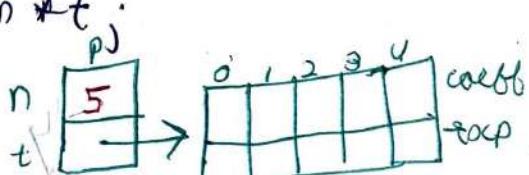
coeff

struct Terms struct Poly

2 int coeff
int exp;

{ ; }

1 int n
struct Term *t;



int main()

struct Poly p

printf("No of non-zero terms")

scanf("%d", &p.n)

p.t = new Term [p.n];

printf("Enter polynomial terms");

for (i=0; i<p.n; i++)

{ printf("Term no %d", i+1);

scanf("d %d", &p.t[i].coeff, &p.t[i].exp);

122

Polynomial Evaluation

struct Poly P

$x = 5; \text{sum} = 0;$

for ($i = 0; i < P.n; i++$)

{
 $\text{sum} += P.t[i].coeff * \text{pow}(x, P.t[i].exp);$

}

\downarrow

return sum;

Polynomial Addition

$i = 0; j = 0; k = 0$

while ($i < P1.n \& j < P2.n$)

{
 if ($P1.t[i].exp > P2.t[j].exp$)

$P3.t[k++].exp = P1.t[i].exp;$

else if ($P2.t[j].exp > P1.t[i].exp$)

$P3.t[k++].exp = P2.t[j].exp;$

else

$P3.t[k].exp = P1.t[i].exp$

$P3.t[k].coeff = P1.t[i].coeff$

$P3.t[k].coeff += P2.t[j].coeff;$

$P1$	
3	

0	1	2
5	2	5
4	2	0

coeff

exp

$P2$	
5	

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

coeff

exp

$j \uparrow$

Link List

Why Linked List

1) Problems with Arrays.

2) Difference B/w Array & Linked List

Problems with Array:

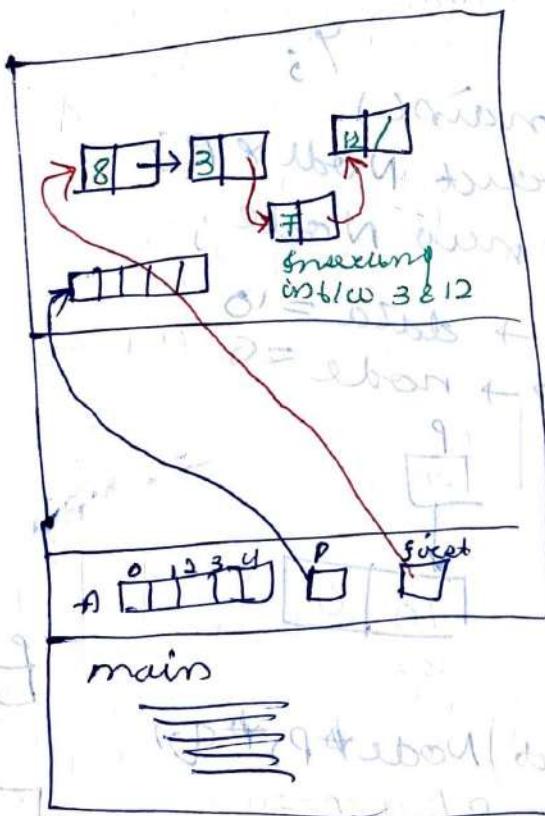
→ Array is of fixed size want
 ∴ at Run time if we need variable size of array
 we cannot do

Difference B/w Array & Link List

We want data structure which can change
 size at the run time

∴ Link List does it

(More flexible than array)

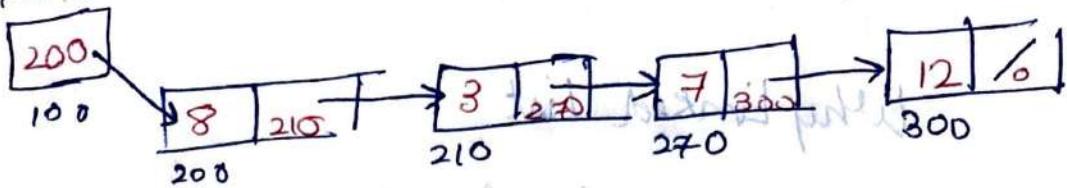


Array is like
 continuous bunches
 of fixed size

Linked List is stored in Heap Memory.

About Linked List

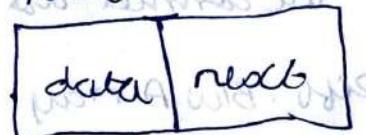
First



1) What is Link List?

Linked List is a collection of ~~common~~ nodes where each nodes contains data and pointer to next node

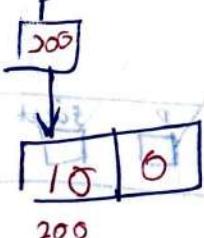
How to make Linked List Node



points to the same structure
pointer to the next node

(THIS IS CALLED)
SELF REFERRENTIAL
POINTER

```
int main()
{
    struct Node *p;
    p = new Node();
    p->data = 10;
    p->node = NULL; // Create null
```

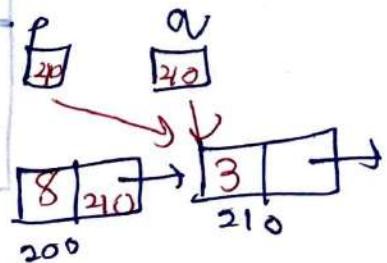


struct Node *p, *q;

1 q = p // q = p = 200

2 q = p->next // q = 210

3 p = p->next; // p = 210



```

struct Node* p=NULL;
if(p==NULL)
if(p==0) } To check if it is null pointer
if(!p) "p=0" & any non zero value will
        be considered as true.

if(p!=NULL) } To check that pointer is pointing
if(p!=0)   towards the node
if(p)

```

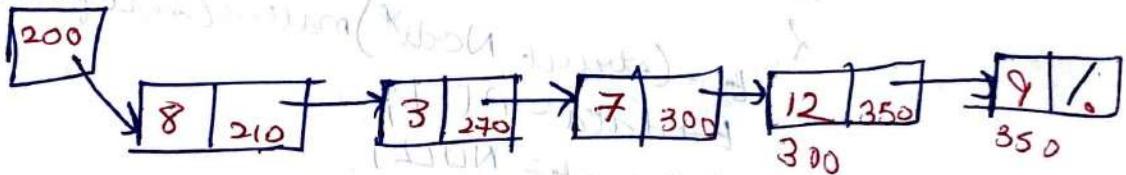
To check whether it's last Node or not

```
if(p->next==NULL)
```

If true then last Node

Display Linked List

first



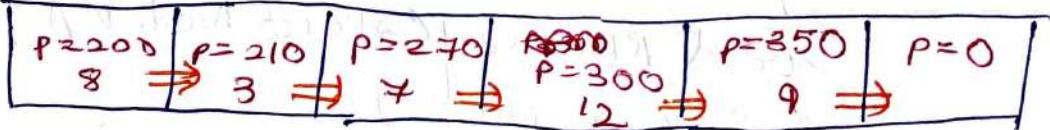
display(struct Node *p);

```

{
    while(p!=NULL)
    {
        printf("id ", p->data);
        p=p->next;
    }
}

```

display(first);



Code to Create & Display Linked List.

```

#include <stdio.h>
#include <stdlib.h>

struct Node
{
    int data;
    struct Node *next;
};

void *first = NULL;
void create(int A[], int n)
{
    int i;
    struct Node *t, *last;
    first = (struct Node *) malloc(sizeof(struct Node));
    first->data = A[0];
    first->next = NULL;
    last = first;
    for (i = 1; i < n; i++)
    {
        t = (struct Node *) malloc(sizeof(struct Node));
        t->data = A[i];
        t->next = NULL;
        last->next = t;
        last = t;
    }
}

void display(struct Node *P)
{
    while (P != NULL)
    {
        printf("%d", P->data);
        P = P->next;
    }
}

void RDdisplay(struct Node *P)
{
    if (P == NULL)
    {
        RDdisplay(P->next);
        printf("%d", P->data);
    }
}

```

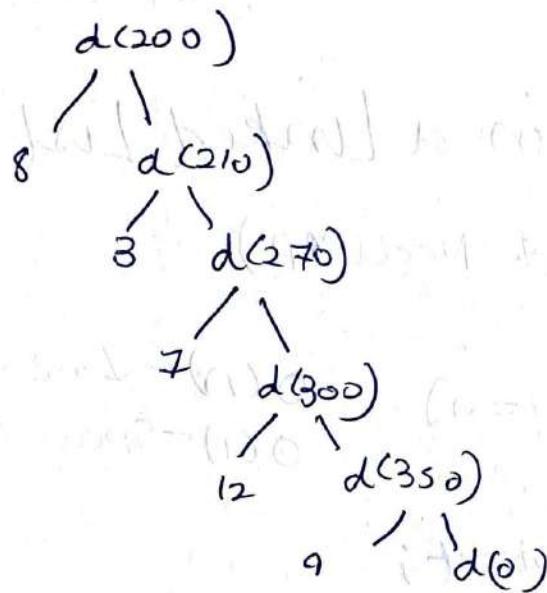
```

int main()
{
    struct Node *temp;
    int A[8] = { 3, 5, 7, 10, 25, 8, 32, 2 };
    create(A, 8);
    display(first);
    return 0;
}

```

↓

Recursive Display of Linked List.

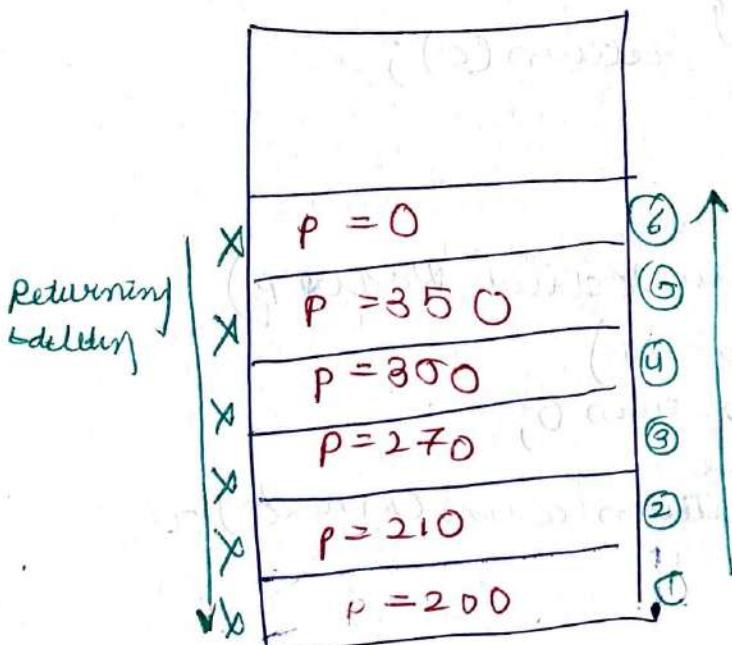


```

void display(struct node *p)
{
    if (p != NULL)
    {
        printf("%d", p->data);
        display(p->next);
    }
}

```

display(first)



Stack

 $[O(n): n]$

Space complexity : 6 activation records in stack
More memory consuming than loop.

Displaying & Reversing.

```
void Rdisplay (struct Node *p)
```

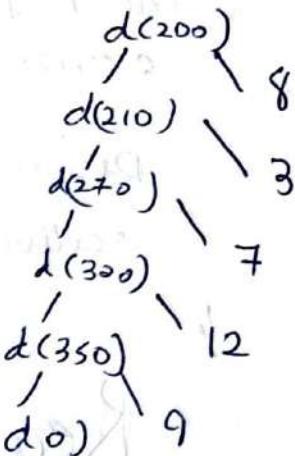
```
{ if (p!=NULL)
```

```
    Rdisplay (p->next);
```

```
    printf (" %d ", p->data);
```

```
}
```

Rdisplay first



∴ output: 8 12 7 3 9

Counting Nodes in a Linked List.

```
int count (struct Node *p)
```

```
{ int c=0;
```

```
while (p!=0)
```

```
    c++;
```

```
    p=p->next;
```

```
}
```

return (c);

$O(n)$ - Time Complexity

$O(1)$ - Space Complexity

Recursive

```
int count (struct Node *p)
```

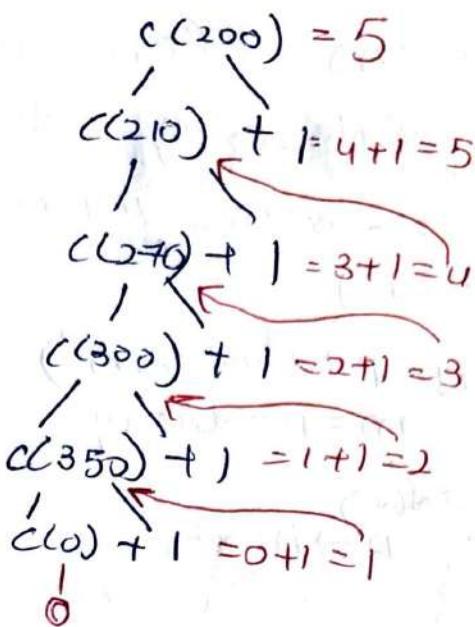
```
{ if (p==0)
```

```
    return 0;
```

```
else
```

```
    return count (p->next) + 1;
```

```
}
```



$\therefore n+1$ calls

$O(n)$

Time complexity

Sums of All Elements.

Add (Struct)

```
int Add (struct Node *P)
```

```
{
    int sum=0;
    while(P)
    {
        sum=sum+p->data;
        p=p->next;
    }
}
```

Time complexity: $O(n)$

using recursion

```
int Add (struct Node *P)
```

```
{
    if (P==0)
        return 0
}
```

```
else
    return Add (P->next)+P->data;
```

(130)

Maximum Element.

```

int max (Node *p)
{
    int m = -32768 // MIN-INT
    while (p)
    {
        if (p->data > m)
            m = p->data;
        p = p->next;
    }
    return m;
}

```

Recursive

```
int max (Node *p)
```

```

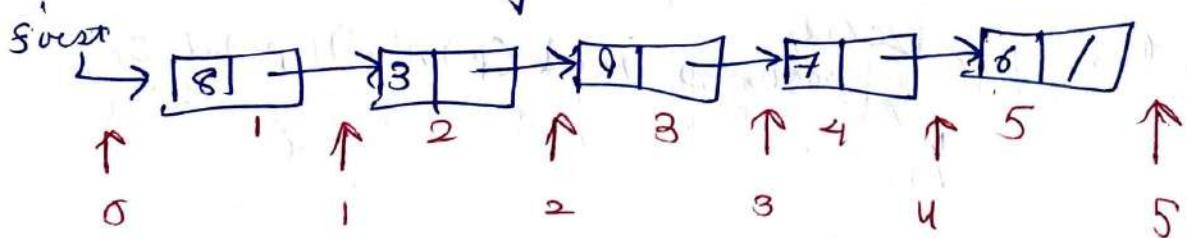
int x=0
if (p==0)
    return MIN-INT;
else
    if (x>p->data)
        x = max (p->next);
    else
        return p->data;
}

```

int $x = 0$;
 $x = \max(p \rightarrow \text{next})$,
 $x = \max(p \rightarrow \text{data})$,
 $x > p \rightarrow \text{data}$,
 $x = p \rightarrow \text{data}$;

Pg 13 2

Inserting in a Linked List.



Searching in a Linked List

```

Node* search(Node* P, int key)
{
    while (P != NULL)
    {
        if (key == P->data)
            return (P);
        P = P->next;
    }
    return NULL;
}

```

Recursion

```

Node* search(Node* P, int key)
{
    if (P == NULL)
        return NULL;
    if (key == P->data)
        return (P);
    return search(P->next, key);
}

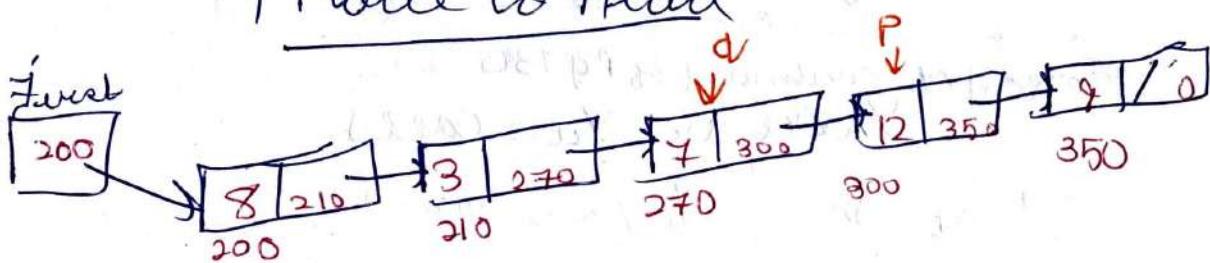
```

Improve Searching (already learned in arrays)

1) Transposition

2) Move to Head.

→ Move to Head



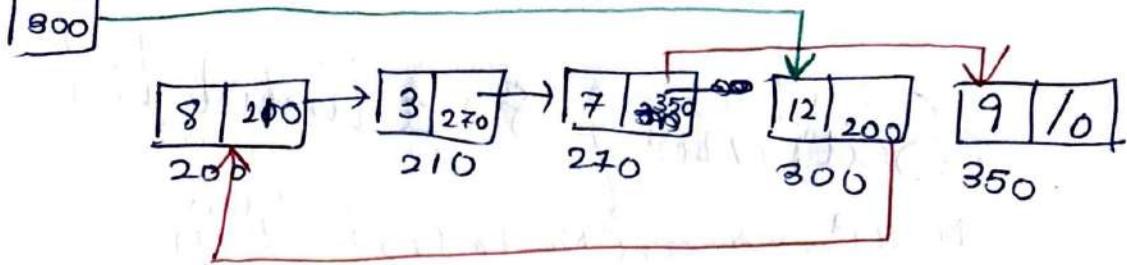
$$\text{key} = 12$$

∴ we have to Bring 12 to Head.

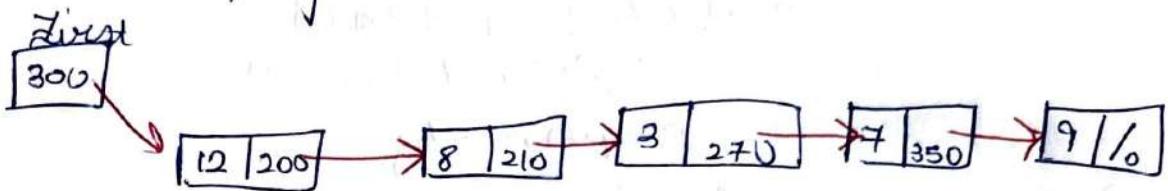
as following p when $P \rightarrow \text{data} \neq \text{key}$

First

After operation:



Rearranging it.



struct Node *LSearch (struct Node *p, int key)

```

    {
        struct Node *q = NULL;
        while (p != NULL)
            {
                if (key == p->data)
                    {
                        if (p == first) // If key is first then
                            q->next = p->next; // q will be NULL
                        p->next = first; // No need to do
                        first = p; // at next
                        return p;
                    }
                q = p;
                p = p->next;
            }
        return NULL;
    }

```

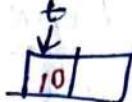
Program continued on Pg 130

There are Two cases

- 1 Insert before First.
- 2 Insertion after given Position

I Insert before First

159

- Step : creation new
- (i) Initialize a node . 
 - (ii) Enter data in node
 - (iii) Make it point on First



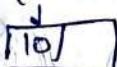
(iv) First should be pointed to t now.

\therefore 4 steps are required : time complexity. O(1)

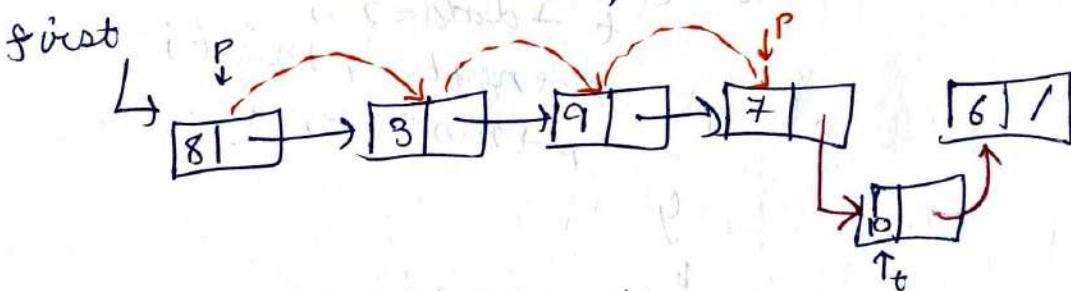
Node *t = new Node;
 $t \rightarrow \text{data} = x;$
 $t \rightarrow \text{next} = \text{first};$
 $\text{first} = t;$

II Insert Node at a given Position

Ex: pos = 4



Take a pointer p to take it to the 4th position
of link list ,



Node *t = new Node;

$t \rightarrow \text{data} = x$

$p = \text{first}$

for (i=1; i<=pos-1 ; i++)

{ $p = p \rightarrow \text{next};$

$t \rightarrow \text{next} = p \rightarrow \text{next}$

$p \rightarrow \text{next} = t;$

Time complexity : $O(n)$ (worst case)
 $O(1)$ (Best Case)

1384

Pointers which are pointing on other Nodes
are called Links

Now we will combine both cases.

void Insert(int pos, int x)

```
{ Node *t, *p;
if (pos <= 0)
    t = new Node();
t->data = x;
t->next = first;
first = t;
```

else if (pos > 0)

```
{ p = first;
for (i=0; i < pos - 1 && p) {
    p = p->next;
```

if (p) // checking p is pointing to a node or not

```
{ t = new Node();
t->data = x;
t->next = p->next;
p->next = t;
```

g

p

}

if (p) { p->next = t; }

else { t->next = first; }

Creating List by Inserting at Last.

```
void insertLast(int x)
```

```
{ Node *t = new node;
  t->data = x;
  t->next = NULL;
  if (first == NULL)
```

```
{ first = last = t
```

```
else
```

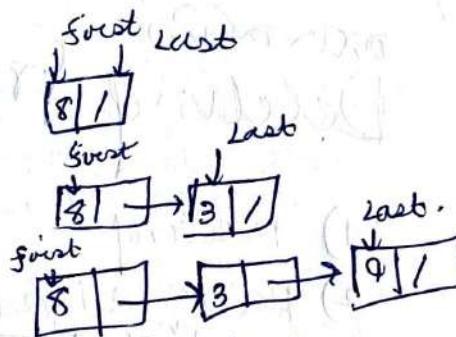
```
{ last->next = t;
  last = t
```

y

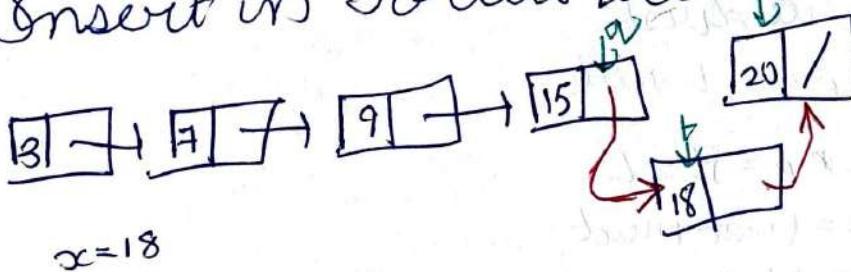
insertLast(8)

insertLast(3)

insertLast(9)



\Rightarrow Insert in Sorted List



Code.

```
void SortedInsert(struct Node *p, int x)
```

```
{ struct Node *t, *q = NULL;
  t = (struct Node *) malloc(sizeof(struct Node));
  t->data = x;
  t->next = NULL;
  if (first == NULL)
    first = t;
  else
```

Resolving
special
case

```

    while ( $p \neq p \rightarrow \text{next} < x$ )
    {
        q = p;
        p = p → next;
    }
    if ( $p = \text{first}$ )
    {
        t → next = first;
        first = t;
    }
    else
    {
        t → next = q → next;
        q → next = t;
    }
}

```

Operations from Linked List -

Deleting from Linked List -

1) Deleting First Node.

$\text{first} \downarrow$
2) Delete a Node at given Position



Delete First -

one pointer to delete

Node $*p = \text{first}$

$\text{first} = \text{first} \rightarrow \text{next}$

$x \# = p \rightarrow \text{data}$

delete $P;$

Time complexity - O(1)

$PSG = 4$
(first , $\text{first} \rightarrow \text{next}$, $x \#$ which is to be deleted)
One pointers at positions of which is to be deleted
and pointers before $x \#$ position

127

Node *P = first
Node *q = NULL
for (i=0; i<pos-1; i++)

{
 q = P;
 P = P->next;

}
 q->next = P->next;
 x = p->data

 delete P; *worst case*

Time complexity : O(n)

Making A Single Function

int Delete(int pos)

{ Node *P, *q;

 int x = -1; i;
 if (pos == 1)

{
 x = first->data;
 P = first;
 first = first->next;
 delete P;

 else *remove O(n)*

{ P = first;

 q = NULL;

 for (i=0; i<pos-1; i++) & P; i++)

{ q = P;

 P = P->next;

} if (P)

{ q->next = P->next;

 x = P->data;

 delete P;

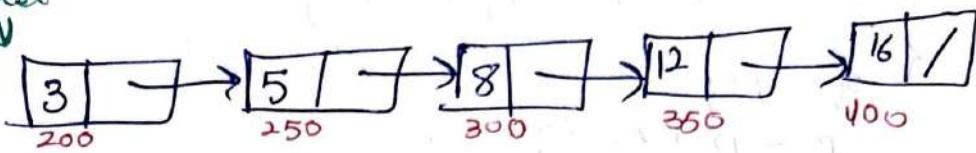
}

Returns x

y

135

Check if Linked List is Sorted

first
↓

$$\alpha = -32768$$

3 ~~200~~ $p = 200$
 5 $p = 250$
 8 $p = 300$
 12 $p = 350$
 16 ~~400~~ $\text{END } p = 400$
 $p = \text{NULL} \therefore \text{program } \underline{\text{OVER}}$

```
int alpha = -32768
```

```
Node *p = first;
```

while ($p \neq \text{NULL}$)

{ if ($p \rightarrow \text{data} < \alpha$)

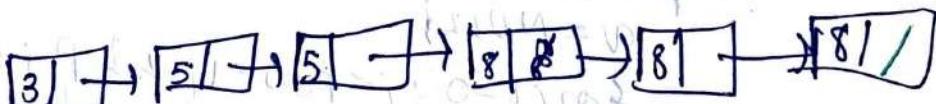
 return false;

$\alpha = p \rightarrow \text{data};$

$p = p \rightarrow \text{next};$

}
return true.

Remove Duplicates from linked list.



```
Node *p = first;
```

```
Node *q = first + next;
```

while ($q \neq \text{NULL}$)

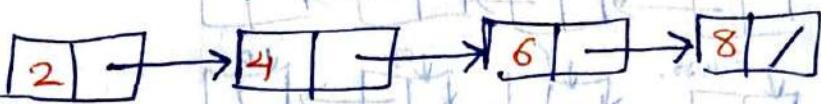
{ if ($p \rightarrow \text{data} \neq q \rightarrow \text{data}$)

$p = q;$
 $q = q + \text{next};$

 use

$p \rightarrow \text{next} = q \rightarrow \text{next};$
 delete q_j ; $q = p \rightarrow \text{next};$

Reversing a Linked List

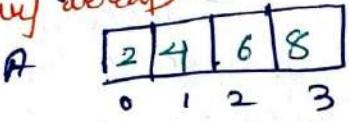


1 Reversing element

2 Reversing Links

use proper Reversing Links in Linked List

⇒ Reversing Element
(Auxiliary array)



copying elements of linked list in array

• 4 Then ~~loop~~ put the value in linked list from $i=3$ to $i=0$

Assuming A array is created & elements are copied in that auxiliary array

$p = \text{first};$

$i = 0;$

while ($p \neq \text{NULL}$)

$\quad A[i] = p \rightarrow \text{data};$
 $\quad p = p \rightarrow \text{next};$

$i + 1$

n

Total complexity: $O(n)$

$p = \text{first}; i = -1$

while ($p \neq \text{NULL}$)

$\quad p \rightarrow \text{data} = A[i - 1];$
 $\quad p = p \rightarrow \text{next};$

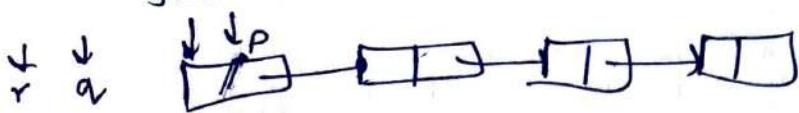
$\frac{n}{2n}$

100

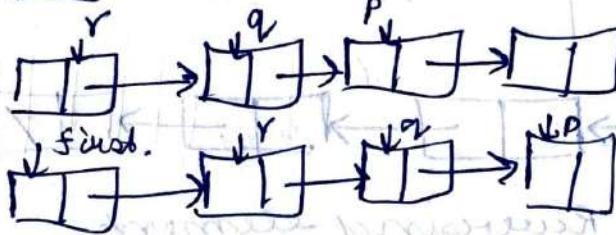
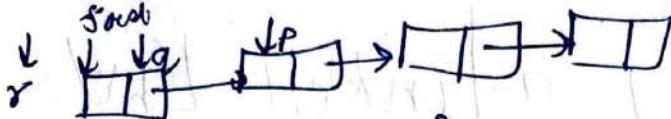
→ By Reversing Links

we will use sliding points

first



first



$p = \text{first}$

$q = \text{NULL}$

$r = \text{NULL}$

while ($P \neq \text{NULL}$)

$r = q;$

$q = p;$

$p = p + \text{next}$

$q \rightarrow \text{next} = r;$

p for
storing
base value
node
address

q for
changing
links to
location

r storing
address
of previous
value

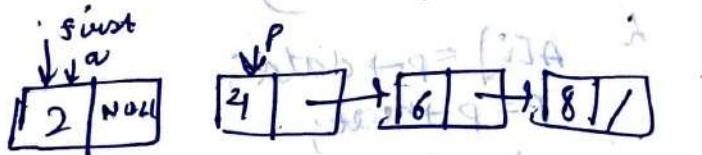
$\text{first} = q;$

Tracing Algorithm

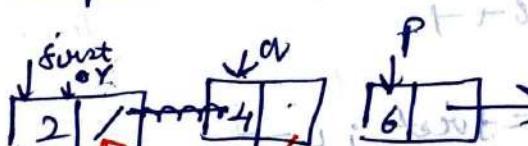


→ Initialization

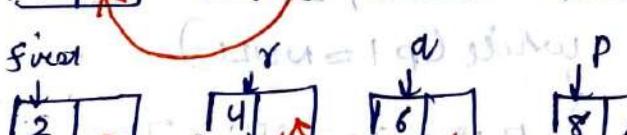
\downarrow
 \downarrow
 NULL NULL



→ Loop 1st run



→ Loop 2nd run



→ Loop 3rd run



→ Loop 4th run

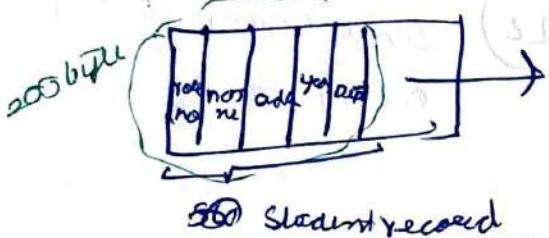
$\text{first} = q;$

→ Loop ends

Reversing link is most favourable.

We only need three pointer slides which will consume less time. ∵ Reversing element will take much space for copying all the data in Array then Reversing.

If in case we have nodes such as:



Now to create array of structure of each 200 byte
large amount of memory will be used.
But pointers will use same amount of memory
irrespective of size of data in node

Recursive Reverse
↓ First

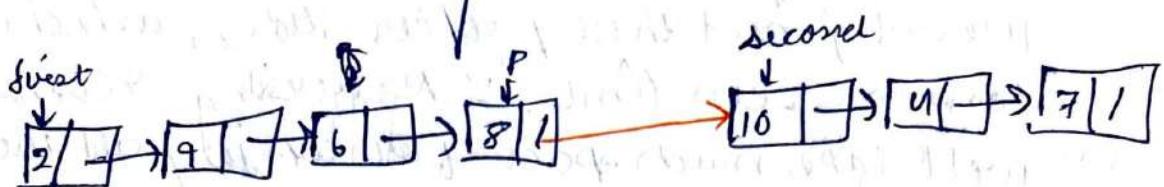


```
* void Reverse(Node *q, Node *p)
{
    if (p != NULL)
        Reverse(p, p->next);
    p->next = q;
}
```

```
else
{
    first = q;
    if (first->next = first)
        first->next = p;
    else
        first->next = p;
}
```

142

Concatenating 2 Linked List.



$p = \text{first};$
 $\text{while } (p \neq \text{NULL})$ time complexity

{ $p = p \rightarrow \text{next}$

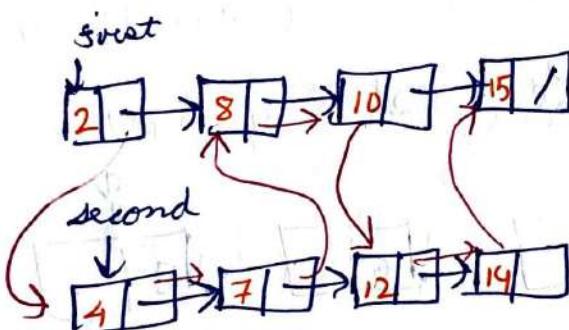
}

$p \rightarrow \text{next} = \text{second};$

$\text{second} = \text{NULL};$

$O(n)$

Merging 2 List (sorted)



if ($\text{first} \rightarrow \text{data} < \text{second} \rightarrow \text{data}$)

{ $\text{third} = \text{last} = \text{first};$
 $\text{first} = \text{first} + \text{next};$
 $\text{last} + \text{next} = \text{NULL};$

}
else

{ $\text{third} = \text{last} = \text{second};$

$\text{second} = \text{second} + \text{next};$

$\text{last} + \text{next} = \text{NULL};$

}

while (first != NULL & second != NULL)

{ if (first->data < second->data)

{ last->next = first ;
 last = first ;
 first = first->next ;
 last->next = NULL ; }

}

else

{ last->next = second ;
 last = second ;
 second = second->next ;
 last->next = NULL ; }

}

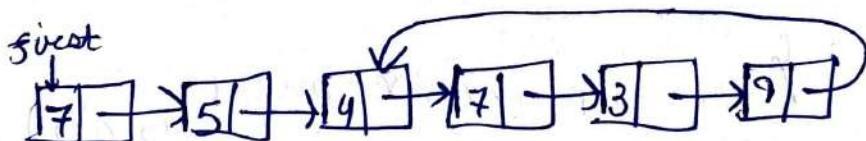
if (first == NULL)

last->next = first ;

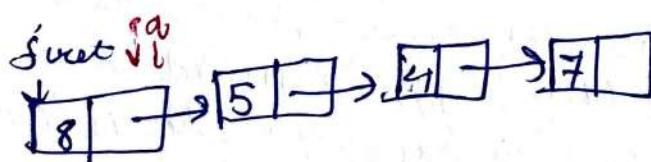
else

last->next = second ;

Check for Loop in Linked List .



Loop



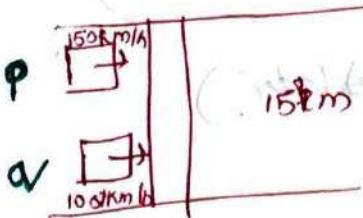
Linear

Best Method : p will be incremented by one position

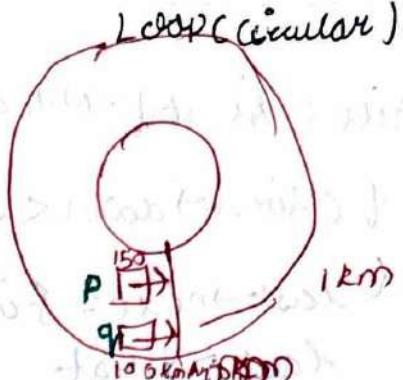
q will be incremented by 2 positions

144

Linear



p & q will never meet



p & q will meet after some time,

Code:

```

int isLoop(struct Node *s)
{
    struct Node *P, *Q;
    P = Q = s;
    do
        if (P->next == Q->next) // meeting by 2 places.
            return 1; // is Q != NULL
        P = P->next;
        Q = Q->next;
    } while (P != Q && P != NULL);
    if (P == Q) // P is NULL or
        return 1; // occurs 1; because not equal
    else
        return 0;
}

```

2 pointers required
Time complexity $O(n)$

Other Method (not good)

(i) Store Address of every Nodes while travelling through a linked list if some address is repeated then loop

(ii) If elements are unique in Linked List then if value repeats then Loop,

Floyd Theorem, For circular linked list,

144 @

If we want to know from where circular loop is starting this go:

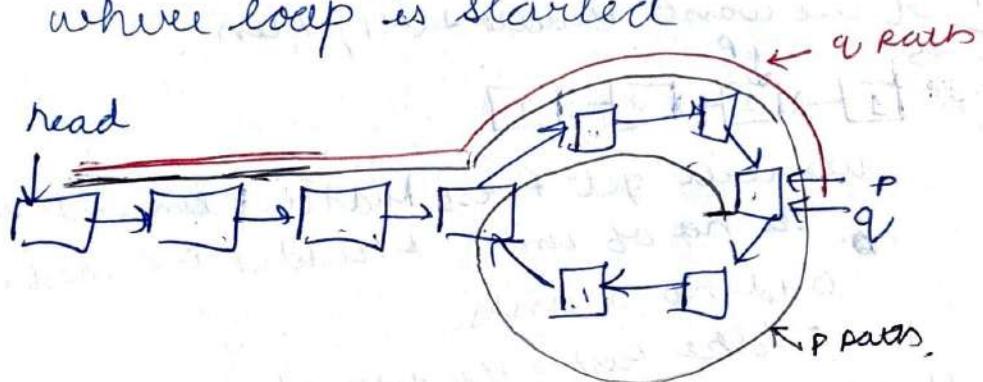
first:

p ka 2 baar move
 q ka 1 baar move

Then, $p \neq q$ yadi mile tab circular loop h linked list me

not isaha bhi $p = q$ mile
Now third points & will start from head. Now p & q will move to next every time & p also

Locations where p & q meets is where loop is started

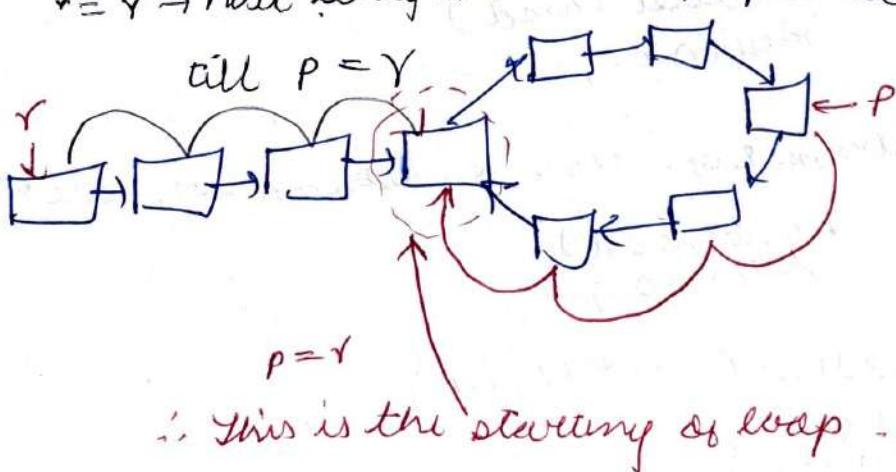


Now

$r = \text{head}$

$r = r \rightarrow \text{next every time}$

$p = p \rightarrow \text{next}$



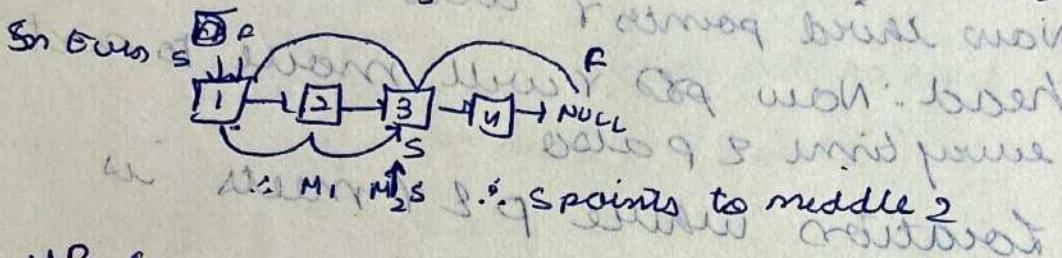
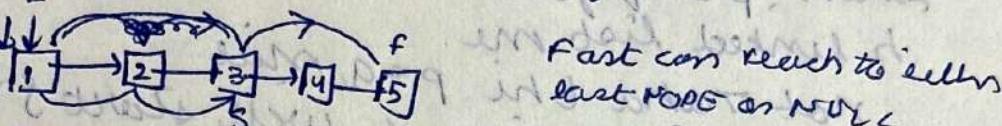
144 ⑥

→ Find middle element in linked list.

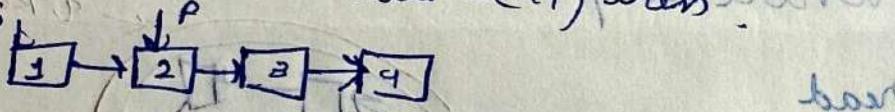
$p = \text{head}$ // Fast moving pointer
 $q = \text{head}$ // Slow moving pointer

while ($p \neq \text{NULL}$ & $p \rightarrow \text{next} \neq \text{NULL}$) VIMP
 { $p = (p + \text{next}) \rightarrow \text{next};$ \Rightarrow call me 6hi
 $q = q + \text{next}$ $p + \text{next} \neq \text{NULL} \& p \neq \text{NULL}$
 $}$ $\underline{\text{not issma}}$
 $\text{mid} = q$ $\text{mid is pointer to Node.}$

Midpoint Removal Technique



IMP If we want middle = (n/2), then we do



If we will get first middle element if even
 & no of links & middle element in
 odd no of links.

= Take fast = head + next.

⇒ LL operator overloading

istream & operator >> (istream & is, node * & head)
 { buildlist (head);
 return is; }

ostream & operator << (ostream & os, node * & head)
 { print (head);
 return os; }

void buildlist (node * & head)
 { int data;
 cin >> data;
 while (data != -1) { insertatend (head, data);
 cin >> data; }

[LL Merging 2 Sorted LL]

→ Recursive Approach

```

node * merge(node * a, node * b)
{
    if (a == NULL)
        return b;
    else if (b == NULL)
        return a;
    else
        node * c; // chose value to point to next case.
        if (a->data < b->data)
        {
            c = a;
            c->next = merge(a->next, b);
        }
        else
            c = b;
            c->next = merge(a, b->next);
        return c;
}

```

Merge Sort

```

node * mergesort(node * head)
{
    // Base case
    if (head == NULL || head->next == NULL) {
        return head;
    }
    // Rec Case
    // 1. Mid Point
    node * mid = midPoint(head);
    node * a = head;
    node * b = mid->next;

```

~~midPoint~~ midnext = null;

// [2. Recursively sort]

a = mergeSort(a);
b = mergeSort(b);

→ pointing how mergeSort
is working

// [3. Merge a & b]

node * c = merge(a, b);
return c;

Time complexity

Array

midPoint $O(1) = k$

Recurse case $2T\left(\frac{n}{2}\right)$

Merge $O(n)$

linked list

$O(n)$

$2T\left(\frac{n}{2}\right)$

$O(n)$

$$T(n) = k + 2T\left(\frac{n}{2}\right) + kn$$

↓

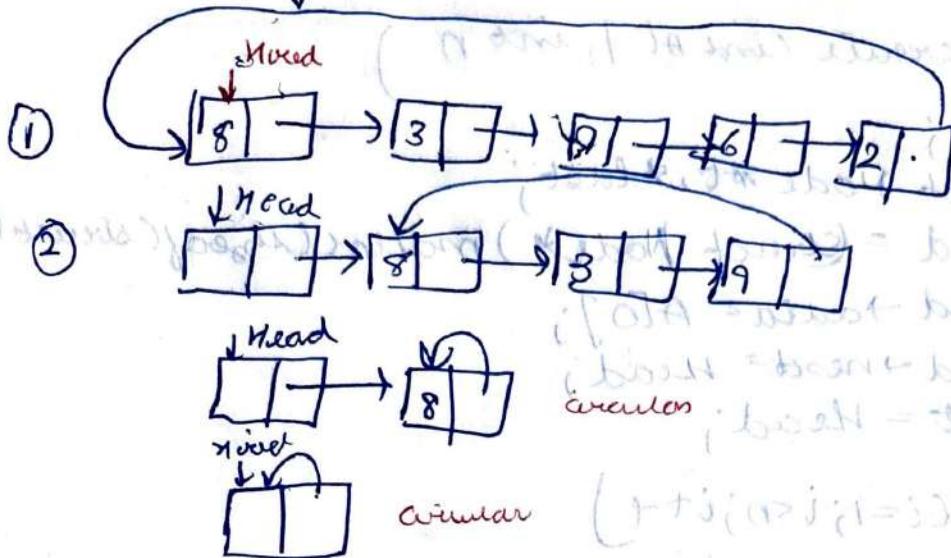
$O(n \log n)$

$$kn + 2T\left(\frac{n}{2}\right)$$

↓

$O(n \log n)$

Singly Circular Linked List



Q: We want to take null Node as a Circular
Ans we use ~~loop~~ representation ②

Display

```
void Display (Node * p)
```

^

dd

printing ("1. d", P->data)

P = P->next;

while (P != Head)

Display (Head)

```
void Display (Node ** pp)
```

^

static int flag = 0; // so that first time

it passes in if condition

if (P != Head), flag = 1

when p comes on Head for second time it stops

flag = 1;

printing ("1. d", P->data)

Display (P->next)

flag = 0;

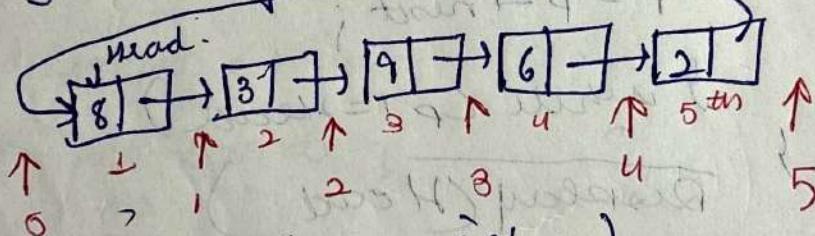
→ Create function

```
void create (int A[], int n)
```

```
{ int i;
struct Node *t, *last;
Head = (struct Node *) malloc(sizeof(struct Node));
Head->data = A[0];
Head->next = Head;
last = Head;
```

```
for (i=1; i<n; i++)
{
    t = (struct Node *) malloc(sizeof(struct Node));
    t->data = A[i];
    t->next = last->next;
    last->next = t;
    last = t;
}
```

b) Inserting in a Circular Linked List.



```
void insert (int pos, int x)
```

```
{ Node *t, (*P);
```

~~int *i;~~

```
if (pos == 0)
{
    t = (new Node);
    t->data = x;
    if (Head == NULL)
    {
        Head = t;
        Head->next = Head;
    }
}
```

else

{ p = Head;

while ($p \rightarrow \text{next} \neq \text{Head}$) $p = p \rightarrow \text{next};$

$p \rightarrow \text{next} = t; t \rightarrow \text{next} = \text{Head}; \text{Head} = t$

}

else

{ if $\text{desert} + q = q$ ($\text{desert} = ! \text{desert} + q$)

for (i = 0; i < p->s - 1; i++) $t = \text{next} = \text{next} \rightarrow \text{next}$

$p = p \rightarrow \text{next};$

$t = \text{new Node}$

$t \rightarrow \text{data} = x;$

$t \rightarrow \text{next} = p \rightarrow \text{next};$

$p \rightarrow \text{next} = t;$

}

}

Length of Circular List

int Length (struct Node *p)

{ int len = 0;

do { $t = p \rightarrow \text{next};$ if ($t = \text{next}$) $t = ! t$; } while ($t = p$)

$len++;$ $t = p \rightarrow \text{next};$ if ($t = p$) $t = ! t$;

while ($t \neq p$) $t = p \rightarrow \text{next};$ if ($t = p$) $t = ! t$;

return len;

}

}

(if $t = p$)

148

Delete Function

int Delete(struct Node *p, int index)

{ struct Node *q;

int i, x;

if (index < 0 || index > length(Head))

return -1;

if (index == 1)

{ while (p->next != Head) p = p->next;

x = Head->data;

if (Head == p)

{ free(Head);

Head = NULL;

}

else

{ p->next = Head->next;

free(Head);

Head = p->next;

}

else

{ for (i=0; i<index-2; i++)

p = p->next;

q = p->next;

p->next = q->next;

x = q->data;

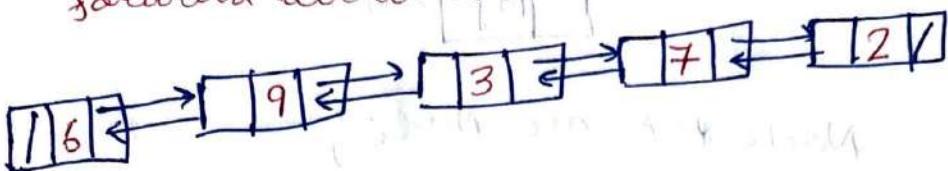
free(q);

return x;

}

Doubly Linked List.

In this we can move in forward as well as backward direction. In singly linked list we can only move in forward direction.



struct Node

```

struct Node *prev;
int data;
struct Node *next;
  
```

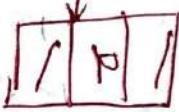
struct Node *pt;

b = new Node

t->prev = NULL;

t->data = 10;

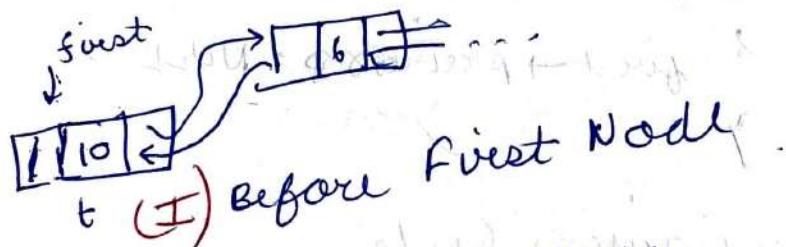
t->next = NULL;



Insert

1 Before first Node

2 Between other Nodes



Node *t = new Node

t->data = 2

t->prev = NULL

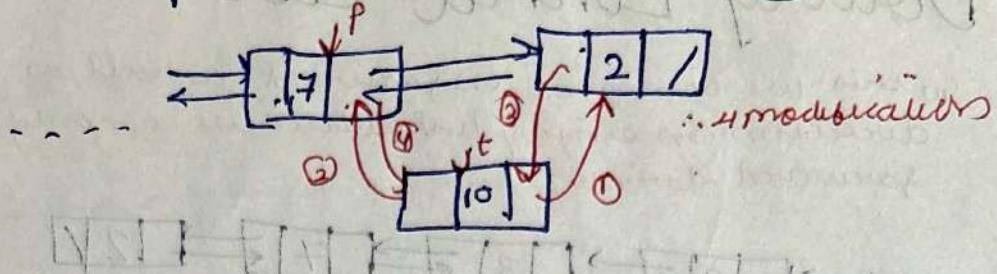
t->next = first

first->prev = t

3 links modified

first = t

(150) (II) Inserting at other position (4 lines are modified)



Node *t = new Node;

t->data = x

for (i=0; i < pos-1; i++)
p = p->next;

t->next = p->next; *about worst case*
OCN

t->prev = p; *i worst case*

If (p->next) *& mostly handle*

p->next->prev = t

p->next = t *for about average*

Deleting from Doubly List

① Deleting First Node.

p = first;

first = first->next

x = p->data

delete p; *wasted 1*

if (first)

1 first->previous = NULL

2 *both twist*

Deleting given Index

assuming : pos 4 (do)

p = first;

for (i=0; i < pos-1; i++)
p = p->next;

PO = p->prev->next = p->next;

P is ($p \rightarrow \text{next}$)

$\Rightarrow p \rightarrow \text{next} \rightarrow \text{prev} = p \rightarrow \text{prev}$.

$x = p \rightarrow \text{data}$
delete P;

Best case: $O(1)$
Worst case: $O(n)$

Reverse a Linked List.

p = first

while (P)

{ temp = $p \rightarrow \text{next}$

$p \rightarrow \text{next} = p \rightarrow \text{prev};$

$p \rightarrow \text{prev} = \text{temp}$

$p = p \rightarrow \text{prev};$

\oplus if ($p \rightarrow \text{next} = \text{NULL}$)
first = p;

y

~~IMP~~
we are just
reversing
prev and next
values
of each node

Code for Doubly Linked List

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

struct Node

{ struct Node *prev, int data, struct Node *next; }

int data;

struct Node *next;

}; $\oplus \& \text{first} = \text{NULL};$

void create(int A[], int n)

{ struct Node *t, *last; int i;

first = (struct Node *)malloc(sizeof(struct Node));

first->data = A[0];

152

```

first->prev = first->next=NULL;
last=first;
for(i=1; i<n; i++)
{
    t=(struct Node*)malloc(sizeof(struct Node));
    t->data=A[i];
    t->next=last->next;
    t->prev=last;
    last->next=t;
    last=t;
}

```

void display(struct Node *p)

```

while(p)
{
    printf("%d", p->data);
    p=p->next;
    printf("\n");
}

```

int length(struct Node *p)

```

int len=0;
while(p)
{
    len++;
    p=p->next;
}
return len;

```

{ A linked list stores data }

i first + c + < > each fields }

length(*list)=first

Total = m+1 + first

```

void Insert (struct Node *p, int index, int x)
{
    struct Node *t;
    int i;
    if (index < 0 || index > length(p))
        return;
    if (index == 0)
    {
        t = (struct Node *) malloc(sizeof(struct Node));
        t->data = x;
        t->prev = NULL;
        t->next = first;
        first->prev = t;
        first = t;
    }
    else
    {
        for (i=0; i<index-1; i++)
            p = p->next;
        t = (struct Node *) malloc(sizeof(struct Node));
        t->data = x;
        t->prev = p;
        t->next = p->next;
        if (p->next) p->next->prev = t;
        p->next = t;
    }
}

int delete (struct Node *p, int index)
{
    struct Node *q;
    int x = -1, i;
    if (index < 1 || index > length(p))
        return -1;
    if (index == 1)

```

```

{ first = first->next;
  if (first) first->prev=NULL;
  x=p->data;
  free(p);
}

else
{
  for (i=0; i< index-1; i++)
    p = p->next;
  p->prev->next = p->next;
  q = (p->next);
  p->next->prev = p->prev;
  x = p->data;
  free(p);
}

return x;
}

void Reverse(struct Node *p)
{
  struct Node *temp;
  while (p != NULL)
  {
    temp = p->next;
    p->next = p->prev;
    p->prev = temp;
    p = p->prev;
    if (p != NULL && p->next == NULL)
      first = p;
  }
}

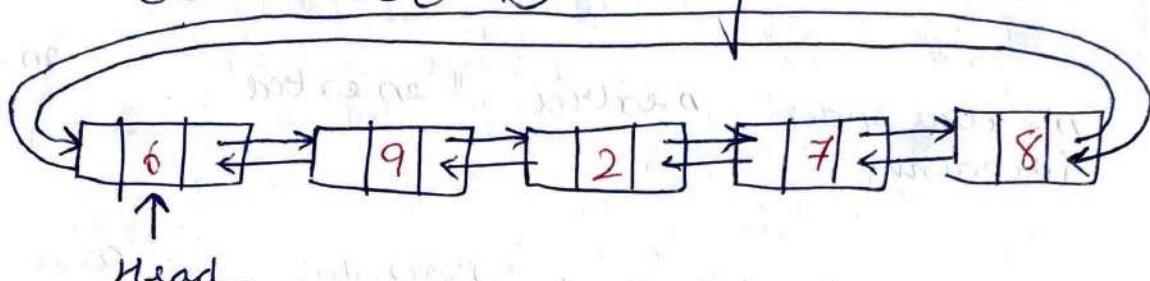
```

int main()

```
{ int A[] = { 10, 20, 30, 40, 50 } ;
    create(A, 5);
    reverse(first);
    display(first);
    return 0;
}
```

y

Circular Doubly Linked List.



Display

```
P = Head;
do
{ printf("%d", p->data);
  P = p->next;
}
```

} while (P != Head);

other operations are similar

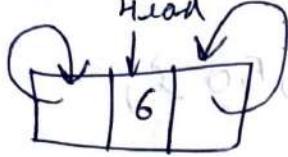
Inserting

For inserting after head at position x
 we need to take 'p' pointer to position x
 and then modify the links

For inserting before head no 'p' pointer
 is required only modifications in links
 are required.

156

Single Node



Comparision of Linked List.

~~Implementation~~

Linear Singly

Circular
SinglyLinear
DoublyCircular
Doubly

Space

n extra space
for point

n extra

n extra

n extra

Traverse

Forward

Forward
CircularBoth dir
clrsCircular
→ 8 dir
clrs.

Insert

Before
first $O(1)$ 1 link
modified $O(n)$ 2 links
are
modified $O(1)$ 3 link
modif. $O(1)$ 4 links
are(i) Any
other
positionmin: $O(1)$
max: $O(n)$ 2 links
modifiedmax: $O(n)$
min: $O(1)$ 2 links
modifiedmax: $O(n)$
min: $O(1)$ 4 links
modifiedmax: $O(n)$
min: $O(1)$ 4 links
modified

Delete

 $O(1)$ 0 links
modified $O(n)$ 1 links
modified3 links
(whichever is
last)
 $O(1)$ 2 links
are
modified $O(1)$ 2 links
are modifiedJAVA
has only
this type
of Linked
List.

ARRAY

can be in
Stack or Heap

cannot change
size

Chance of Perfect
Utilization of mem -
only if we

Space: only space
equal to data

Can be randomly
accessed

Inserting :

8	3	4	7	6	2		
---	---	---	---	---	---	--	--



$O(n)$
shifting
of Data

Deleting : Same
as Inserting

Search:

Linear Search: $O(n)$

Binary Search: $O(\log n)$

Linked List

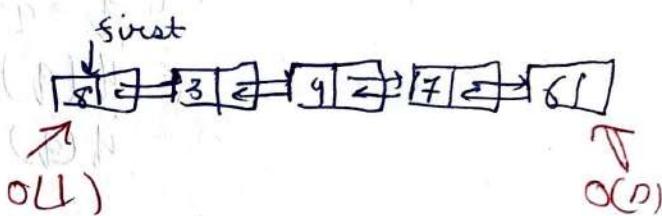
only in Heap.

Size can be changed

Perfect Utilization of
mem memory

Double the space of
more space than data

Can be a only sequen -
tially accessed.



Deleting: Same as
Inserting.

Linear search - $O(n)$

Binary : $O(n \log n)$
(not useful.)

Insertion & Merge
Sort are designed
for Link List

Finding middle node of a Linked List

Method I:

1. Find Length = 7
2. Recur to mid node $\left\lceil \frac{7}{2} \right\rceil = 4$

We have to scan list for 2 times

Method II:

Time O(n) space O(1)

Two pointers p & q • q moves 2 times
and p moves one time

$p = q \rightarrow \text{first};$

while (q)

{ $q = q \rightarrow \text{next}$

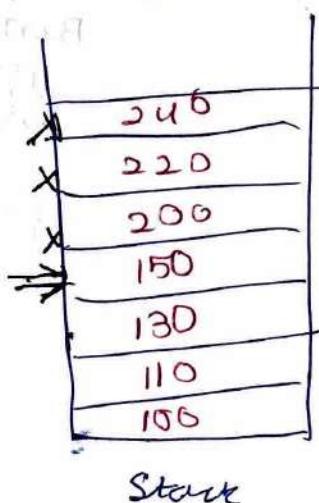
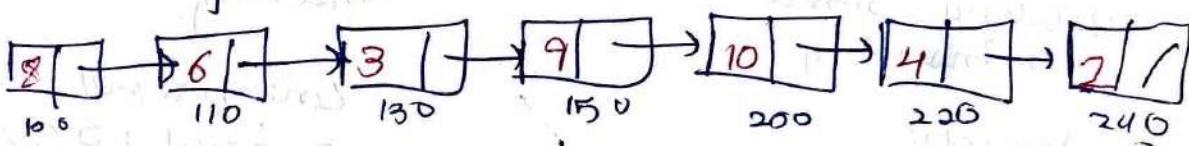
if (q) $q = q \rightarrow \text{next};$

if (q) $p = p \rightarrow \text{next};$

4

Method III

Using stack



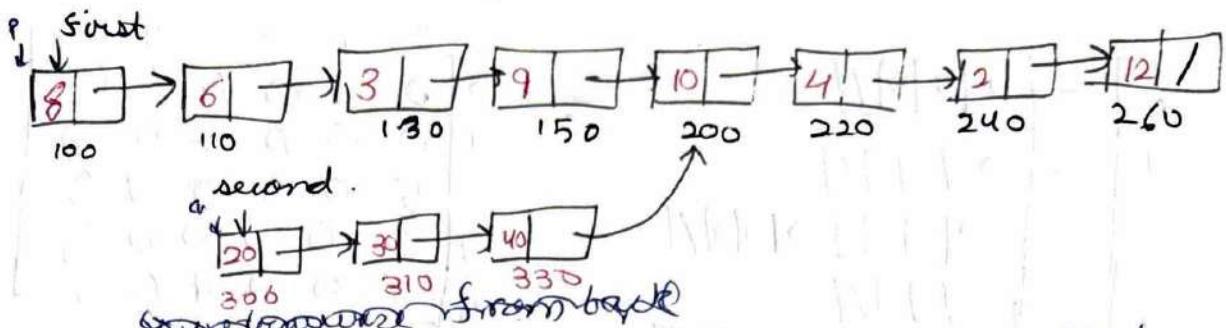
7 elements in stack

$$\left\lceil \frac{7}{2} \right\rceil = 3$$

∴ 3 elements will be popped out

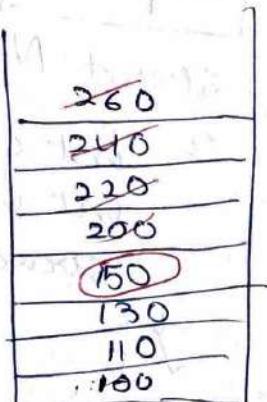
Time complexity: $O(n)$

Finding Intersection of 2 Linked List

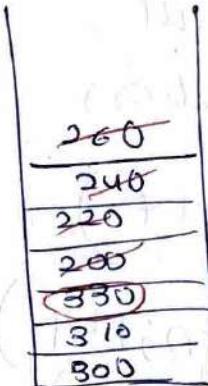


If we traversal from back then after Node intersection address of both will be different but reverse-traversal is not possible in single linked list. ∴ we use stack.

260
240
220
200
150
130
110
100
Entire-
ctors
point.



S1



S2

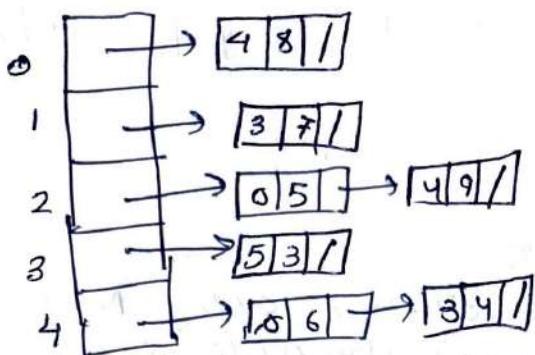
$\frac{n}{n}$
 $\frac{n}{n}$
 $\frac{3n}{3n}$
 $10(n)$

 $p = \text{first}$ while ($p \neq \text{null}$) push (&stk1, p); $p = \text{second}.$ while ($p \neq \text{null}$) push (&stk2, p);

while (stackTop(stk1) == stackTop(stk2))

{ $p = pop(\&stk1)$; $pop(\&stk2)$;}
printf("1. d", p->data);

Sparse Matrix Representation



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

$m \times n$
 5×6

Node *A[m] // arrays & linked list

A[0] = new Node

& similarly for all .

// display function

```

for(i=0; i<m; i++)
{
    p = A[i];
    for(j=0; j<n; j++)
    {
        if(p == NULL)
            printf("0");
        else
        {
            printf("%d", p->val);
            p = p->next;
        }
    }
    printf("\n");
}
    
```



struct Node

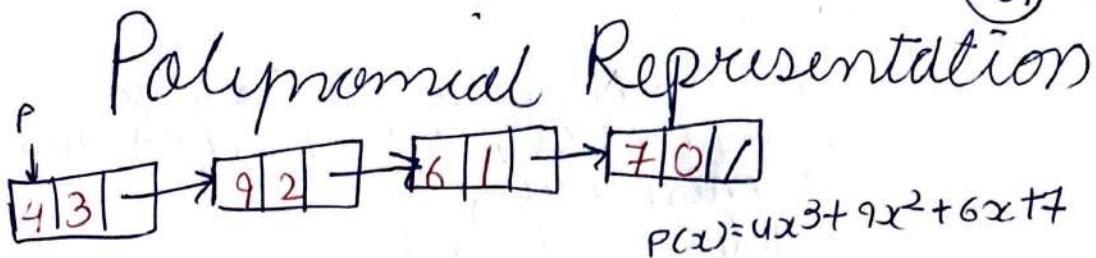
{ int col;

int val;

struct Node* next;

j

// Here when P becomes NULL if
should not check p->col .
There will be add on in this



How to evaluate this Express

let $x=2$

double eval(int x)

{ double sum=0.0;

Node *q=p;

while(q!=NULL)

{ q->exp)*q->coeff

sum+=pow(x,q->exp);

q=q->next;

}

return sum;

}

STUDENT EXERCISE

Adding two Polynomials:

lets Code Polynomial:

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

struct Node

{ int coeff;

int exp;

struct Node *next;

} *poly=NULL;

void create()

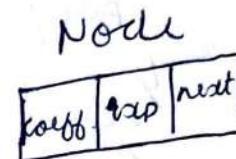
{ struct Node *t, *last=NULL;

int num,i;

printf("enter number of terms");

scanf("%d", &num);

printf("enter each term with coeff and exp ");



struct Node

{ int coeff;

int exp;

struct Node *next;

};

162
 for (i=0; i<num; i++)
 {
 t=(struct Node *)malloc(sizeof(struct Node));
 scanf("%d %d", &t->coeff, &t->exp);
 t->next=NULL;
 if (poly==NULL)
 { poly=last=t;
 }
 else
 { last->next=t;
 last=t;
 }
 }
 void display (struct Node *P)
 { while (P)
 { printf("%d x^%d + ", P->coeff, P->exp);
 P=P->next;
 }
 printf("\n");
 long eval (struct Node *P, int x)
 { long val=0;
 while (P)
 { val+=P->coeff * pow(x, P->exp);
 P=P->next;
 }
 return val;
 }
 int main()
 { create();
 display(poly);
 printf("1. add 2. mul 3. eval 4. exit\n");
 eval(poly, 1));
 return 0;
}

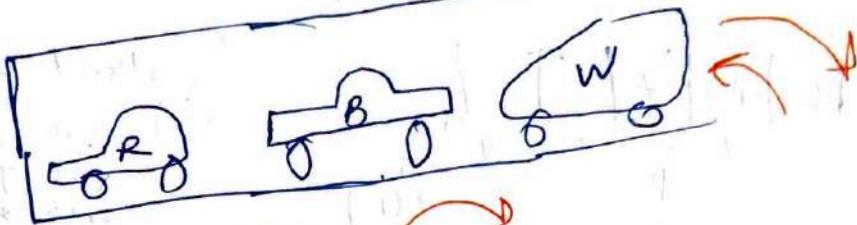
Stack

LIFO: Last-in First out

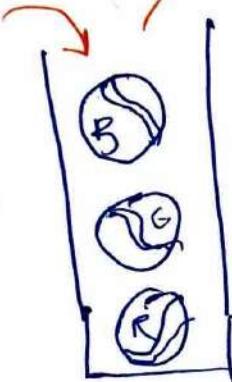
Collection of elements which follow LIFO

examples

(i)



(ii)



(iii) Recursion uses stack

converting into iterative, programmer has to create stack sometimes

ADT Stack

Data:

- 1. Space for storing elements

- 2. Top pointer

operations

- 1. push(x) add element at top

remove top element

- 2. pop()

Removing a value at a position
from top.

- 3. peek(index)

Knowing a value at a position
from top.

- 4. stackTop()

to know topmost value

- 5. isEmpty()

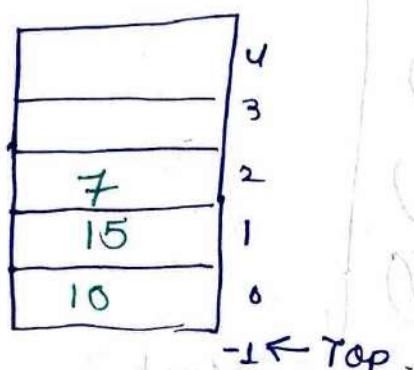
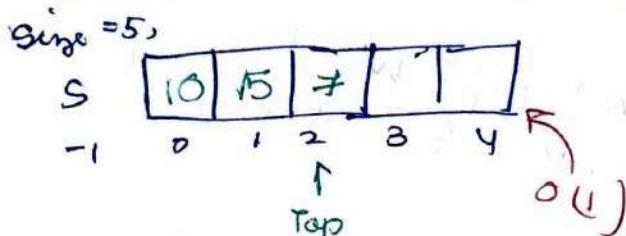
whether stack is empty or not

- 6. isfull()

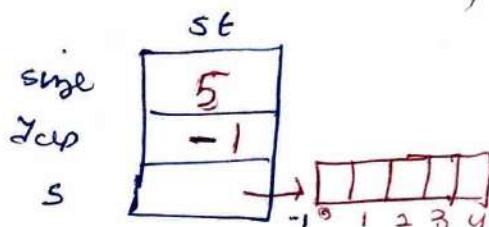
Implementation of Stack can be done by

- (i) Array
- (ii) Linked List.

Implementation of Stack using ARRAYS



struct stack
{ int size;
int Top;
int *S;
};



```
int main()
{
    struct Stack st;
    printf("Enter size of stack");
    scanf("%d", &st.size);
    st.S = new int [st.size];
    st.Top = -1;
}
```

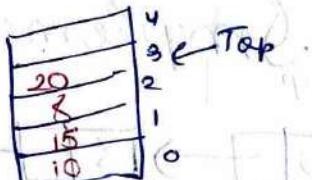
empty
if ($Top == -1$)
Full
if ($Top == size - 1$)

$\Rightarrow \text{push}(\text{Stack } *st, \text{int } x)$

{ if $Cst \rightarrow Top == st \rightarrow size - 1$
 printf("Stack overflow");
 else
 { $st \rightarrow Top++;$ Time-complexity: O(1)
 $st \rightarrow S[st \rightarrow Top] = x;$
 }
 }
 }

$\Rightarrow \text{pop}(\text{Stack } *st)$ when stack is empty
it will give -1

{ int $x = -1;$
 if $(Cst \rightarrow Top == -1)$
 printf("Stack underflow");
 else
 { $x = st \rightarrow S[st \rightarrow Top];$ Time complexity: O(1)
 $st \rightarrow Top--;$
 return $x;$
 }



Peek

Pos	Index = Top - Pos + 1
1	3
2	2
3	1
4	0

$$\begin{aligned}3-1+1 &= 3 \\3-2+1 &= 2 \\3-3+1 &= 1 \\3-4+1 &= 0\end{aligned}$$

peek (stack st, pos)
{ int $x = -1;$
 if $(st \rightarrow Top - pos + 1 < 0)$
 printf("Invaled position");
 else
 $x = st \rightarrow S[st \rightarrow Top - pos + 1];$
 return $x;$

4

166

int stackTop(stack st)

1 if ($st.\text{Top} == -1$) $O(1)$
 returns -1;
else

 returns $st.s[st.\text{Top}]$;

int isEmpty(stack st)

1 if ($st.\text{Top} == -1$) $O(1)$
 returns 1;

else
 returns 0;

int isFull(stack st)

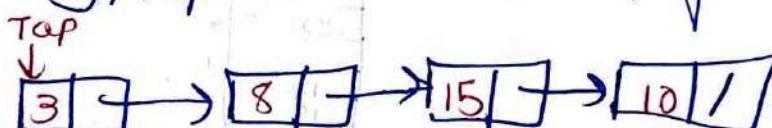
1 if ($st.\text{Top} == (st.\text{size} - 1)$) $O(1)$
 returns 1;

else
 returns 0;

Implementation by Linked List.

pushing
 \Rightarrow
a pop from
this direction

$O(1)$



struct Node
{
 int data;
 struct Node* next;
};

$O(n)$

elements
are inserted
from here!

⇒ No specific size like array

↑ Empty

$\text{if } (\text{Top} == \text{NULL})$

full

$\text{Node* t = new Node}$

$\text{if } (\text{t} == \text{NULL})$

Push Function

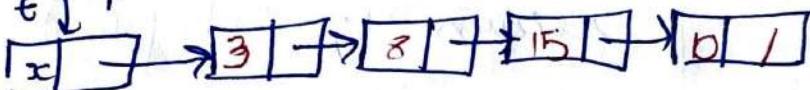
void push(int x)

1

```
Node *t = new Node  
if (t == NULL)  
    printf("Stack Overflow");  
else  
    t->data = x  
    t->next = Top;  
    Top = t;
```

}

\downarrow
t
Top



Pop Function

int pop()

1 Node *P;

int x = -1;

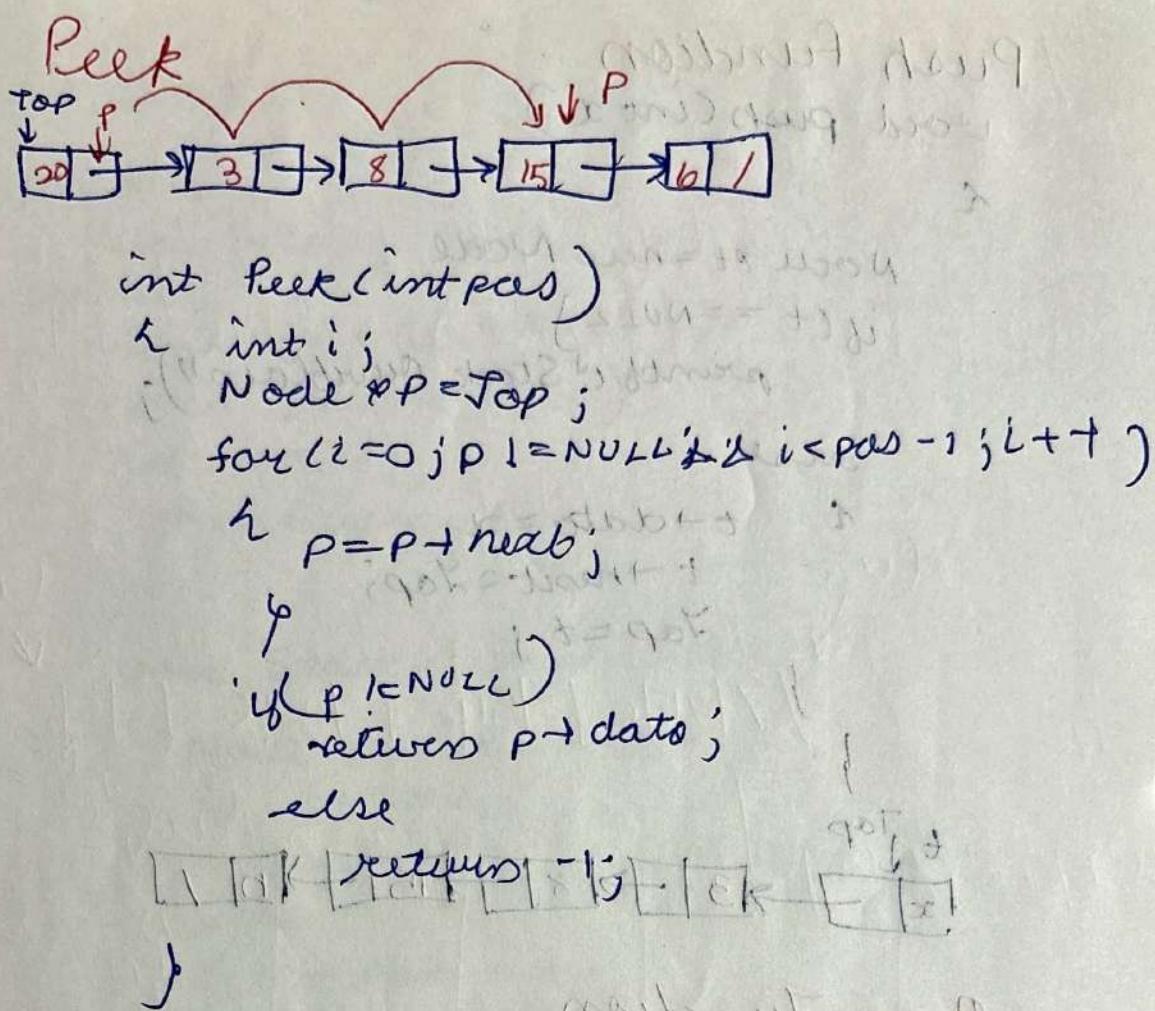
base case if (Top == NULL)
 printf("Stack is Empty");

else

```
P = Top;  
Top = Top->next;  
x = P->data  
free(P);
```

return x;

4



```
int stackTop()
```

```

    if (Top)
        return Top->data;
    return -1;
}
```

```
int isEmpty()
```

```

    return Top ? 0 : 1;
}
```

```
int isFull()
```

```

    Node *t = new Node;
    int r = t->data;
    free(t);
    return r;
}
```

C++ class for Linked List.

```
#include <iostream.h>
using namespace std;
class Node
{
public:
    int data;
    Node *next;
};
```

class Stack

{ private:

Node *top;

public:

Stack() { top = Null; }

void push(int x)

int pop();

void display();

}

void Stack::push(int x)

{ Node *t = new Node;

if (t == NULL)

cout << "Stack is Full\n";

else

{ t->data = x;

t->next = top;

top = t; }

}

int Stack::pop()

{ int x = -1;

if (top == NULL)

cout << "Stack is Empty\n";

else

{ x = top->data;

Node *t = top;

top = top->next;

delete t;

return x; }

void Stack::display()

{ Node *p = top;

while (p != NULL)

{ cout << p->data << " ";

p = p->next; }

170

cout << endl

```

int main ()
{
    stack stk;
    stk.push(10);
    stk.push(20);
    stk.push(30);
    stk.display();
    cout << stk.pop();
    return 0;
}

```

Paranthesis Matching

$$(a+b)*(c-d))$$

We have to check whether all brackets are closed or not we will store (' ') in stack while we found '(' we will push in stack & while we found ')' we pop from stack. At last when string is traversed the stack should be empty. If not then parentheses are not matching.

Here only Balance of parentheses are checked

exp
$$\boxed{((a + b) * (c - d)) / 10}$$

0	1	2	3	4	5	6	7	8	9	10	11	12	13
---	---	---	---	---	---	---	---	---	---	----	----	----	----

int isBalance(char *exp)

```

    struct Stack st;

```

```

    st.size = strlen(exp);

```

```

    st.top = -1;

```

```

    st.s = new char [st.size];

```

```

struct Stack
{
    int size;
    int top;
    char *s;
}

```

y

//Initiating stack

for (i=0; exp[i] != ']' ; i++)

{ if (exp[i] == 'c') push(&st, exp[i]);

else if (exp[i] == ')')

{ if (isEmpty(st)) return false;

else

pop(&st);

return isEmpty(st) ? true : false;

More on Parenthesis Matching

1. ([a+b]*[c-d]) le

Programs

int isBalance(char *exp)

{ char t;
struct Stack st; //①

st.size = strlen(exp)

setTop = -1;

st.s = new char[st.size];

for (i=0; exp[i] != ']' ; i++)

{ if (exp[i] == 'c' || exp[i] == '{' || exp[i] == '[')

{ push(&st, exp[i]);

} + * D

else if (exp[i] == ')' || exp[i] == '}' || exp[i] == ']')

{ if (exp[i] == ')' && isEmpty(st)) return false;

else { pop(&st); }

{ if (t != exp[i])

{ return false; }

return isEmpty(st) ? true : false;

Infix to Postfix Conversion

- 1) What is Postfix
- 2) Why Postfix
- 3) Precedence
- 4) Manual Conversion

1) Infix : operand operator operand

eg: $a + b$ means a added with b

2) Prefix operation opnd opnd

eg: $+ a b$ add a & b

3) Postfix opnd opnd operation

eg: $a b + a b$ are added

① $a + b * c$

$(a + (b * c))$

symbol	Precedence
$+$	1
$*$	2
$)$	3

prefix

$(a + [*bc])$

$+ a * bc$

postfix

$(a + [bc*])$

$abc*+$

not commonly used.

$$\textcircled{2} \quad a+b+c * d$$

~~prefix~~
 $a+b+(c * d)$

$$\underline{a+b+[*cd]}$$

$$+ab+[*cd]$$

$$++ab*cd$$

~~postfix~~
 $a+b+c * d$

$$\underline{a+b+[cd*]}$$

$$[ab+] + [cd*]$$

$$[ab+][cd*]+$$

$$\textcircled{3} \quad (a+b) * (c-d)$$

~~Prefix~~

$$(a+b)* (c-d)$$

~~Postfix~~

$$[ab+] * \underline{(c-d)}$$

$$[ab+] * [cd-]$$

$$\underline{[ab+][cd-]*}$$

$$ab+cd-*$$

left side
first
* bracket
evaluati
son

$$[ab+] * \underline{(c-d)}$$

$$[+ab] * [-cd]$$

$$*\underline{[ab] [-cd]}$$

$$+ab - cd$$

Associativity

Syb	Pre	Asso
+,-	1	L-R
* , /	2	L-R
^	3	R-L
-	4	R-L
()	5	L-R

i) $a+b+c-d$
 $\downarrow ((a+b)+c)-d$

ii) $a=b=c=5$

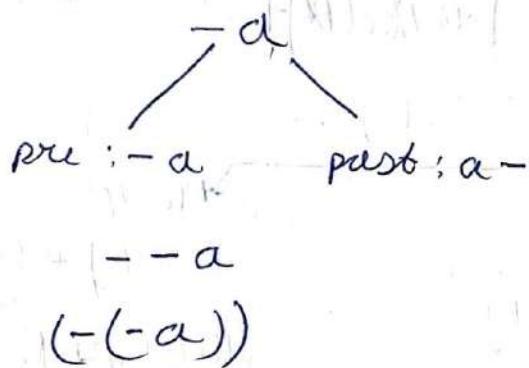
$$(a=(b=(c=5))) \text{ goes to } 2$$

(i) Conversion of (i) in postfix $a+b+c+d$

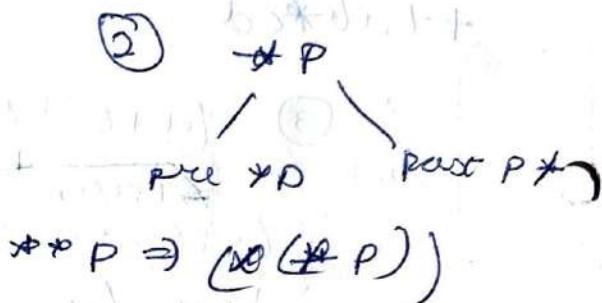
(ii) a^b^c from RPN c^b^a $a^b^c = [abc]^{^N}$
in postfix

Unary Operator

①



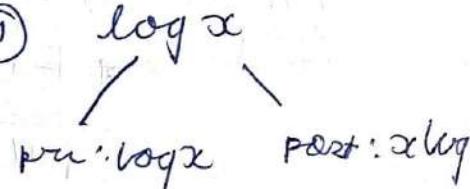
②



③



④



Unary operators have higher precedence

$$-a + b * \log \underline{n!}$$

$$-a + b * \log \underline{\underline{n!}}$$

$$\underline{-a + b * [n! \log]}$$

$$[a-1 + b * \underline{\underline{\underline{n! \log}}}]$$

$$[a-1 + \underline{\underline{b n! \log}}]$$

$$[a-6n! \log]$$

3 unary operators

$\log, !, \underline{\underline{\underline{n!}}}$ now to convert into postfix highest precedence & associativity applies

Infix to Postfix using Stack

Method I

$a + b * c - d / e$

sym	pre	Asso
+	1	L-R
*	2	L-R

sym	stack	postfix
a	empty	a
+	+,	a
b	+,	ab
*	* 2, +,	ab
c	* 2, +,	abc
-	-,	abc +
d	-,	abcd
/	/ -,	abcd /
e	/ -,	abcd / e
-		abcd / e -

Procedure :

→ operand 'a' sent it to postfix
 → '+' push it

→ 'b' sent it to postfix

→ '*' precedence 2 & precedence to op. Top in stack is 1
 ∴ push '*' in stack ∵ it is greater

→ 'c' sent it to postfix

→ '-' precedence is one But Top of stack has precedence 2 ∴ pop. Now Top has precedence 1 ∵ now it cannot be equal
pop '+' also. Now stack is empty ∵ precedence is zero. ∴ push '-'

→ 'd' sent to postfix

→ '/' push it

→ 'e' sent to postfix

→ end ∵ pop '/' & then pop '-'

Sym	Pre	DSO
+,-	1	L-R
*,/	2	L-R
a,b,c	3	L-R → having highest precedence

In this every symbol is gaining through stack

Programs for Infix to Postfix Conversion

infix $a + b * c - d / e | 0$

~~char * convert (char * infix)~~
~~struct Stack st; // Global~~

```
int isOperand(char x)
{
    if (x == '+' || x == '-' || x == '*' || x == '/')
        return 0;
    else
        return 1;
}
```

```
int pre(char x)
{
    if (x == '+' || x == '-')
        return 1;
    else if (x == '*' || x == '/')
        return 2;
    else
        return 0;
}
```

Sym	Pre	DSO
+	1	L-R
-	2	L-R

```

char * convert(char * infix)
{
    struct stack st; // Initialized
    char * postfix = new char [strlen(infix) + 1];
    int i=0, j=0;
    while (infix[i] != '\0')
    {
        if (!isOperand(infix[i]))
            postfix[j++] = infix[i++];
        else
        {
            if (Pre(infix[i]) > Pre(stackTop(st)))
                push(&st, infix[i++]);
            else
                postfix[j++] = pop(st);
        }
    }
    while (!isEmpty(st))
        postfix[j++] = pop(st);
    postfix[j] = '\0';
    return postfix;
}

```

↳ returns long type of data.

↳ now i++ is not done. we have use while & not for because in for i++ would have been done always.

In the above discussion we discussed operators having associativity from Left to Right.

(76)

Infix to Postfix with Associativity and Parenthesis

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

$$((ab+)*c) - d^e\underline{f}$$

$$\cancel{((ab+c)*)} - d^e [ef^]$$

$$\cancel{((ab+c)*)} - [def^{^n}]$$

$$\cancel{((ab+c)*d e f^{^n})}$$

Symbol	out stack prec	in stack prec
+	1	2
-		4
*	3	5
/		
\wedge	6	7
(7.	0
)	0	

$$\underline{\underline{ab + c * def^{^n} -}}$$

Here is L-R associativity outStack Prec < inStack Prec
 & is R-L associativity outStack Prec > inStack Prec

symbol	stack	Postfix
(C_0	
*	$C_0 C_0$	a
a		a
+	$+_2 C_0 C_0$	ab
b		ab+
)	C_0 <small>pop +, (with prec 01) = C pop & this don't pop the other one. & (&) are not saved in postfix</small>	ab+
*	$*_4 C_0$	ab+
c	$*_4 C_0$	ab+c
)	empty	ab+c*
-	-	ab+c*

d	-2	$ab+c+d$
\wedge_6	$\wedge_5 - 2$	$ab+c+d$
\wedge_6	$\wedge_5 - 2$	$ab+c+d$
\wedge_6	$\wedge_5 \wedge_5 - 2$	$ab+c+d$
f	$\wedge_5 \wedge_5 - 2$	$ab+c+d$
END	empty	$ab+c+d$

In this) will never be copied in Postfix & (also.

Evaluation of Postfix Expression

$$3 * 5 + 6 / 2 - 4 = 14$$

$$3 5 * 6 2 / + 4 -$$

Idea: every no is pushed in stack and as soon as operator is encountered we pop 2 operands & perform operations & push it on the stack.

symbol	stack	operations Happening
3 (i)	3	
5 (ii)	5 3	
*	15	$3 * 5$
6 (iv)	6 15	
/ (v)	2 6 15	$6 / 2$
+	3 15	$15 + 3$
4 (vii)	18	$18 - 4$
- (ix)	14	
END	EMPTY	result : 14

NOTE: Precedence & Associativity are meant for parentheses they don't decide who will execute first ex: $x = 6 + 5 * 3 - 4$ According to Precedence 3 & 4 are multiplied first.

$$x = 6 + 34 \cancel{*} 4 + =$$

Now we evaluate by above Method 6 & 5 will be added

180

int Eval(char *postfix)

```

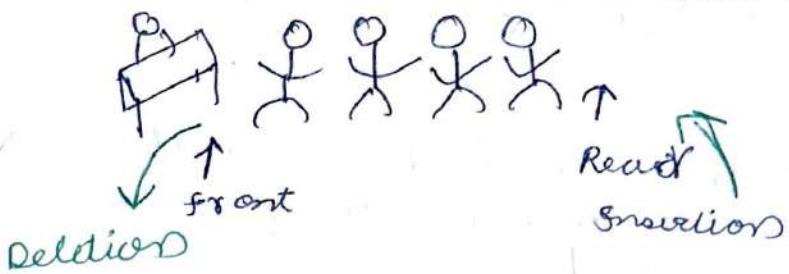
    {
        int i = 0
        int x1, x2, r = 0;
        for (i = 0; postfix[i] != '\0'; i++)
        {
            if (isOperand(postfix[i]))
                push(postfix[i] - '0');
            else
            {
                x2 = pop();
                x1 = pop();
                switch (postfix[i])
                {
                    case '+': r = x1 + x2; break;
                    case '-': r = x1 - x2; break;
                    case '*': r = x1 * x2; break;
                    case '/': r = x1 / x2; break;
                }
                push(r);
            }
        }
        return top->data;
    }

int main()
{
    char *postfix = "234*+82/-";
    printf("Result is %.d\n", Eval(postfix));
    return 0;
}

```

Queue

FIFO : First in First out



Queue ADT

Parts :

- 1) space for storing elements.
- 2) front for deletes (points)
- 3) rear for insertions (points)

operations

- 1) enqueue () inserting in queue
- 2) dequeue () deleting from queue
- 3) isEmpty ()
- 4) isFull ()
- 5) first ()
- 6) last ()

Queue can be implemented by 2 physical data

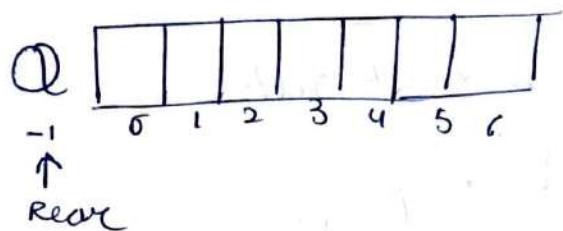
structures : (i) Array
(ii) Linked List.

Queues using Array

- 1) Queue using single pointer.
- 2) Queue using front and rear
- 3) Drawbacks of queue using ARRAY

182

size = 7



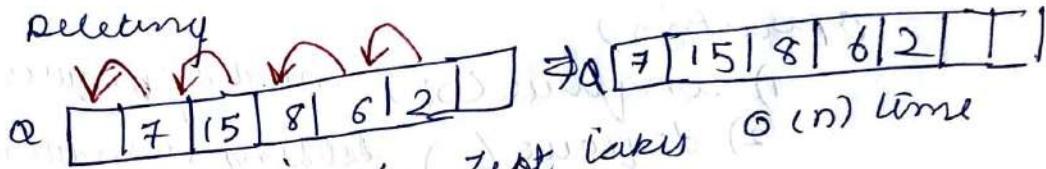
To add element move rear to next and then insert value at that position.

[Insert : $O(1)$]

For deleting the element we reverse $A[0]$ and $A[1]$ and shift elements to left size.



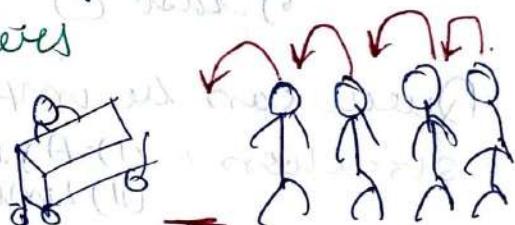
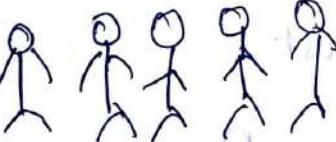
deleting



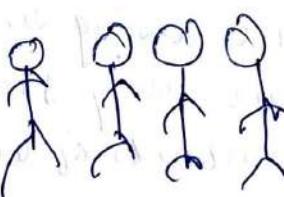
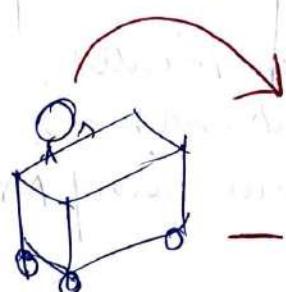
∴ Moving to last takes $O(n)$ time

[Deleting : $O(n)$]

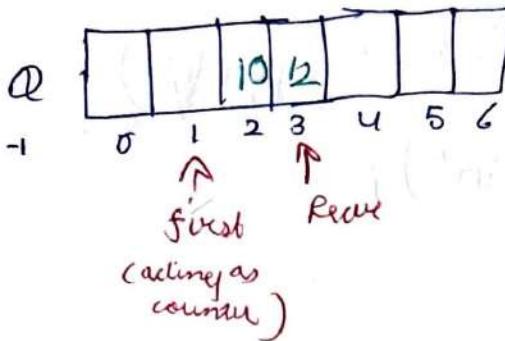
since Deleting is taking $O(n)$ we will use Two Pointers



Deleting using one pointer



Deleting using 2 pointers



83

enqueue - $O(1)$
dequeue - $O(1)$

Empty
if (front == rear)

Full
if (rear == size - 1)

Code:

```
#include <iostream>
using namespace std;
```

```
class Queue
```

```
{ private:
```

```
int front;
```

```
int rear;
```

```
int size;
```

```
int *Q;
```

```
public:
```

```
Queue() { front = rear = -1; size = 10; Q = new int[size]; }
```

```
Queue(int size) { front = rear = -1; this->size = size;
```

```
Q = new int[this + size]; }
```

```
}
```

```
void enqueue(int x);
```

```
int dequeue();
```

```
void display();
```

```
}
```

void Queue::enqueue(int x)

{ if (rear == size - 1)
printf("Queue Full\n"); }

else

{ rear++;
Q[rear] = x; }

}

int Queue::dequeue()

{ int x = -1;

if (front == rear)
printf("Queue is Empty\n");

else

{ x = Q[front + 1];
front++; }

}

return x;

}

void Queue::display()

{ for (int i = front + 1; i <= rear; i++)
printf("%d", Q[i]);
printf("\n"); }

int main()

{ Queue q(5);

q.enqueue(10);

q.enqueue(20);

q.enqueue(30);

q.display();

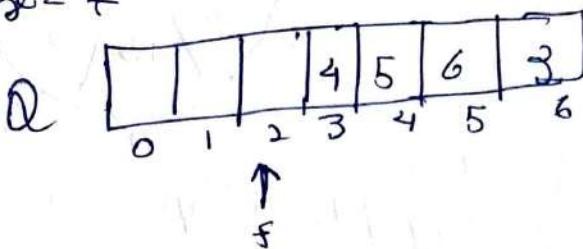
return 0;

}

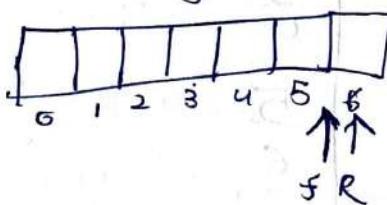
REMEMBER
 front marks
 start of none b.
 free node ?.

Drawback of Queue using Array.

$\text{size} = 7$



- (i) We cannot reuse the deleted memory
- (ii) We can use these space only one time
- (iii) There can be possibility when queue is full but actually it is fully empty.



we want to reuse the space

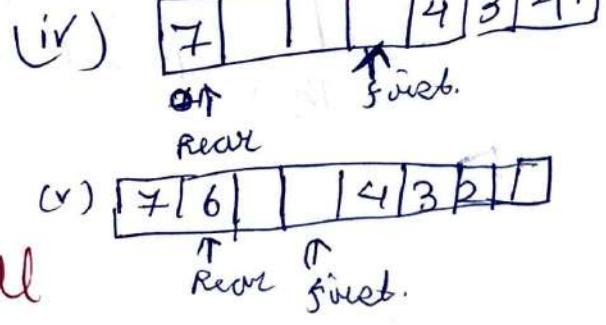
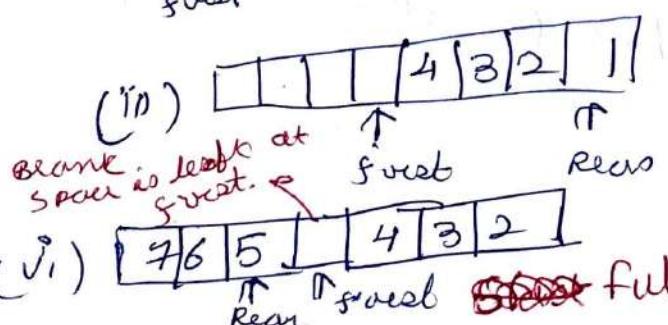
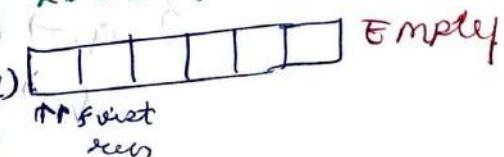
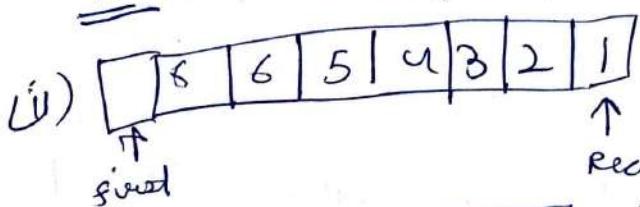
Solutions :

1 Resetting Pointers

As soon as Queue becomes empty.
ie $\text{first} = \text{rear} = -1$
Reset $\text{first} = \text{rear} = -1$

THIS DOES NOT GUARANTEE REUSE OF MEMORY. IF QUEUE WILL NOT BECOME EMPTY IT WON'T REUSE.

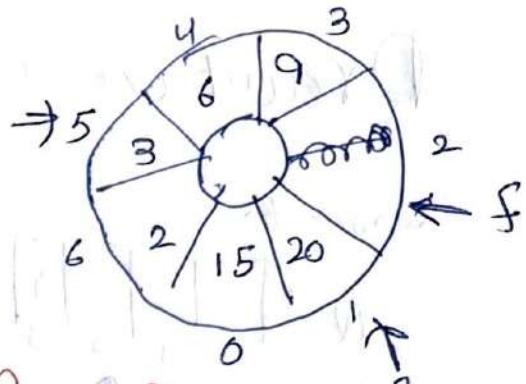
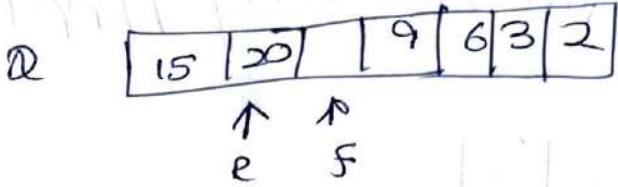
2 Circular Queue



full

first.

size = 7



This circular fashion can be achieved by Mod Operator to achieve circular values

$$\text{rear} = (\text{rear} + 1) \% \text{size}$$

0	$(0+1)\%7$	1
1	$(1+1)\%7$	2
2	$(2+1)\%7$	3
3	$(3+1)\%7$	4
4	$(4+1)\%7$	5
5	$(5+1)\%7$	6
6	$(6+1)\%7$	0

```

void enqueue (struct Queue *q, int a)
{
    if ((a + rear + 1) \% q->size == q->front)
        printf ("Queue is full");
    else
        q->rear = (a + rear + 1) \% q->size;
        q->Q[q->rear] = a;
}
    
```

This is Best Method for Queue using array.

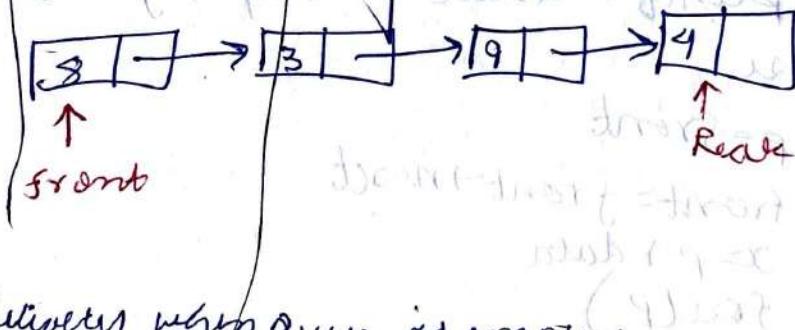
int dequeue (struct Queue *q)

```
{ int x = q->front;
  if (q->front == q->rear)
    printf ("Queue is Empty");
  else
  {
    q->front = (q->front + 1) % q->size;
    x = q->Q[q->front];
  }
  return x;
```

void display (struct Queue q)

```
{ int i = q.front + 1;
  do
  { printf ("%d", q.Q[i]);
    i = (i + 1) % q.size
  } while (i != (q.rear + 1) % q.size);
  printf ("\n");}
```

Queue using Linked List



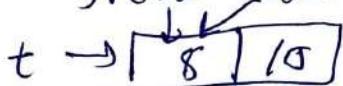
- Initially when queue is empty
front = rear = NULL

Empty

if (front == NULL)

- Get first element

front rear



Full

```
Node *t = new Node  
if (t == NULL)
```

```
void enqueue (int x)
```

```
{ Node *t = new Node;  
if (t == NULL)  
    cout << "Queue is Full";
```

```
else
```

```
{ t->data = x;  
t->next = NULL;  
if (front == NULL) front = rear = t;  
else
```

```
{ rear->next = t;  
rear = t;
```

```
y
```

```
int dequeue ()
```

```
{ int x = -1;
```

```
Node *p ;  
if (front == NULL)
```

```
    cout << "Queue is Empty";
```

```
else
```

```
{ p = front
```

```
front = front->next
```

```
x = p->data
```

```
free(p)
```

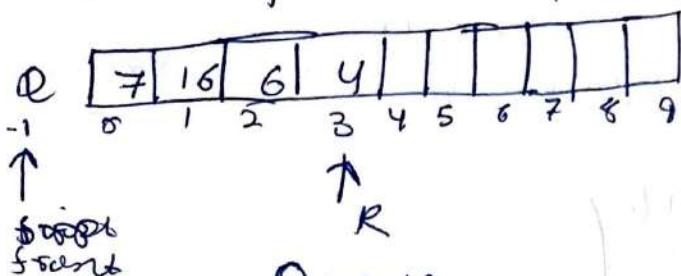
```
y
```

```
return x;
```

```
}
```

Time Complexity: O(1)

Doubly Ended Queue (DEQueue)



Queue

	Insert	Delete
front	X	✓
Rear	✓	X

DE Queue		Insert	Delete
front	✓	✓	
Rear	✓	✓	

DeQueue is of TWO Types

i) P Restricted (Input i/P)

O/P Restricted content o/p

DEQUEUE

	Insert	Delete
front	X	✓
Rear	✓	✓

	Insert	Delete
front	✓	✓
Rear	✓	X

Means deletion is Restricted
∴ front can only delete

Priority Queues.

1) Limited Set of Priorities

2) Element Priorities

Limited Priority

smaller priorities = 3 Rich (front me khada b queue)
 the priority element = $\rightarrow A, B, C$ Middle class
 no greater D E F G H Poor (sabse reache b queue
 priority I J me)

Priority Queues: (FIFO is applicable)

Q1	A	B	F		
----	---	---	---	--	--

queue for priority 2

Q2	C	E	G	I	J
----	---	---	---	---	---

Q3	D	H			
----	---	---	--	--	--

For deleting you must delete from highest priority queue i.e first Q_1
 If Q_1 is empty then go to Q_2 & similarly

Element Priority

elements $\rightarrow 6, 8, 3, 10, 15, 2, 9, 17, 5, 8$

smaller numbers

Higher Priority



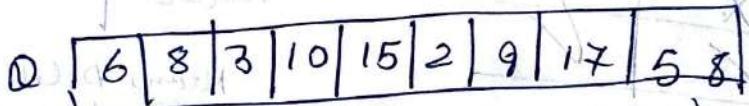
is Array
not Queue

Method 1) Insert in same order

delete Min Priority by searching it.

2) Insert in Increasing Order of Priority
 Delete Last element of Array.

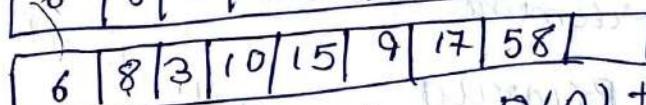
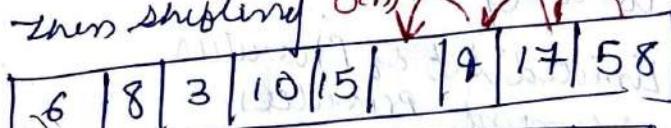
Method I



Min will be searched $\therefore O(n)$

$\because \text{Min} = 2$

\therefore value whose 2 is stored is deleted
 then shifting $O(n)$



\therefore Total Deleting: $O(n) + O(n)$

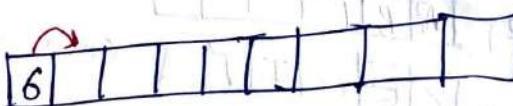
$$= O(2n) = \underline{\underline{O(n)}}$$

Inserting
 $O(1)$

Deleting
 $O(n)$

Method II

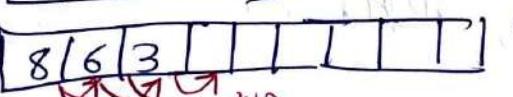
(i) Q



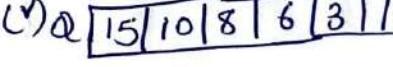
(ii) Q



(iii) Q



+15



+10

\therefore Inserting: $O(n)$

141

Now Deleting:

Since Array Q is in descending order \therefore Value at last pointers will be of least priority
 \therefore Directly delete last elements.

• Inserting
 $O(n)$

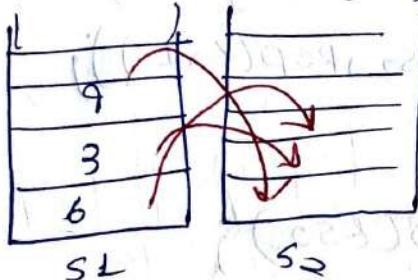
Deleting
 $O(1)$

Refer to some other upcoming part for more...

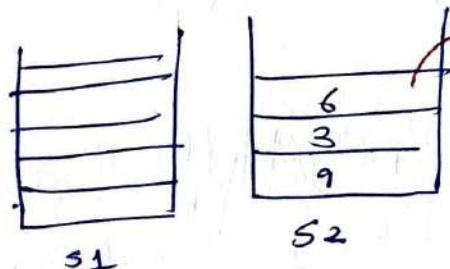
Queue using 2 stacks.

elements $\rightarrow 6, 3, 9, 5, 4, 2, 8, 12, 10$

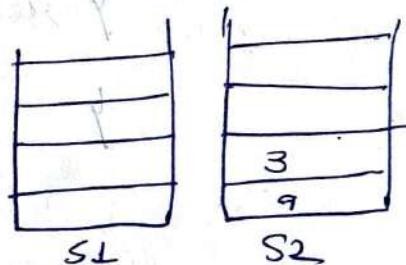
- (i) enqueue(6)
- (ii) enqueue(3)
- (iii) enqueue(9)



- (iv) dequeue()



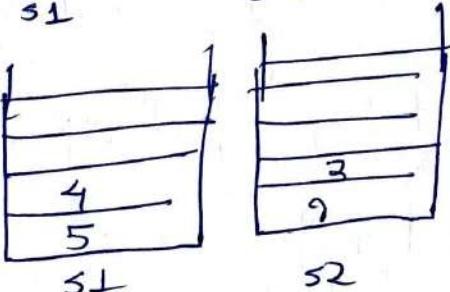
POP \Rightarrow



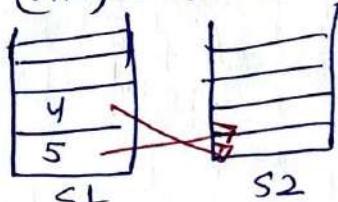
- (v) enqueue(5)

poped 9

- (vi) enqueue(4)



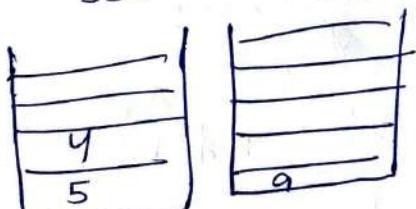
- (vii) dequeue()



- (ix) dequeue()

poped 3

- (viii) dequeue()



- (x) dequeue()

pop



- (xi) dequeue()



- (xii) dequeue()



(192)

enqueue(int x)

{ push(&s1, x);

// Here we didn't checked
whether stack were empty
or full

int dequeue()

{ int x = -1;

if (isEmpty(s2))

{ if (isEmpty(s1))

{ cout << "QueueEmpty";

return x;

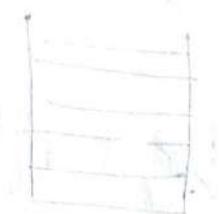
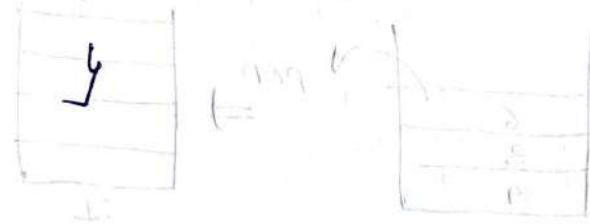
else while (!isEmpty(s1))

{ ~~while~~ push(&s2, pop(&s1));

}

} return pop(&s2);

}



Q. 1. (a) (iii)



Q. 1. (b) (iv)



(a) (iii), (iv)
(b) (ii), (iv)

(c) (i), (ii)

Easy Method

(ii)

Implementation of Circular Queue

```
template <typename T>
class Queue {
    T *arr;
    int f;
    int r;
    int ms; // maximum size.
    int cs; // current size.
```

public:

// constructors

Queue(int ds = 4)

```
{
    f = 0;
    ms = ds;
    r = ms - 1;
    arr = new T[ms];
    cs = 0
}
```

// enqueue

bool isFull() {
 return cs == ms;

bool isEmpty() {
 return cs == 0;

void push(T data)

```
{
    if (!isFull()) {
        r = (r + 1) % ms;
        arr[r] = data;
        cs++;
    }
}
```

Taking current size and maximum size makes implementations much easier.

(ii)

```
void pop() {
    if (!isEmpty()) {
        f = (f + 1) % m
        cs--;
    }
}
```

f
↑

```
T getfront() {
    returns arr[f];
}
```

}

⇒ In STL;

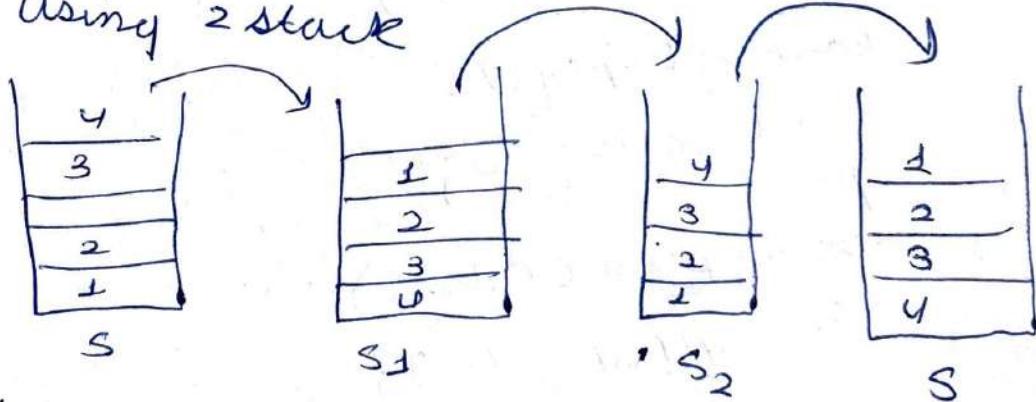
queue <int> q;

q.push(i)

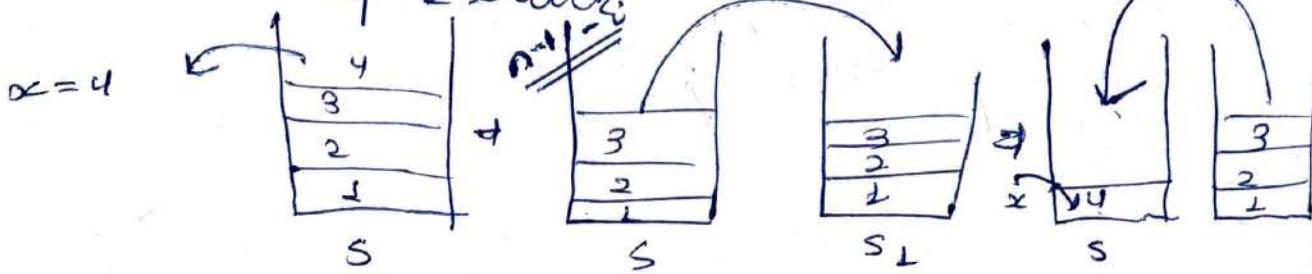
defining q.pop() } in STL
class is
stl. q.empty()
 q.front()

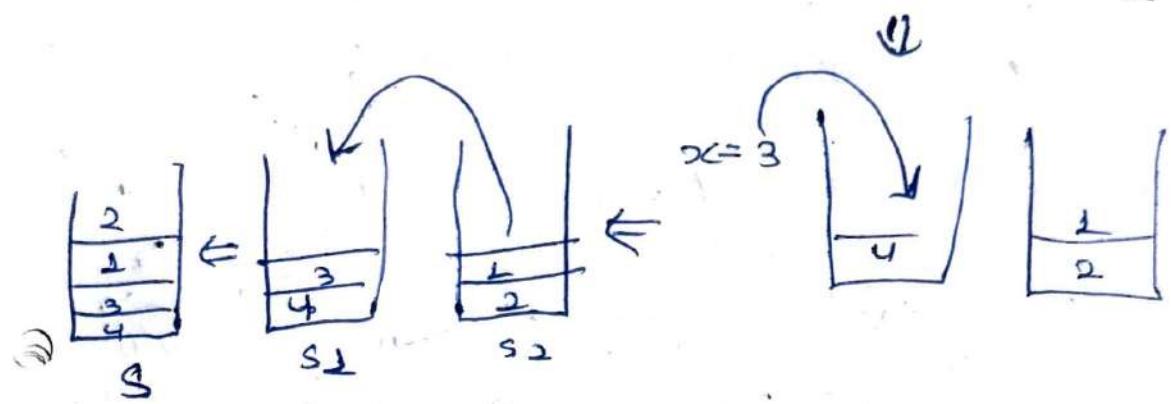
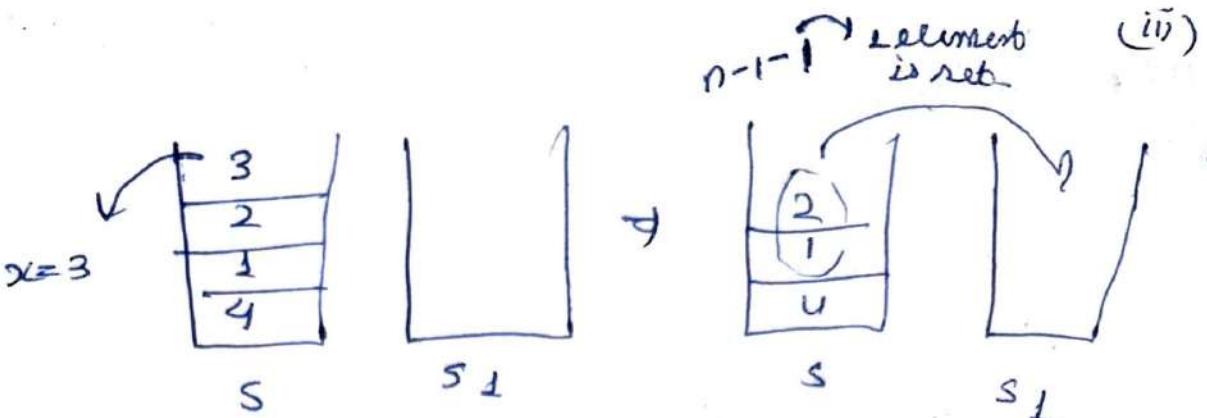
Reverse Stack

(i) Using 2 stack



(ii) Using 1 stack





* as so on.

reverseStack (\oplus stack<int> &ss)

```

stack<int> s2;
int n = ss.size();
for (int i = 0; i < n; i++) {
    int x = ss.top();
    ss.pop();
    transfer ( s1, s2, n - i - 1 );
    ss.push(x);
    transfer ( s2, s1, n - i - 1 );
}

```

4

(10)

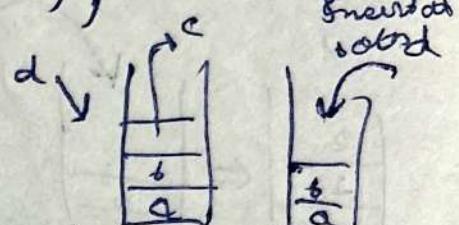
(11) Revision

8000

```
Alg0' reverse (stk) {
    if (empty) (do nothing)
```

```
else
    x = pop();
    reverse (C);
    minvalBottom (stk, x);
```

Insert at Bottom Alg0



Inserted at bottom

and then push c.

minvalBottom {

```
if (empty) push (d)
else
```

```
x = pop();
insertAtBottom
push (x);
```

, Base case.

Minimum in A Stack
in O(1) Time & O(1) Space

Insert 3

Insert 5

getMin

Insert 2

Insert 1

getMin

pop 1

Insertions

\rightarrow x is greater than or equal to current min
 \rightarrow insert x

\rightarrow x is less than current min
insert $x - \text{current_min}$

Problems : Stack - Histogram Area
Each area in
are to consider each bar as
min. height bar.

Stock Span Problem

→ Queue Recursive Reversal -

similar to stack recursive

```
void *Reverse(queue &Q) {
    int i;
    if (!isEmpty(Q)) {
        i = do Q.front();
        Q.pop();
        reverse(Q);
        insert(Q, i);  $\Rightarrow$  in stack is inserted bottom
         $\Downarrow$  means open as rear me hi insert hoga
    }
}
```

→ Sorting Using Queue and Stack.

- Let the elements to be sorted lie in a Queue Q & stack S empty.

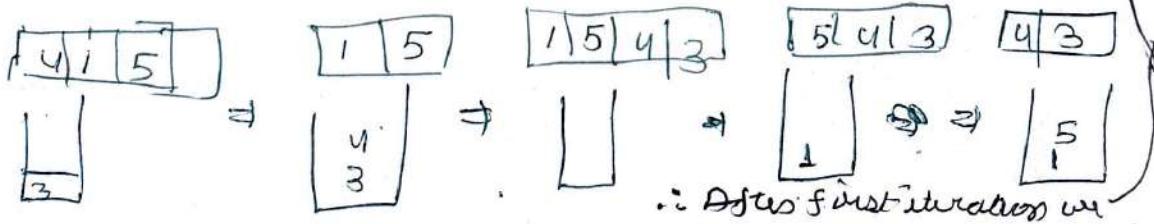
```
while Q is not empty do {
    if S is empty or top(S)  $\leq$  Head(Q) then
        1 x = Dequeue(Q)
        Push(S, x)
    } else {
        x = Pop(S)
        Enqueue(Q, x)
    }
}
```

worst case time complexity: $O(n^2)$

Ex: Q

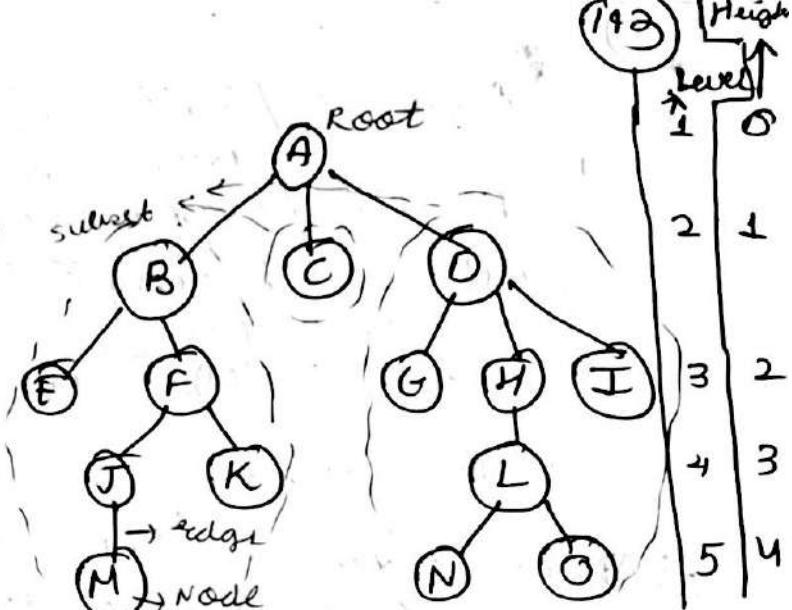
3	4	1	5
---	---	---	---

got min at the bottom of stack



Trees

Tree is collection of nodes where one node is taken as root node and rest of the nodes are divided into disjoint subset and each subset is a tree or subset tree.



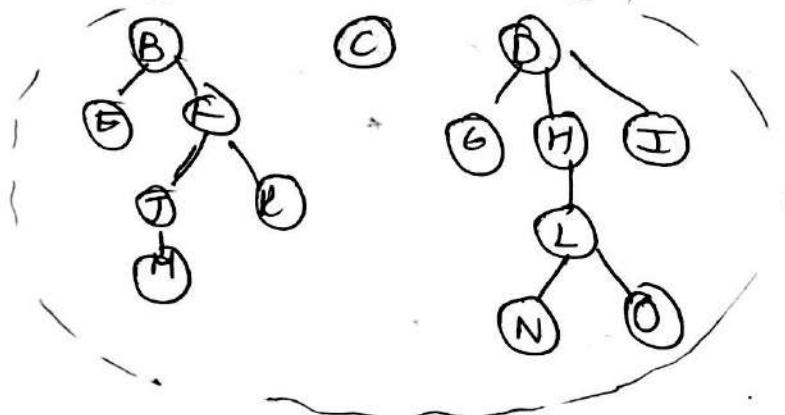
Terminology

- (i) Root: Very first Node at top of tree
- (ii) Parent: A node is Parent to its very next descendants i.e those children which are connected by one edge.
For ex: B, E & F are children & B is Parent of E & F.
- (iii) Children: Nodes having same Parent.
 $\textcircled{G}, \textcircled{H}, \textcircled{I}$ are Sibling
 $\textcircled{J} \& \textcircled{K}$
- (iv) Descendants: All those set of nodes which can be reached from a particular node under that node.
Ex: F, J, K, M are Descendants of B
- (v) Ancestors: From any node, from that to root node all nodes coming in its path are its Ancester.
Ex: N, L, H, D, A are ancestors
- (vi) Degree of Node: No of children it is having
L has 2 degree of Node
C has zero degree of Node

- (viii) Internal / External Nodes
- Non-terminal / Terminal Node
- Non-leaf / Leaf Node
- All Nodes having zero degree of node are external Nodes
ex : (E), (M), (A), (N), (O), (I), (G)
- Nodes whose degree $\neq 0$: Internal Nodes
- Levels: (examined) here we count nodes
we take the paths we taking no of nodes traversing which is equal to levels
→ ex: From Root to L Paths have 4 Nodes
(A), (D), (H), (L)

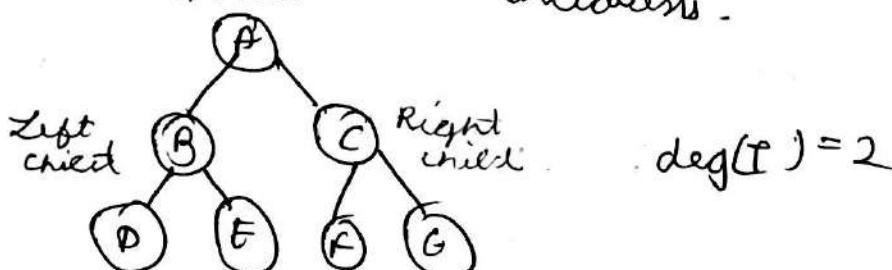
(ix) Height : (here we count edges)

(x) Forest : collection of trees is called Forest ↑



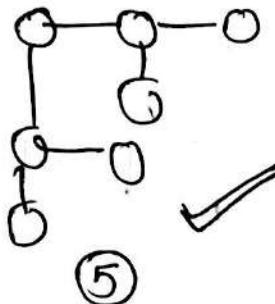
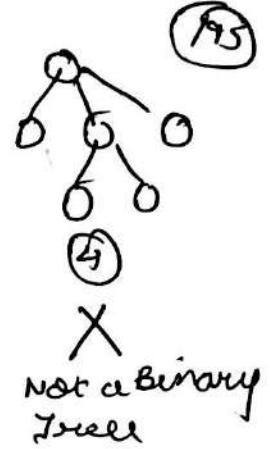
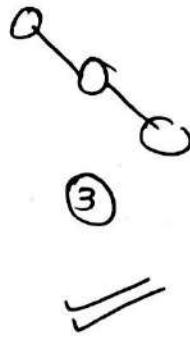
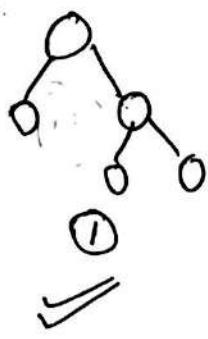
Binary Tree:

Degree = 2 or tree ie every Node can have max. 2 children.



children = {0, 1, 2}

ie Not more than 2.



Number of Binary Trees using N nodes

(i) Unlabelled Nodes

(ii) Labelled Nodes.

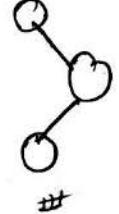
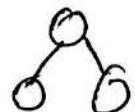
(I) Unlabelled Node

$$n=3$$

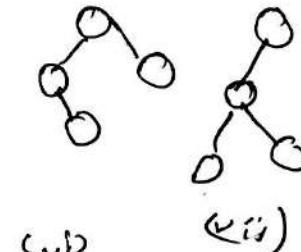
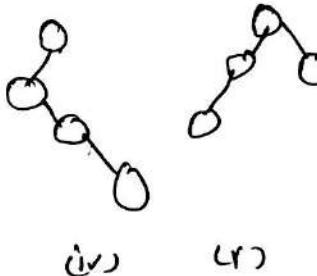
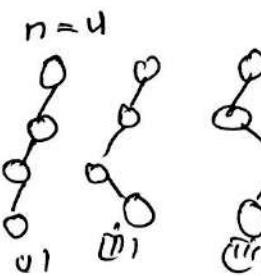
$$0 \quad 0 \quad 0$$



$$T(3) = 5$$



$$\begin{aligned} & \text{Binary trees having maximum height } = 4 = 2 \\ & n=3 \quad " \quad = 8 = 2^3 \\ & n=4 \quad " \quad = 2^7 \\ & n=5 \quad " \quad = 2^{15} \end{aligned}$$



(ii)

And we can make many more of these

\therefore Total 14

$$T(4) = 14$$

Catalan Numbers $\left[T(n) = \frac{^{2n}C_0}{n+1} \right]$ this is combination formula

$$T(5) = \frac{^{20}C_5}{6} = \frac{^{10}C_5}{6}$$

other Method:

n	0	1	2	3	4	5	6
$T(n) = \frac{^{2n}C_0}{n+1}$	1	1	2	5	14	42	

$$T(6) = T(0) \times T(5) + T(1) \times T(4) + T(2) \times T(3) + T(3) \times T(2) \\ + T(4) \times T(1) + T(5) \times T(0)$$

$$T(n) = \sum_{i=1}^n T(i-1) \times T(n-i)$$

this is recursive formula

Labeled Nodes

$$T(n) = \frac{^{2n}C_n}{n+1} * n!$$

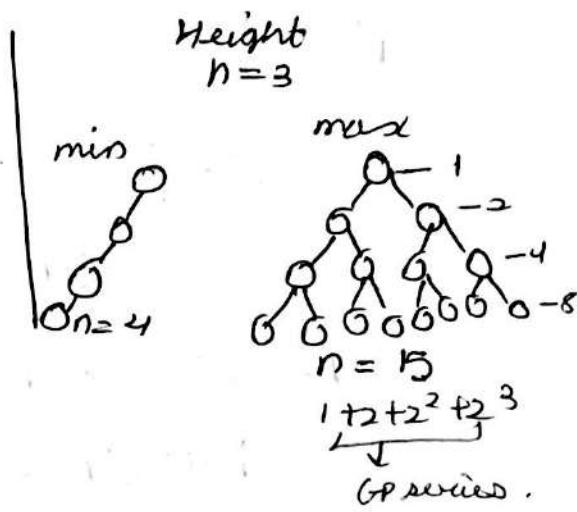
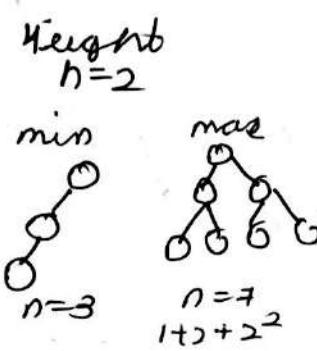
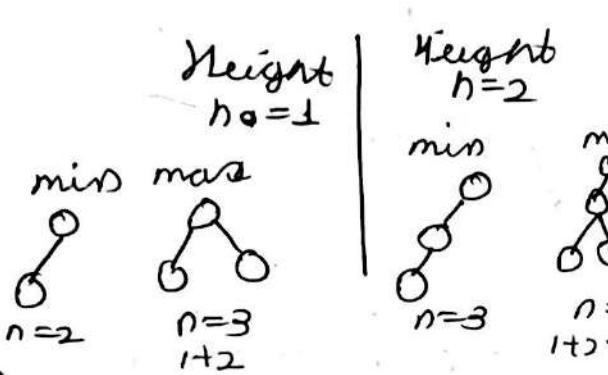
shape sitting

$n!$ ways to have permutations in any one shape of the tree

Formulas Height vs Nodes

145

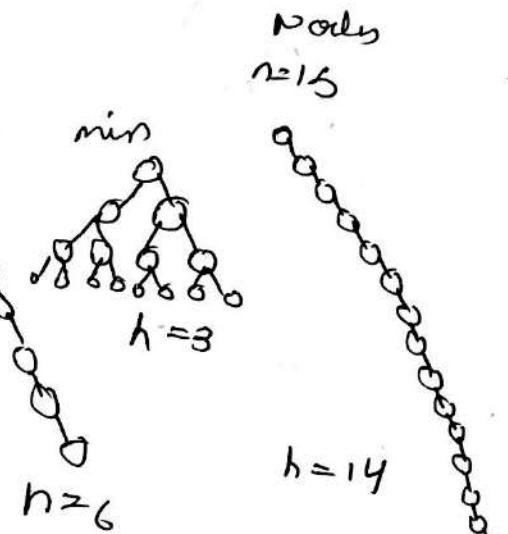
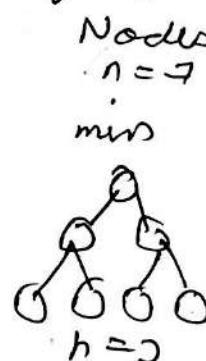
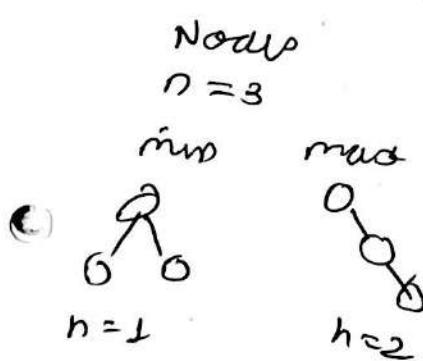
When Height is Given what can be Max & Min no of Nodes.



$$\text{Minimum Nodes } n = h + 1$$

$$\text{Maximum Node } n = 2^{h+1} - 1$$

When Nodes are given what will be max & min no of Heights



$$\text{max Height } h = n - 1$$

$$\therefore \text{Max Node } n = 2^{n+1} - 1$$

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

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

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

minimum Height

178

If Height is given

$$\text{Min Nodes } n = h + 1$$

$$\text{Max Nodes } n = 2^{h+1} - 1$$

If Nodes are given

$$\text{Min Height } \alpha h = \log_2(n+1) - 1$$

$$\text{Max Height } n = n - 1$$

Height of Binary Tree

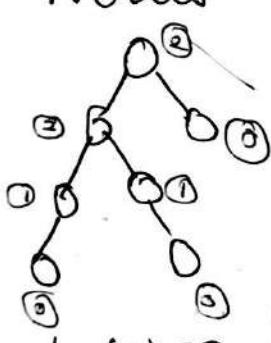
$$\log(n+1)-1 \leq h < n-1$$

$\overset{\text{O(log n)}}{\textcircled{O}}$ $\overset{\text{O(n)}}{\textcircled{O}}$

Number of Nodes in a Binary Tree

$$h+1 \leq n \leq 2^{h+1} - 1$$

Relationship b/w Internal & External Nodes

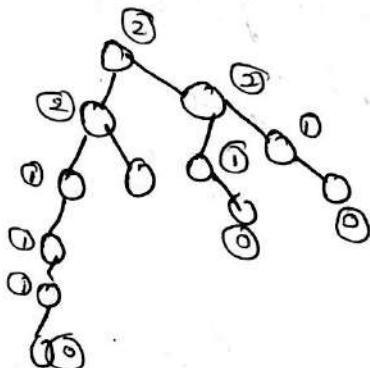


$$\deg(2) = 2$$

$$\deg(1) = 2$$

$$\deg(0) = 3$$

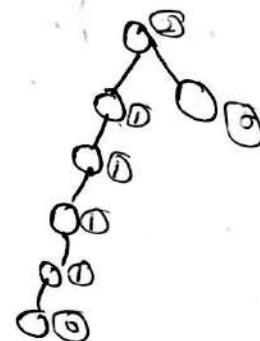
Relation:



$$\deg(2) = 3$$

$$\deg(1) = 5$$

$$\deg(0) = 4$$



$$\deg(3) = 1$$

$$\deg(1) = 4$$

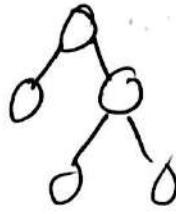
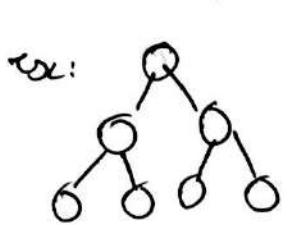
$$\deg(0) = 2$$

$$[\deg(v) = \deg(u) + 1]$$

Always true in Binary tree.

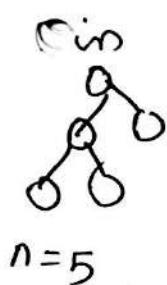
Strict Binary Tree

degree (0, 2) not 1

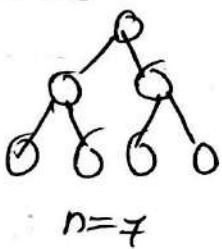


Height vs Node of Strict Binary

$$h=2$$



max



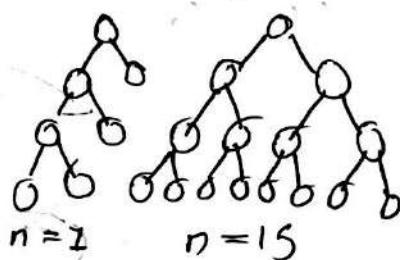
$$n=5$$

$$n=7$$

$$h=3$$

min

max

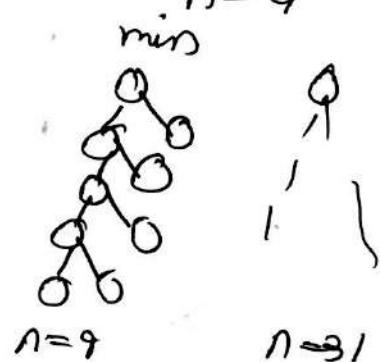


$$n=3$$

$$n=15$$

$$h=4$$

min



$$n=9$$

$$n=31$$

if Height 'h' is given

$$\text{Min Nodes} \Rightarrow n = 2^h + 1$$

$$\text{Max Nodes} \Rightarrow n = 2^{h+1} - 1$$

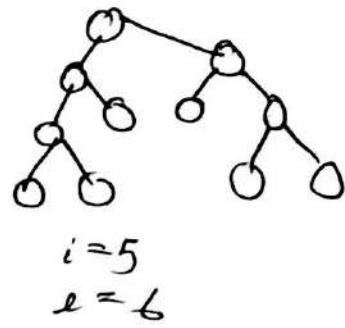
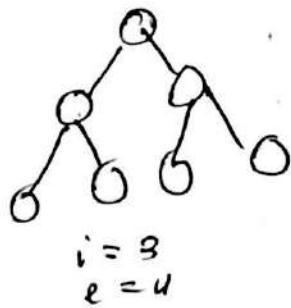
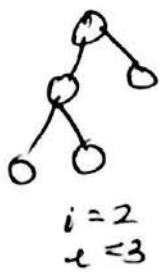
From Min Nodes formula we will get Max Height

$$\text{Max Height} \Rightarrow h = \frac{n-1}{2}$$

$$\text{Min Height} \Rightarrow \log_2(n+1) = h+1$$

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

② Internal Node VS External nodes

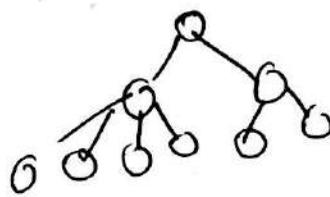
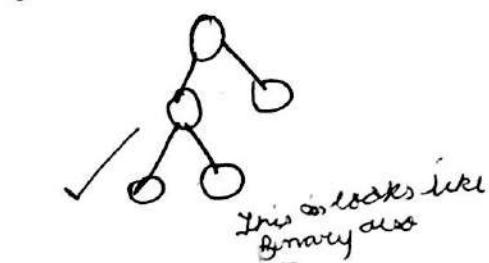
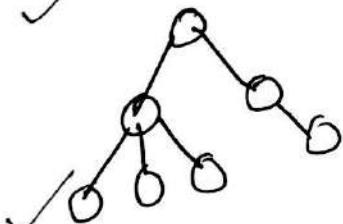
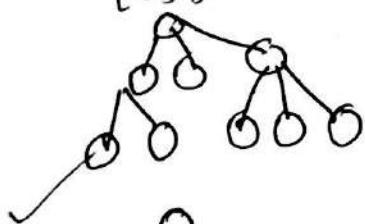


$$\therefore \boxed{e = i + 1}$$

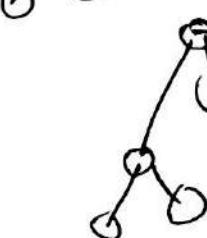
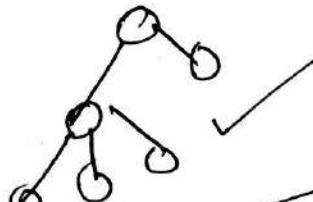
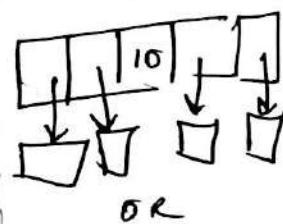
n -ary Tree

4-way tree $\{0, 1, 2, 3, 4\}$

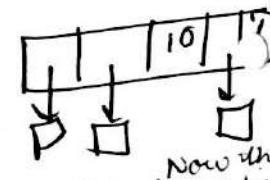
3-way Tree
 $\{0, 1, 2, 3\}$



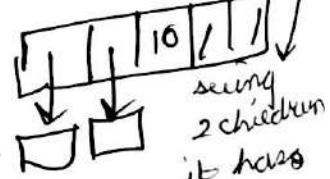
If we are making 4-way tree



X



Now this is not binary tree by just



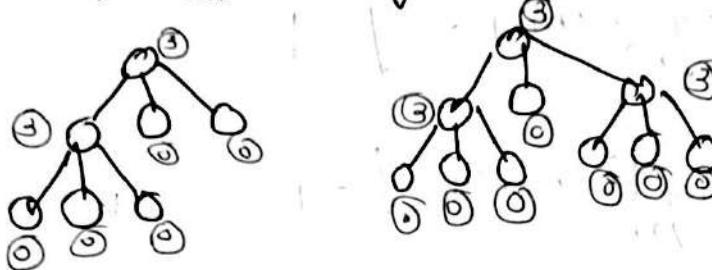
seeing 2 children it has capacity of 4 in which 2 are null.

Root tree does not decide degree indeed degree of tree is predecided.

(2-1)

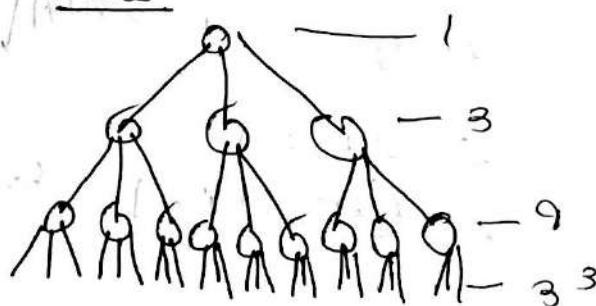
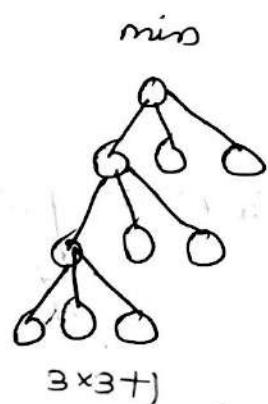
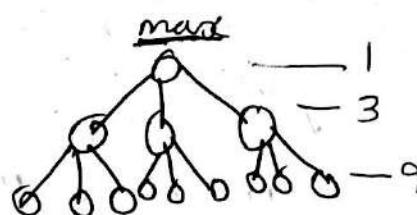
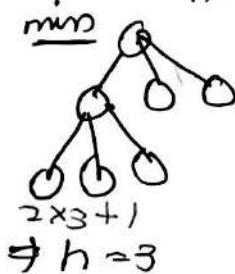
Strict n -ary Tree
degree can be 0 or n only : $\{0, n\}$

strict 3-ary tree



Analysis of m -Ary Tree

strict m -ary tree
 $h=2$



$$\therefore 1 + 3 + 3^2 + 3^3 + \dots + 3^{h-1} = \frac{(3^h - 1)}{2}$$

if height is given,

$$\text{Min nodes } n = mh + 1$$

$$\text{Max Nodes } n = \frac{m^{h+1} - 1}{m - 1}$$

if n^1 nodes are given

max height can be evaluated from min node

$$n = mh + 1$$

$$\Rightarrow \text{max height } h = \frac{n-1}{m}$$

202

For min height

$$n(m-1) = m^{h+1} - 1$$

$$m^{h+1} = [n(m-1) + 1]$$

$$h = \log_m [n(m-1) + 1] - 1$$

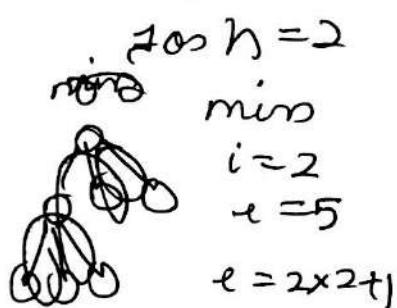
 \therefore if n nodes are given

$\boxed{\text{min height } h = \log_m [n(m-1) + 1] - 1}$

$\text{Max Height } \Rightarrow h = \frac{n-1}{m}$

Internal & External Nodes

in strict 3-ary tree



max
 $i = 4$
 $e = 9$
 $e = 2 \times 4 + 1$

min
 $i = 3$
 $e = 7$
 $e = 2 \times 3 + 1$

max
 $i = 13$
 $e = 27$
 $27 = 2 \times 13 + 1$

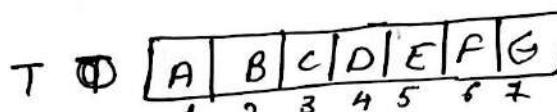
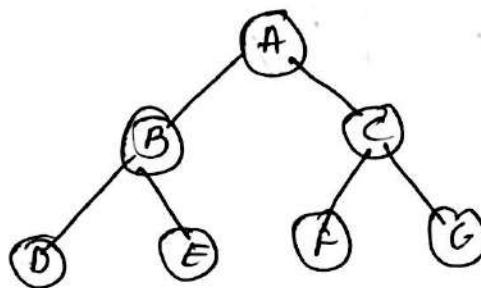
$e = 2^i + 1$
 $[e = (m-1)i + 1]$ for Strict n -ary Tree

Representation of Binary Tree

- 1) Array Representation
- 2) Linked List Representation

For array Representation

We need to store elements as well as their relationship
To preserve relationship we will use index of Array



space is left blank

element	index	Lchild	Rchild
A	1	2	3
B	2	4	5
C	3	6	7
	i	$2 \cdot i$	$2 \cdot i + 1$

element - i

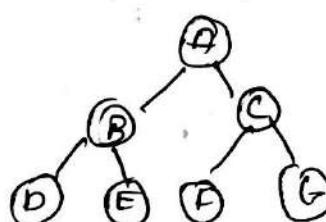
$$\text{Lchild} = 2 \cdot i$$

$$\text{Rchild} = 2 \cdot i + 1$$

$$\text{Parent} = \left\lfloor \frac{i}{2} \right\rfloor \text{ ie } \left\lfloor \frac{3}{2} \right\rfloor = 1$$

Linked Representation

Node



struct Node

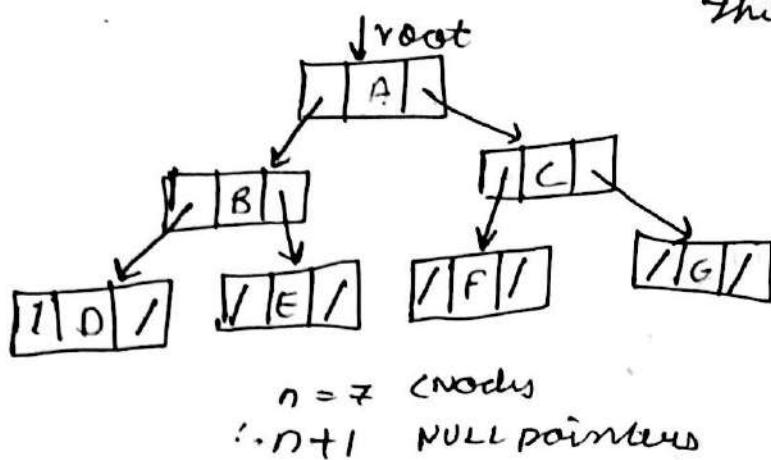
{ struct Node *lchild; }

int data;

struct Node *rchild;

}

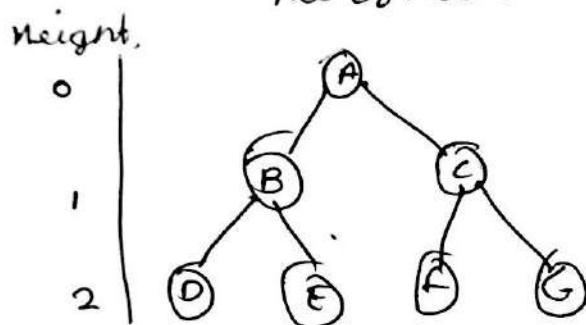
2Dy



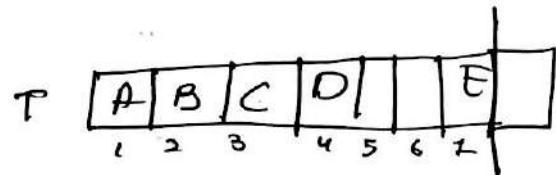
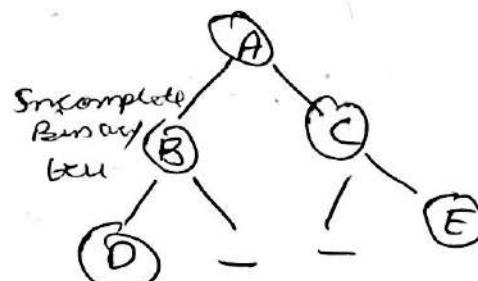
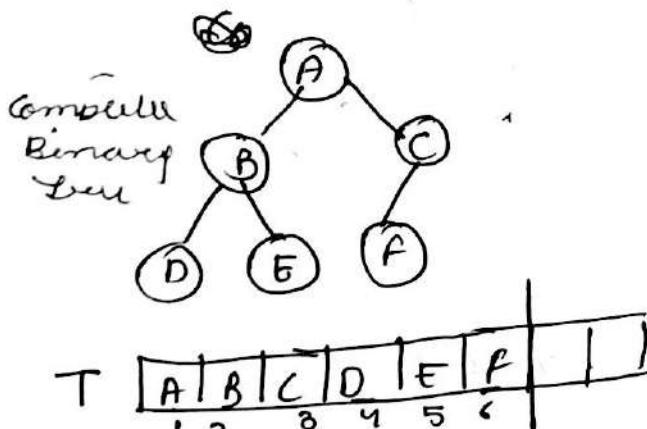
This structure is dynamic.

Full vs Complete Binary Tree

⇒ Full Binary Tree: A tree having 2^h nodes in level h with no NULL pointers.

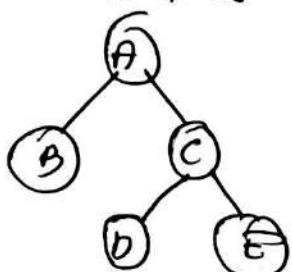


$$n=2^h \cdot 2^{h+1} - 1 = 2^{2h+1} - 1 = 2^3 - 1 = 7$$



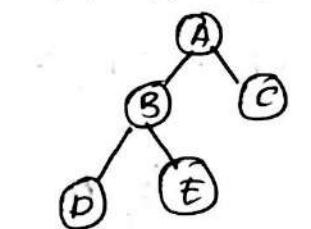
A Full Binary Tree is always a Complete Binary Tree but vice versa is not true.

Strict vs Complete
↓
complete



T	A	B	C	-	-	D	E
	1	2	3	4	5	6	7

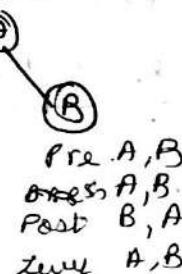
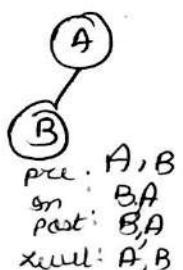
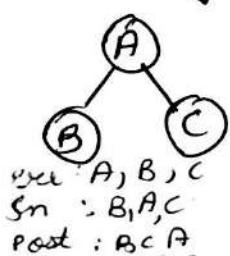
✓ Strict
✗ Complete



T	A	B	C	D	E	-
	1	2	3	4	5	6

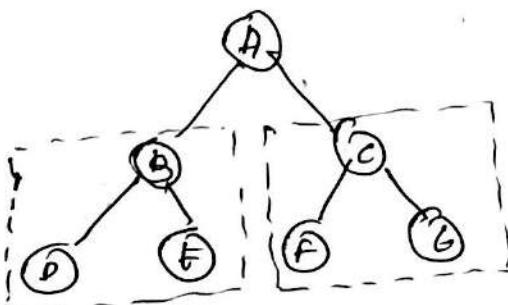
✓ Strict
✓ Complete

Binary Tree Traversals.



Different Methods:

- (i) Preorder: visit(node), Preorder(left subtree), Preorder(right subtree)
- (ii) Inorder: inorder(left), visit(node), inorder(right)
- (iii) Postorder: Postorder(left), Postorder(right), visit(node)
- (iv) Level order: Level by Level.



Bee : A, C preorder, subtree I C preorder, subtree II
 $\Rightarrow A, (B, D, E), (C, F, G)$
 $\Rightarrow \underline{A, B, D, E, C, F, G}$

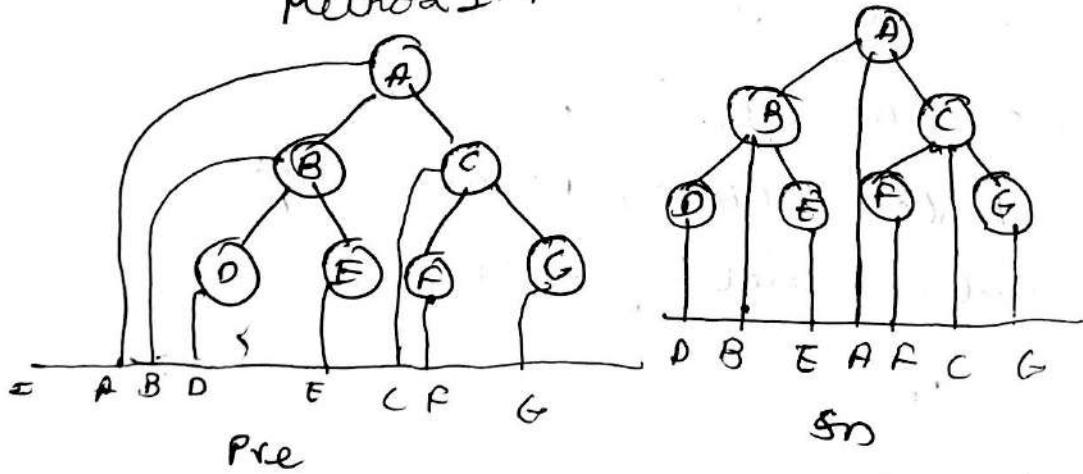
Sn : $(), A, ()$
 $\Rightarrow (D, B, E), A, (F, C, G)$
 $\Rightarrow \underline{D, B, E, A, F, C, G}$

Post : $(), (), A$
 $\Rightarrow (D, E, B), (F, G, C) \rightarrow A$
 $\Rightarrow \underline{D, E, B, F, G, C, A}$

Level : $\underline{\underline{A, B, C, D, E, F, G}}$

Binary Tree Traversal Easy Method

Method I :

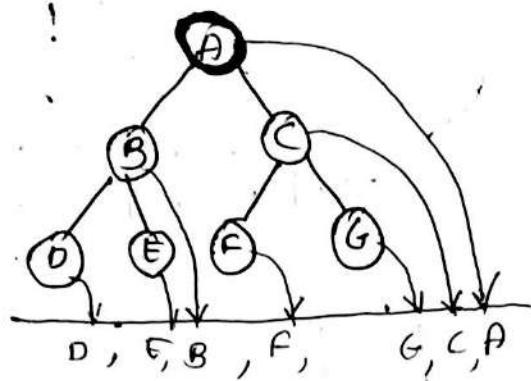


For left hand side of every node we have to draw lines to connect I. lines cannot be intersecting.

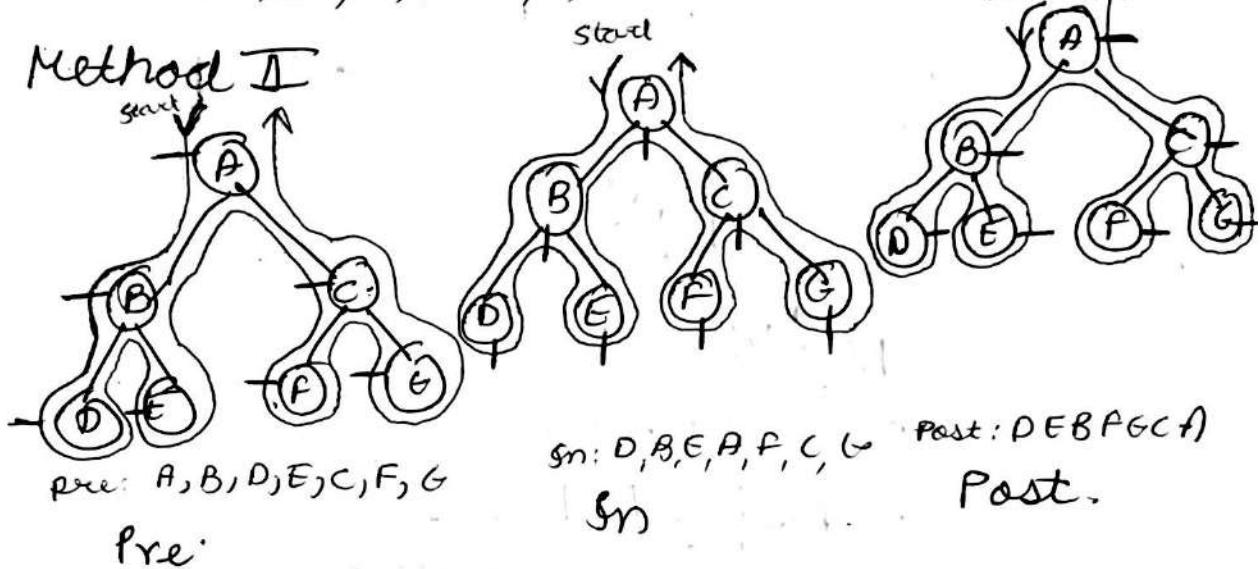
For every Node join one by making line b/w child.

221

207



Method I



Pre.

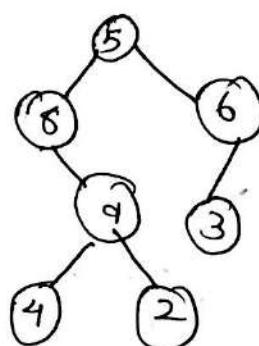
(On this move along the tree)

Method II: (Final)

Finger Painting Method: Pre In Post

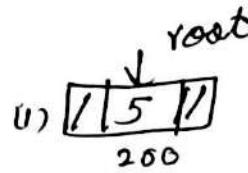
Creating Binary Tree

By Level by Level using Linked List
we need a Queue.

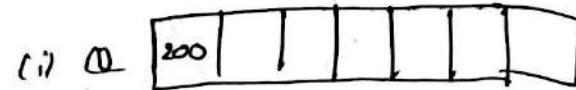


205
206

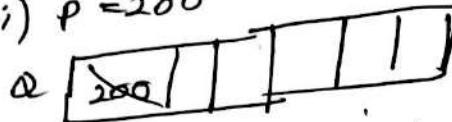
Step 1



push 200

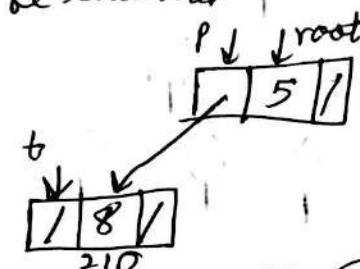


(iii) $p = 200$

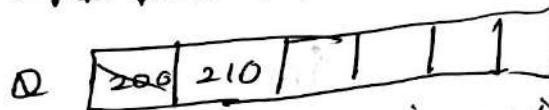


Now check p left child if ~~was~~ not there -1 will be returned

Step 2

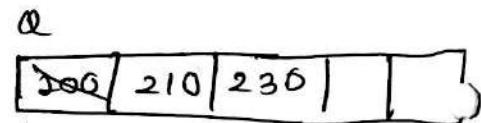
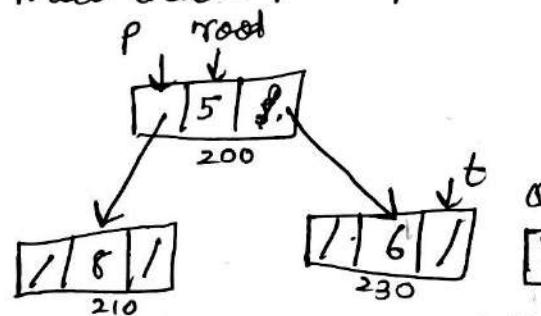


Done check p's left



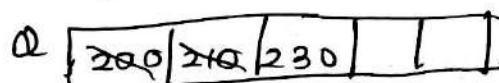
Now check p right child.

Step 3



Now dequeue from Q

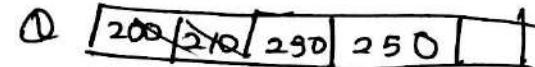
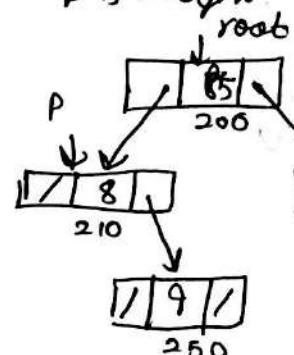
$$\therefore p = 210$$



Step 4

p's left :: not there $\therefore -1$

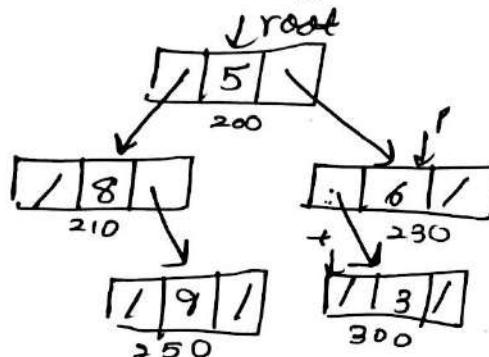
p's right



Step 5: Now \diamond dequeue from Q & store in p

$$\therefore p = 230$$

Now p' 's left.



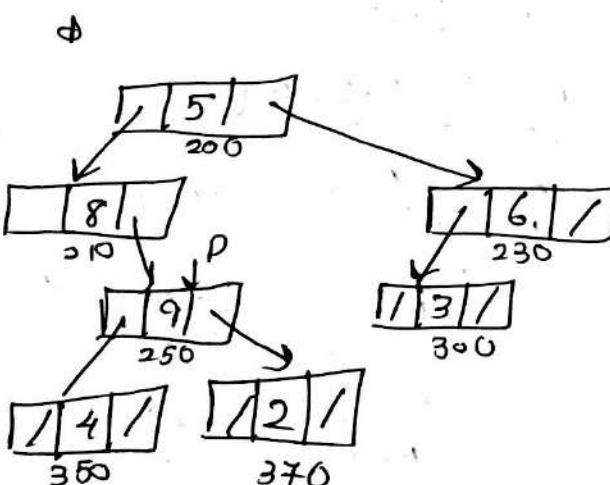
α [200 | 210 | 230 | 250 | 300]

Now p' 's left or right child not there

Step 6: dequeue \diamond

$$p = 250$$

$\therefore p'$'s left & p' 's right.



α [200 | 210 | 230 | 250 | 300 | 350 | 370]

Step 7: Now dequeue from Q
 $p = 300$ No left & right child.

Step 8: dequeue from Q

$p = 350$ No left & right child.

Step 9: dequeue from Q

$p = 370$ No left & right child

Finally we created a tree using linked.

Binary Tree Create.

```

#include <stdio.h>
#include <stdlib.h>
#include <Queue.h>

struct Node *root = NULL;

void TreeCreate()
{
    struct Node *P, *t; // t is for assigning new node
    int x; // x is for connecting one current node with t.
    struct Queue q; // create (&q, 100);
    printf("enter root value");
    scanf("%d", &x);
    root = (struct Node *)malloc(sizeof(struct Node));
    root->data = x;
    root->lchild = root->rchild = NULL;
    enqueue(&q, root);
    while (!isEmpty(q))
    {
        P = dequeue(&q);
        printf("enter left child of %d", P->data);
        scanf("%d", &x);
        if (x != -1)
        {
            t = (struct Node *)malloc(sizeof(struct Node));
            t->data = x;
            t->lchild = t->rchild = NULL;
            P->lchild = t;
            enqueue(&q, t);
        }
        printf("enter right child of %d", P->data);
        scanf("%d", &x);
        if (x != -1)
        {
            t = (struct Node *)malloc(sizeof(struct Node));
            t->data = x;
            t->lchild = t->rchild = NULL;
            P->rchild = t;
            enqueue(&q, t);
        }
    }
}

```

```

 $t \rightarrow data = x$ 
 $t \rightarrow lchild = t \rightarrow rchild = NULL$ 
 $P \rightarrow rchild = t;$ 
enqueue(&Q, t);
}

↓
void Preorder (struct Node *P)
{
    if (P)
    {
        printf ("%d", P->data);
        Preorder (P->lchild);
        Preorder (P->rchild);
    }
}

↓
void Inorder (struct Node *P)
{
    if (P)
    {
        Inorder (P->lchild);
        printf ("%d", P->data);
        Inorder (P->rchild);
    }
}

↓
void Postorder (struct Node *P)
{
    if (P)
    {
        Postorder (P->lchild);
        Postorder (P->rchild);
        printf ("%d", P->data);
    }
}

```

```

int main()
{
    Tree create();
    Preorder( root );
    printf ("In PostOrder");
    Postorder( root );
    return 0;
}

```

Queue Header File -

```

struct Node
{
    struct Node *lchild;
    int data;
    struct Node *rchild
};

struct Queue
{
    int size;
    int front;
    int rear;
    struct Node **Q;
};

void create (struct Queue *q, int size)
{
    q->size = size;
    q->front = q->rear = 0;
    q->Q = (struct Node **) malloc(q->size *
        sizeof(struct Node *));
}

void enqueue (struct Queue *q, struct Node *x)
{
}

```

213
+ insert (int arr[], int start, int end, int value)

```
if ((q->rear + 1) % q->size == q->front)
    printf ("Queue is Full");
else
    q->rear = (q->rear + 1) % q->size;
    q->Q[q->rear] = x;
```

{
y

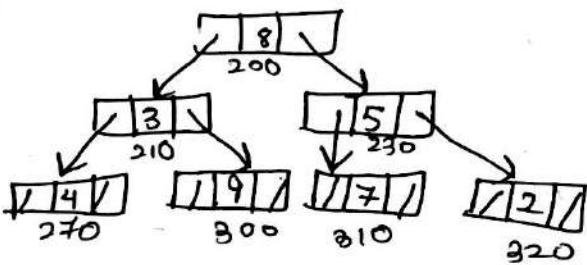
struct Node *dequeue(struct Queue *q),

```
struct Node *x = NULL;
if (q->front == q->rear)
    printf ("Queue is Empty\n");
else
    q->front = (q->front + 1) % q->size;
    x = q->Q[q->front];
return x;
```

f

```
int isEmpty (struct Queue q)
{
    return q.front == q.rear;
}
```

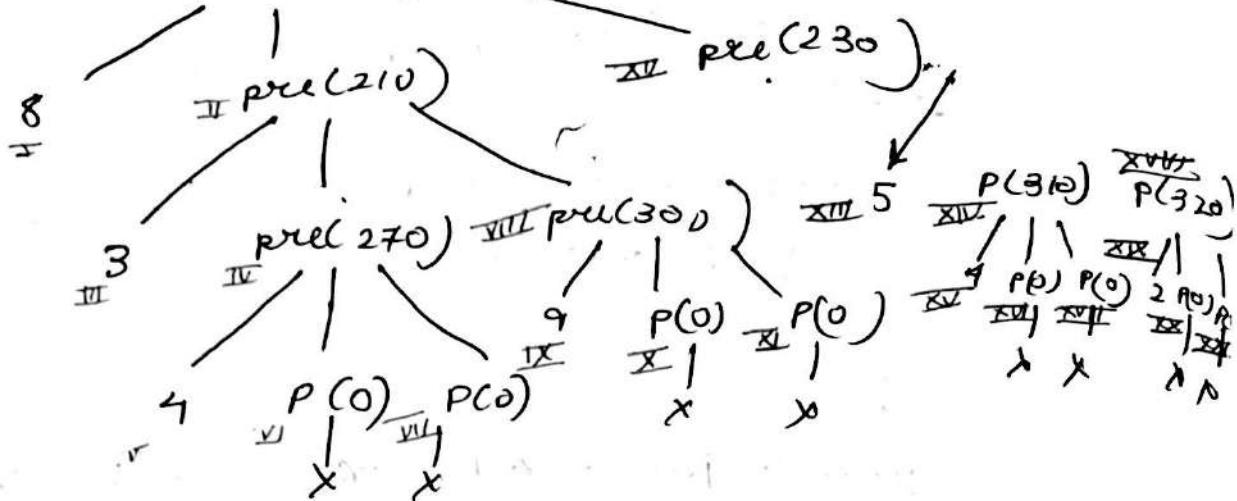
→ Tree Traversal



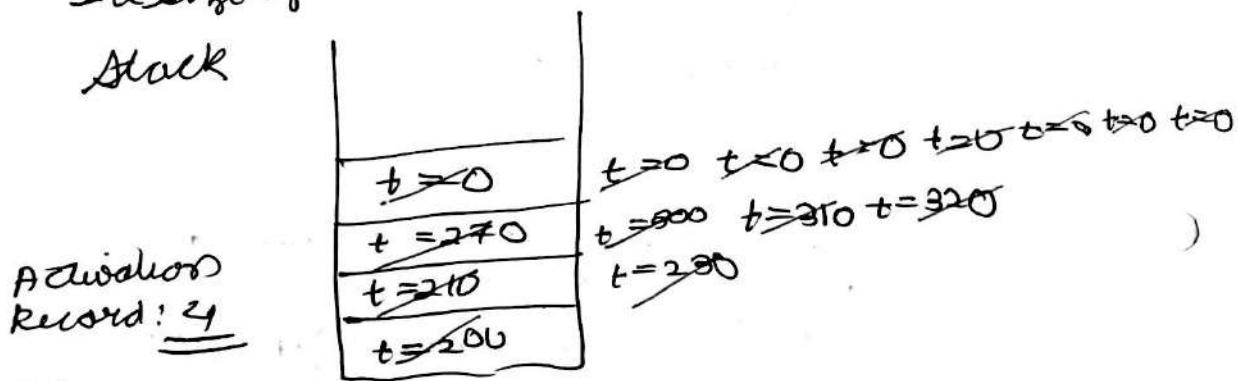
Preorder (root)

```
void Preorder (Node *t)
{
    if (t != NULL)
    {
        printf ("%d ", t->data);
        Preorder (t->left);
        Preorder (t->right);
        Preorder (t->right);
    }
}
```

214

Tracing
pre(200)output: 8, 3, 4, 9, 5, 7, 2 (as n+1 null points)Total no. of calls : $n + \frac{n+1}{2} = \underline{2n+1 \text{ calls}}$
from nodes

The size of stack will be height of tree + 2



ii

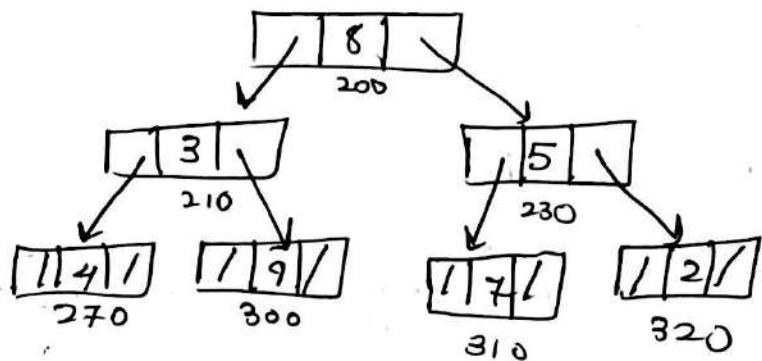
 $n=2$ $\therefore 4 \oplus 2+2$

→ Inorder Traversal

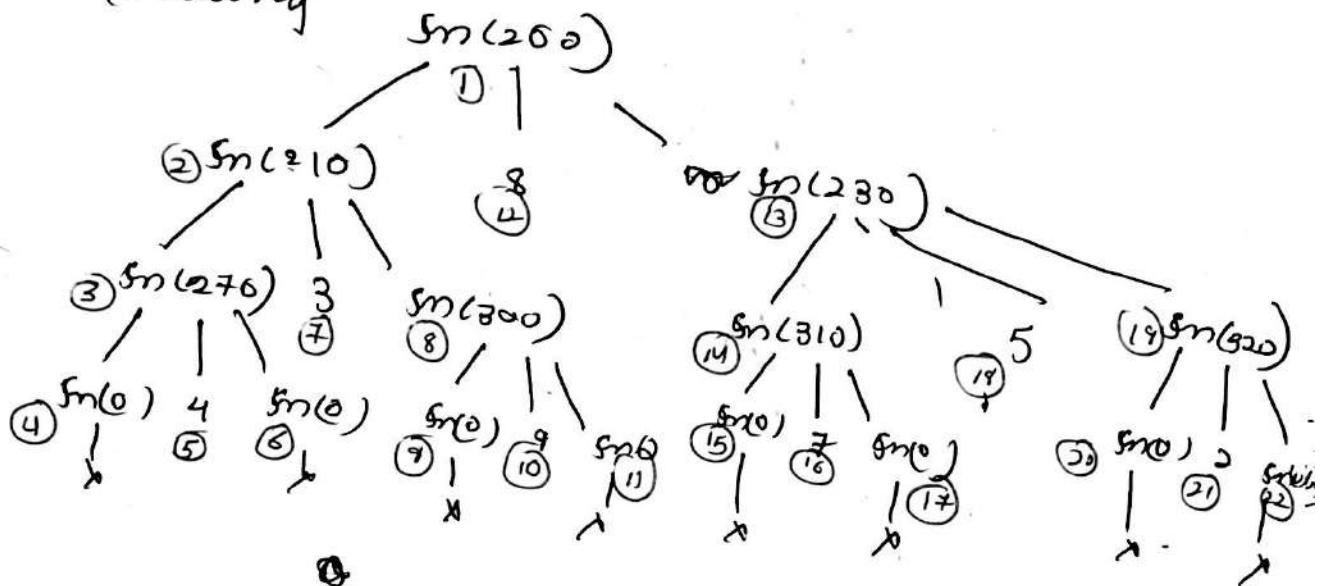
void Inorder(Node *b)

1 if ($t \neq \text{NULL}$)1. Inorder($t \rightarrow \text{left}$)2 printf("%d", $t \rightarrow \text{data}$)3 Inorder($t \rightarrow \text{right}$)1 Inorder(root)

int search(int arr[], int start, int end, int value)



Tracing



Output : 4, 3, 9, 8, 7, 5, 2

No. of call: $2n+1$

In one call it is just printing.

∴ Time Complexity: $2n+1: O(n)$

Iterative Traversal

⇒ In post order

We have to push the same address two times. One time for going to right child and second time for printing.

216

Pre order

```

void Preorder(Node *t)
{
    struct Stack st;
    while (t != NULL || !isEmpty(st))
    {
        if (t == NULL)
        {
            cout << " " << t->data;
            push(&st, t);
            t = t->leftchild;
        }
        else
        {
            t = pop(&st);
            cout << " " << t->data;
            t = t->rightchild;
        }
    }
}

```

Note:
while $t == \text{NULL}$ stack is empty
tell the continue
karnab.
some traversal
me

```

isLchild()
print
isRchild()

```

to convert to inorder (2) is placed
in preorder at position (6)

```
void Inorder(Node *t)
```

```
{
    struct Stack st;
    while (t != NULL || !isEmpty(st))
```

```
{
    if (t == NULL)
```

```

    {
        push(&st, t);
        t = t->leftchild;
    }
}
```

```

else
{
    t = pop(&st);
    cout << " " << t->data;
    t = t->rightchild;
}
```

}

,

Scanned with CamScanner

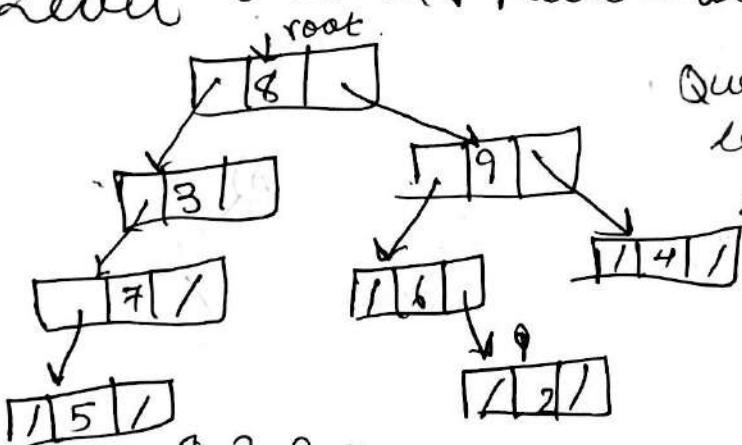
Post order:

```
void Postorder(Node *t)
```

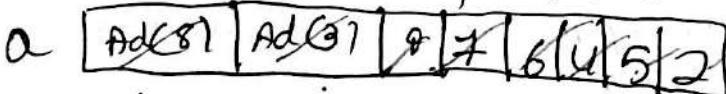
```

    struct Stack st;
    long int temp;
    while(t!=NULL || !isempty(st))
    {
        if(t!=NULL)
            {
                push(&st,t);
                t=t->lchild;
            }
        else
            {
                temp=pop(&st);
                if(temp>0) "This will tell that we have to go to right child"
                push(&st,temp);
                t=((Node*)temp)->rchild;
            }
        } else "this will print after getting from right child."
        {
            printing("d",((Node*)temp)->data);
            t=NULL;
        }
    }
}
```

Level Order Traversal



OP: 8, 3, 9, 7, 6, 4, 2



Same as creating queue tree

Queue helps us to go level by level

& stack helps us to go Pre, in & post order

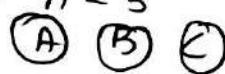
```

void levelorder(Node *p)
{
    queue q;
    printf("d", p->data);
    enqueue(&q, p);
    while (!isEmpty(q))
    {
        p = dequeue(&q);
        if (p->lchild)
            printf("l.d", p->lchild->data);
        enqueue(&q, p->lchild);
        if (p->rchild)
            printf("r.d", p->rchild->data);
        enqueue(&q, p->rchild);
    }
}

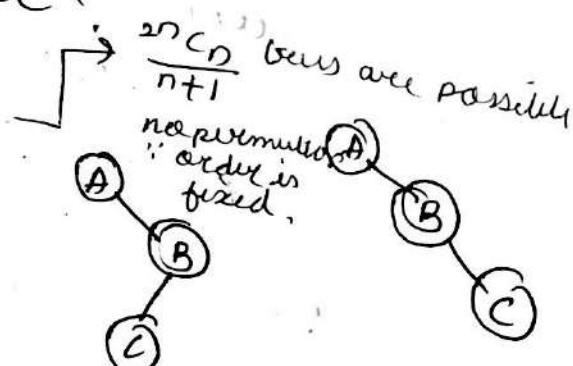
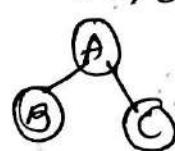
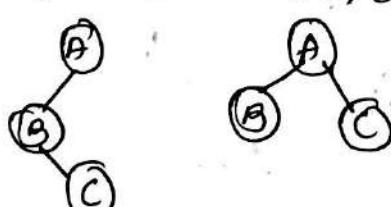
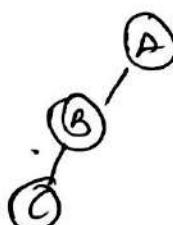
```

p
 First printing
 then enqueue
 then dequeue it & go to left child & right child
 Can we generate Tree from
 Traversal?

ex: $n = 3$



preorder: A, B, C



$\therefore 5$ ways are possibly

\therefore If only pre-order is given we cannot create tree.

If we take preord: ABC &
postord - CBA

Then also more than 1 tree.

∴ $\begin{array}{l} \text{① Preorder} \\ \Rightarrow \text{② Postorder} \\ \text{③ Inorder} \end{array}$] $\frac{2^n}{n+1}$ trees] ~~more than one.~~

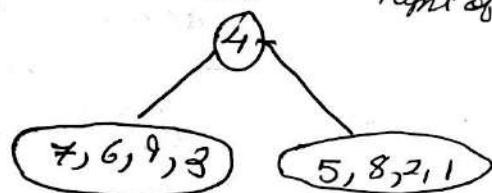
For unique tree
 ① preord + Inorder (Because it has root which will be in between)
 ② postord + Inorder

Generating Tree from Traversal

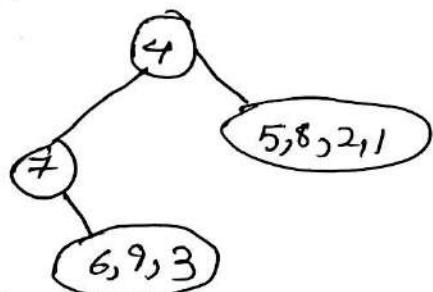
Preorder - 4, 7, 9, 6, 3, 2, 5, 8, 1
 Inorder - 7, 6, 9, 3, 4, 5, 8, 2, 1

(i) $P=4$ (scanning through Preorder from left to right)

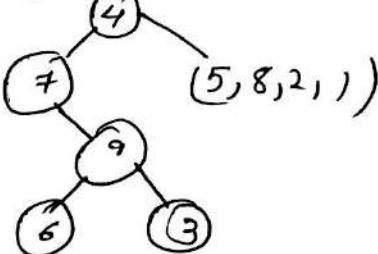
left of 4 right of 4



(ii) $P=7$



(iii) $P=9$



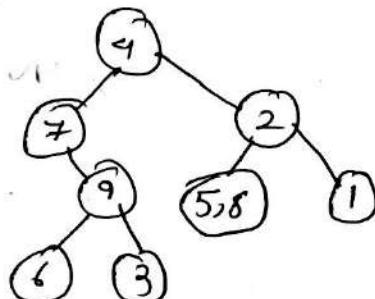
(iv) $P=6$

or since only one node

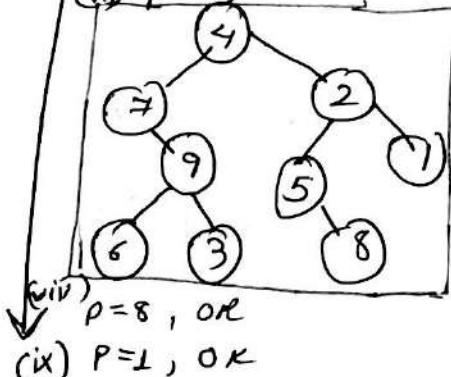
(v) $P=3$

or since only one node.

(vi) $P=2$



(vii) $P=5$



(viii) $P=8$, or
 (ix) $P=1$, OK

220

Time complexity: $n \cdot$ elements are traversed
 $\& n$ time for search each element in inorder

$$O(n) = n \times n = \underline{\underline{n^2}}$$

Algorithm: buildTree()

- 1) Pick an element from Preorder. Increment a Preorder Index variable to pick next element in next recursive call.
- 2) Create a new tree node tNode with the data as picked element.
- 3) Find the picked element's index in inorder. Let the index be inIndex.
- 4) Call buildTree for elements before inIndex and make the built tree as left subtree of tNode.
- 5) Call buildTree for elements ~~before~~^{above} inIndex and make the built tree as right subtree of tNode.
- 6) Returns tNode.

```
#include <iostream>
using namespace std;
```

```
class node
{
public:
    int data;
    node *left;
    node *right;
};
```

```

int search (int arr[], int start, int end, int value)
{
    int i;
    for (i = start; i <= end; i++)
        if (arr[i] == value)
            return i;
}

node * newNode (int data)
{
    node * Node = new node();
    Node->data = data;
    Node->left = NULL;
    Node->right = NULL;
    return (Node);
}

node * buildTree (int ins[], int pre[], int insStart, int insEnd,
                 int preIndex = 0) {
    static int preIndex = 0; node * node = newNode (pre[preIndex]); preIndex++;
    if (insStart > insEnd)
        return NULL;
    // If this node had no children
    if (insStart == insEnd)
        return tNode;
    int insIndex = search (ins, insStart, insEnd, tNode->data);
    int insIndex = search (ins, insStart, insEnd, tNode->data);
    // Using index in inorder, construct left and
    // right tree
    tNode->left = buildTree (ins, pre, insStart, insIndex - 1, preIndex);
    tNode->right = buildTree (ins, pre, insIndex + 1, insEnd, preIndex);
    return tNode;
}

```

Height & Count of Binary Tree

int count(Node *P)

{ int x, y;
if (P!=NULL)

{ x = count(P->lchild);
y = count(P->rchild);
return x+y+1;

y
return 0;

operators
are post
order.

This function is working in post order form
For doing any type of process in Binary tree
mostly post form is done or pre order
form

int count(Node *P)

{ int x, y;

if (P!=NULL)

{ x = count(P->lchild);

y = count(P->rchild);

if (P->lchild && P->rchild)

return x+y+1

else

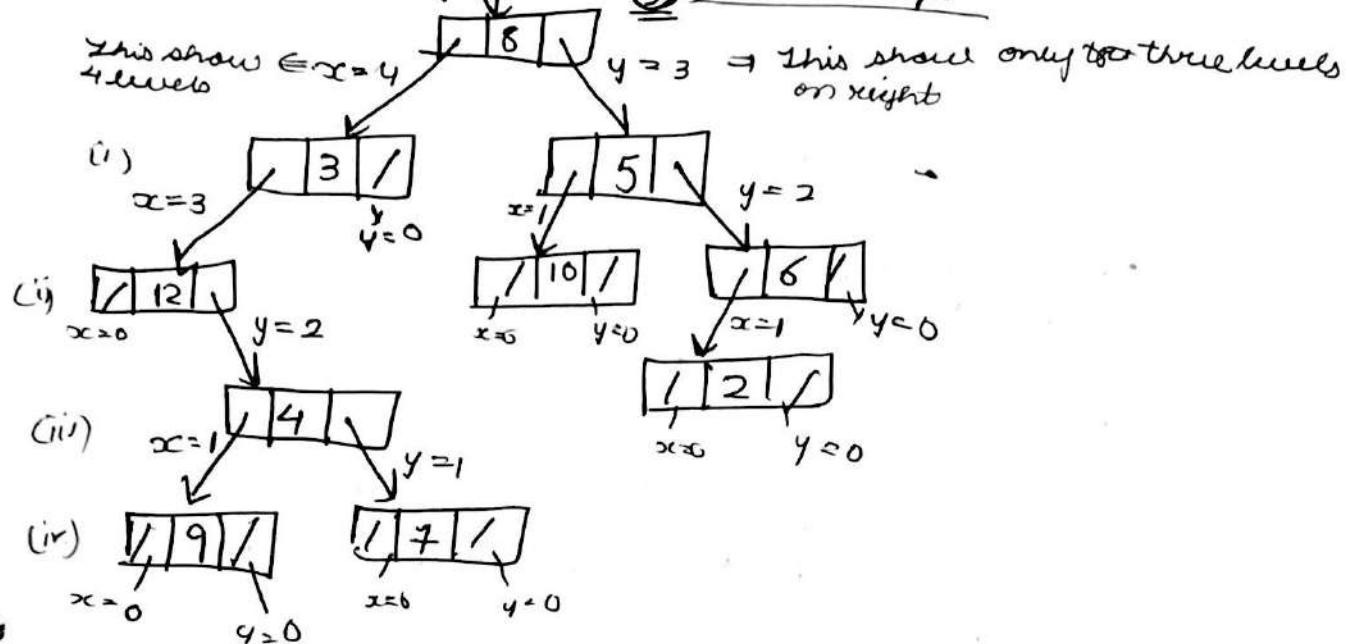
return x+y;

return 0;

y
return 0;

This is count no. of nodes having degree 2

Traversing of Height



* int height (struct Node *root)

```

int x=0, y=0;
if (root==0)
    return 0;
x = height (root->lchild);
y = height (root->rchild);
if (x>y)
    return x+1;
else
    return y+1;
    
```

Count of Leaf Nodes

* int count (struct Node *p)

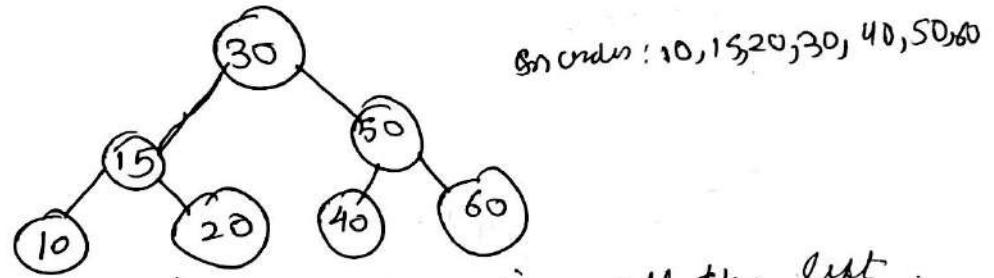
```

int x, y;
if (p==NULL)
    x = count (p->lchild);
    y = count (p->rchild);
    if (p->lchild == NULL & p->rchild == NULL)
        return x+y+1;
    else
        return x+y;
    return 0;
    
```

1. Node deg 0
⇒ if ($p \rightarrow lchild \& p \rightarrow rchild$)
2. Node deg 2
⇒ if ($p \rightarrow lchild \& p \rightarrow rchild$)
3. deg. 1 are 2
⇒ if ($p \rightarrow lchild \& p \rightarrow rchild$)
4. deg 1
⇒ if (($p \rightarrow lchild = \text{NULL} \& p \rightarrow rchild = \text{NULL}$)
 \cup ($p \rightarrow lchild = \text{NULL} \& p \rightarrow rchild \neq \text{NULL}$))
or it can be written as
⇒ if ($p \rightarrow lchild \neq \text{NULL} \wedge p \rightarrow rchild \neq \text{NULL}$)

Binary Search Tree

What is BST



It is Binary tree in which all the left side node is less than root and right side is greater

Properties

- (i) No duplicates in BST
- (ii) Inorder gives Sorted Order
- No. of BST with n nodes $T(n) = \frac{2n}{n+1} C_n$

It is ~~wood~~ represented by linked,

Node * Research (Node * t, int key)

```

{ if (t == NULL)
    returns NULL;
  if (key == t->data)
    returns t;
  else if (key < t->data)
    returns Research(t->lchild, key);
  else
    returns Research(t->rchild, key)
}
  
```

Iterative Version

Node * Search (Node * t, int key)

```

{ while (t != NULL)
  { if (key == t->data)
      returns t;
    else if (key < t->data)
      t = t->lchild;
    else
      t = t->rchild;
  }
  returns NULL;
}
  
```

NOTE tail recursion is converted into loop we definitely
don't require stack

226

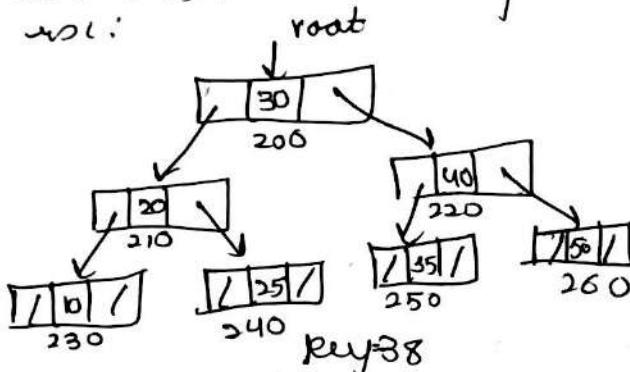
262 Inserting in Binary Search Tree

using tail pointer (r)

initial $r = \text{NULL}$ & $t = \text{root}$
then r will follow t .

when t will become NULL this means key not found.
 \therefore at that time r will be pointing ~~on~~ on node
at which t was pointing earlier.

sol:



steps :

$r = \text{NULL}$	$t = 200$	$\because 30 < 38$
$r = 200$	$t = 220$	$\because 40 > 38$
$r = 220$	$t = 250$	
$\underline{r = 250}$	$t = \text{NULL}$	

void Insert (Node *t, int key)

{ Node *r = NULL, *p;

while ($t \neq \text{NULL}$)

{ $r = t$;
if ($key == t \rightarrow \text{data}$)
return;

else if ($key < t \rightarrow \text{data}$)
 $t = t \rightarrow \text{lchild}$;

else
 $t = t \rightarrow \text{rchild}$;

}

$p = \text{malloc}(\dots)$; $p \rightarrow \text{data} = \text{key}$;
 $p \rightarrow \text{lchild} = p \rightarrow \text{rchild} = \text{NULL}$;
if ($p \rightarrow \text{data} < r \rightarrow \text{data}$) $r \rightarrow \text{lchild} = p$;
else $r \rightarrow \text{rchild} = p$;

}

Recursive Insert in BST

(283)

```

Node *insert(Node *p, int key)
{
    Node *t;
    if (p == NULL)
    {
        t = malloc(...);
        t->data = key;
        t->lchild = t->rchild = NULL;
        return t;
    }
    if (key < p->data)
        p->lchild = insert(p->lchild, key);
    else if (key > p->data)
        p->rchild = insert(p->rchild, key);
    return p;
}

```

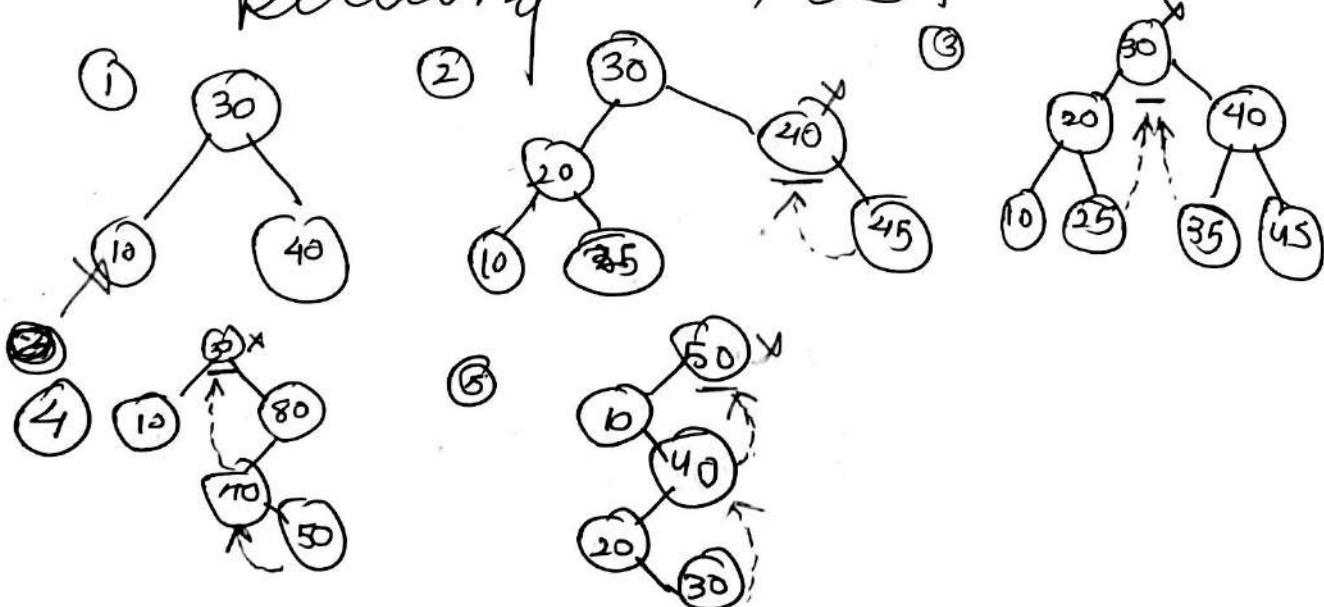
Creating a Binary Search tree.

insert : O

search: log n time for each element

$\therefore O(n \log n) : O(n \log n)$

Deleting from BST



228

226

struct Node * InBst (struct Node * p)

{ struct Node * q; while (p != NULL) { if (p->rchild != NULL) { q = p->rchild; } else { p->rchild = InBst (p->lchild); return p; } } }

↓
struct Node * InDsc (struct Node * p)

{ struct Node * q; while (p != NULL) { if (p->lchild != NULL) { q = p->lchild; } else { p->lchild = InDsc (p->rchild); return p; } } }

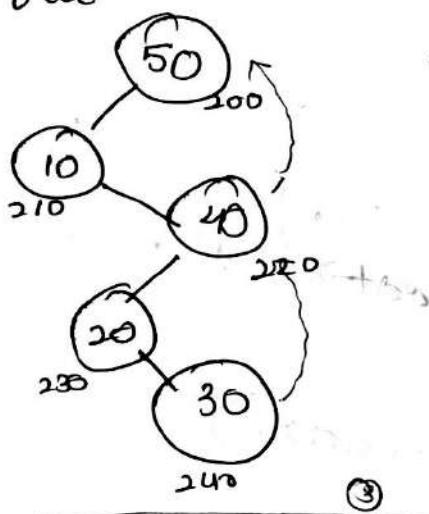
Y
struct Node * Delete (struct Node * p, int key)

{ struct Node * q; if (p == NULL) { return NULL; } if ((p->lchild == NULL & p->rchild == NULL)) { if (p == root) { root = NULL; free(p); return NULL; } } }

if (key < p->data) p->lchild = Delete (p->lchild, key);
else if (key > p->data) p->rchild = Delete (p->rchild, key);
else { if (Height (p->lchild) > Height (p->
rchild)) { q = InBst (p->lchild); } }

②
 $p \rightarrow data = q \rightarrow data$
 $p \rightarrow lchild = Delete(p \rightarrow lchild, q \rightarrow data);$
 ↓
 else
 {
 $a = InPrece(p + rchild);$
 $p \rightarrow data = a \rightarrow data$
 $p \rightarrow rchild = Delete(p \rightarrow rchild, q \rightarrow data);$
 }
 }
 return p;

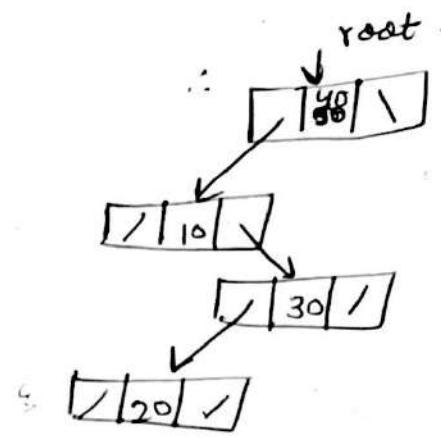
For tree



returns(230)
 returns(220)
 returns(210)
 returns(20)

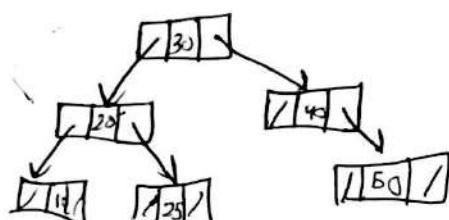
$delete(50, 200, 40)$
 key = $p \rightarrow data$
 height($200 \rightarrow lchild$) = 4
 height($200 \rightarrow rchild$) = 0
 $a = InPrece(210) = 220$
 $200 \rightarrow data = 40$
 ~~$210 \rightarrow data$~~ = $delete(210, 40)$
 $delete(210, 40)$
 key > $p \rightarrow data$
 $\therefore 220 = Delete(220, 40)$
 $delete(220, 40)$
 key = $p \rightarrow data$
 height($220 \rightarrow lchild$) = 2
 height($220 \rightarrow rchild$) = 0
 $a = InPrece(230) = 240$
 $220 \rightarrow data = 30$
 $230 = Delete(230, 30)$
 $Delete(230, 30)$
 key > $p \rightarrow data$
 $230 \rightarrow rchild = Delete(240, 30)$
 $delete(240, 30)$
~~free(240)~~
 return NULL

22
230



Generating BST from Preorder

Preorder: 30 20 10 15 25 40 50 45



IMP void createlpre (int pre[], int n)

{ stack stk;

Node *t;
int i = 0;
root = new Node;
*root → data = pre[i++];
*root → lchild = *root → rchild = NULL

p = root;

while (i < n)

= if (pre[i] < p → data)

{ t = new Node;

t → data = pre[i++]

t → lchild = t → rchild = NULL;

p → lchild = t;

push (&stk, p);

p = t;

}

else

{ if (pre[i] > p → data & pre[i] < stack[stacktop] → data)

{ t = new Node; t → data = pre[i++];

t → lchild = t → rchild = NULL;

p → rchild = t; p = t; }

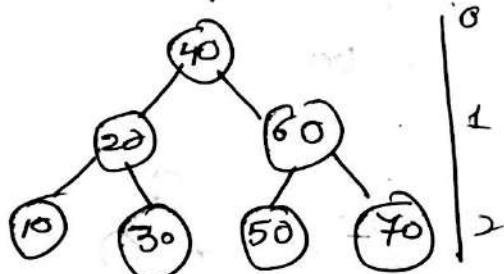
when stack top
is empty it will
return so, largest
value.

else
 $P = \text{pop}(\text{Lst})$

$\frac{1}{2}$
 $\frac{3}{4}$
 $\frac{5}{6}$
 $\frac{7}{8}$

Drawbacks of Binary Search Tree

key 40, 20, 30, 60, 50, 10, 70



height = 2

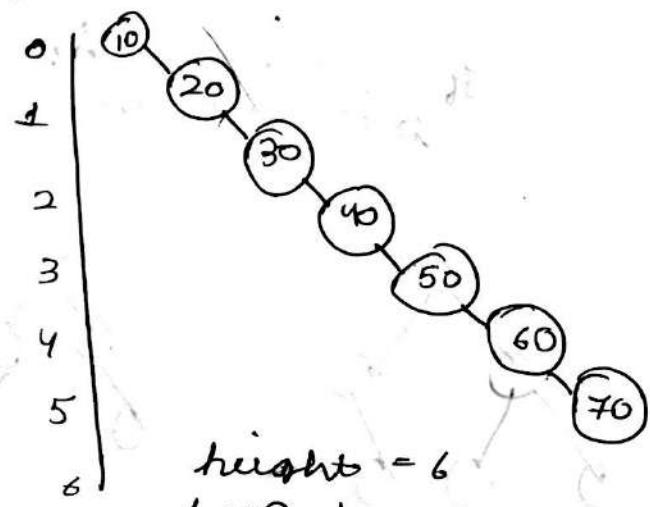
$$h = \log_2(n+1) - 1$$

$O(\log n)$

- ∴ Height of BST depends on order of insertion & ∵ order of insertion is not in control of programmer ∵ user will give the order
- ∴ Height of Binary search tree is also not in control it can be of $O(\log n)$ to $O(n)$

AVL trees are height Balanced tree

∴ Height of BST is not always $\log n$.



height = 6

$$h = n - 1$$

$O(n)$

(i) Binary Search Level order

Recursive :

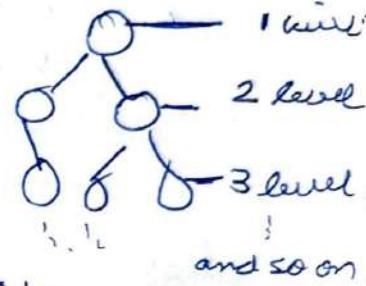
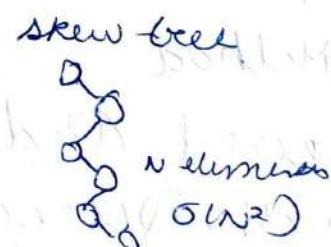
```
void printKthLevel(node *root, int k)
{
    if (root == NULL)
        return;
    if (k == 1) {
        cout << root->data << " ";
        return;
    }
    printKthLevel(root->left, k-1);
    printKthLevel(root->right, k-1);
    return;
}
```

It will print k^{th} level elements when k become 1 it reaches the k^{th} level.

Now to print Level Order.

```
void printAllLevels(node *root) {
    int H = height(root);
    for (int i=1; i<=H; i++)
    {
        printKthLevel(root, i);
        cout << endl;
    }
    return;
}
```

Worst case : $O(N^2)$



(i) Iterative Version
BFS $O(n)$)

→ Queue is used. (Picks level Nodes)

Pseudocode:

```
[ q.push(root)
  while (!q.empty())
    Pick one node at front
    Pop it
    Push its children ]
```

code:

```
void bfs(Node* root)
```

```
queue<node*> q;
```

```
q.push(root);
```

```
while (!q.empty()) {
```

```
node* f = q.front();
```

```
cout << f->data << ", ";
```

```
q.pop();
```

```
if (f->left) {
```

```
q.push(f->left);
```

```
if (f->right)
```

```
q.push(f->right);
```

```
}
```

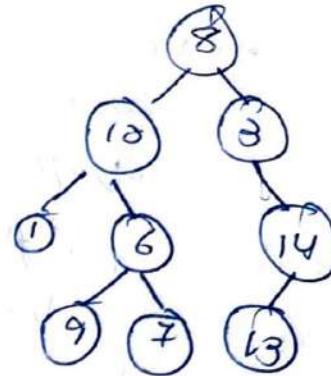
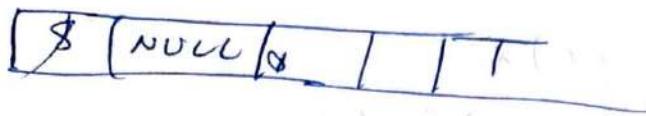
```
return;
```

}

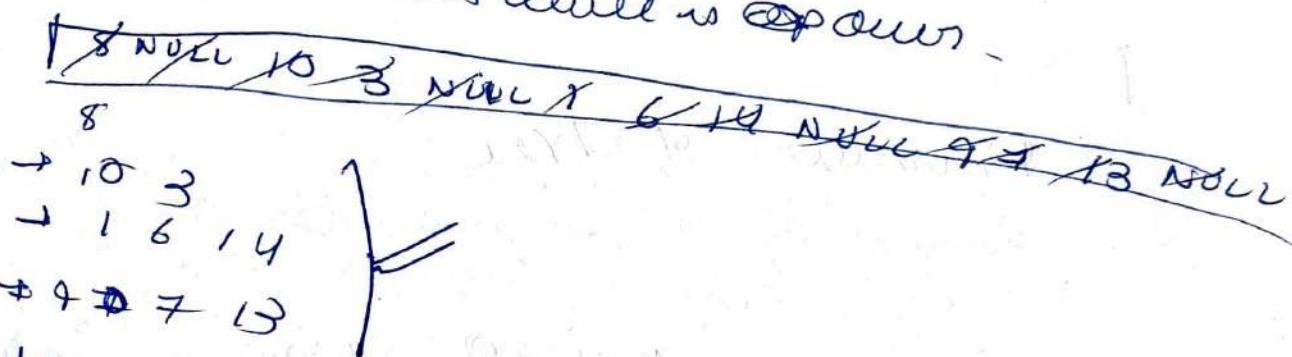
v. ~~Top~~ Method

Level Order Traversal $O(n)$
with Queue.

By using NULL characters
at start



as we encounter NULL we
change line pop up the
NULL check if Queue is empty
it is not empty ~~then~~ we will
push NULL because NULL denotes
that previous level is ~~over~~ over.



Code:

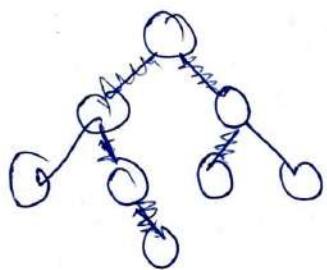
```
void bfs(node* root) {
    queue<node*> q;
    q.push(root);
    q.push(NULL);
    while (!q.empty()) {
        node*f = q.front();
        q.pop();
        if (f == NULL) {
            cout << endl;
            q.push(NULL);
            if (!q.empty())
                q.push(NULL);
        }
    }
}
```

(iv)

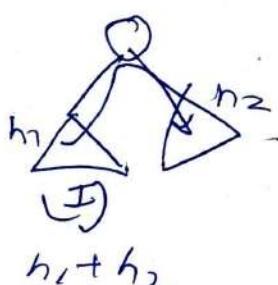
```
else {  
    cout << f->data << ",";  
    q.pop();  
    if (f->left) {  
        q.push(f->left);  
    }  
    if (f->right) {  
        q.push(f->right);  
    }  
}
```

Diameter of Tree

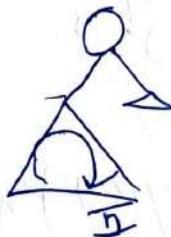
Longest distance between two nodes



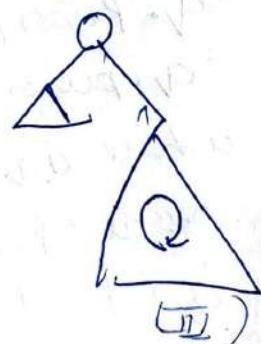
- Three cases if diameter includes ~~includes root~~
- (i) It includes Root.
 - (ii) It is on the left side of Root
 - (iii) on the ~~is~~ right side of root



$$h_1 + h_2$$



diameter (left)



diameter (right)

$$\max(I, II, III)$$

int diameter(node *root) IMP

* if (root == null)
{ return 0; }

else

int h1 = height(root->left);

int h2 = height(root->right);

int op1 = h1 + h2;

int op2 = diameter(root->left);

int op3 = diameter(root->right);

return max(max(op1, op2), op3);

$\downarrow O(N^2)$

Preorder
Traversal

Optimize O(N)

class Pair {

* private,

int diameter;

int height;

,

ie height nahi nikalna
h baas base height
aur diameter of each
node returns karwana h.
+ Revision post-order
form mehndi chain.

Pair fastDiameter(node *root) {

if (root == null) {

p.diameter = p.height = 0;

}

// otherwise

Pair left = fastDiameter(root->left);

Pair right = fastDiameter(root->right);

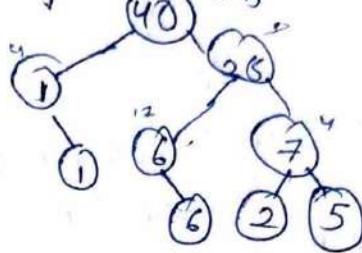
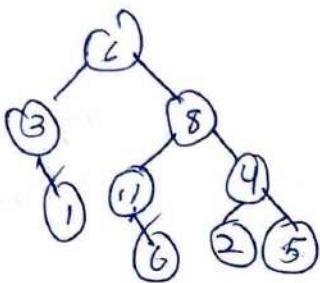
p.height = max(left.height, right.height);

p.diameter = max(left.height + right.height,

max(left.diameter, right.diameter));

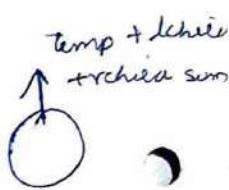
(V1)

Sum replacement
without affecting leaf nodes



No change in leaf nodes.

Solutions:
Post order form



Main concept: we will store $\text{root} \rightarrow \text{data}$ before updating, and the function will return sum of its child and its previous $\text{root} \rightarrow \text{data}$.

```
int replacementSum(node *root) {
    if (root == NULL) {
        return 0;
    }
    if (root->left == NULL && root->right == NULL)
        // returns root->data;
    }
```

// Recursive part

```
int leftSum = replacementSum(root->left);
int rightSum = replacementSum(root->right);

int temp = root->data
```

$\text{root} \rightarrow \text{data} = \text{leftSum} + \text{rightSum}$

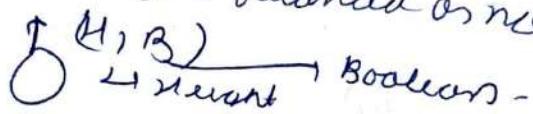
IMP ↗
return $\text{temp} + \text{root} \rightarrow \text{data}$ // New Concept

(cont)

Check if Height Balanced or Not in O(N)

$$S_1 |h_1 - h_2| \leq 1$$

but calculating h_1, h_2 every time will give $O(N^2)$ - we will use pair between cons
cept in which we return height and
bool for balanced or not.



Code:

```
HBPair { public:
```

```
bool isHeightBalanced();
int height;
bool balanced;
```

```
{ HeightBalance (Node* root) };
```

```
HBPairs p
if (root == NULL) {
    p.height = 0
    p.balanced = true
    return p
}
```

// Recursive Case

```
HBPairs left = isHeightBalanced (root->left);
HBPairs right = isHeightBalanced (root->right);
```

```
+ if (abs(left.height - right.height) <= 1) {
    + if (left.balanced & right.balanced) {
        p.balanced = true
    }
}
```

```
+ else if (p.balanced = false) {
    p.balanced = false
}
```

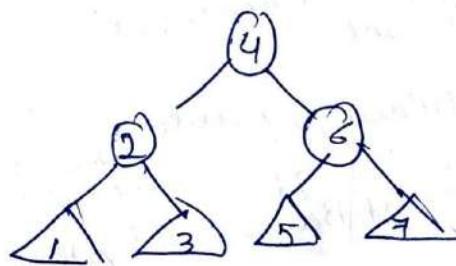
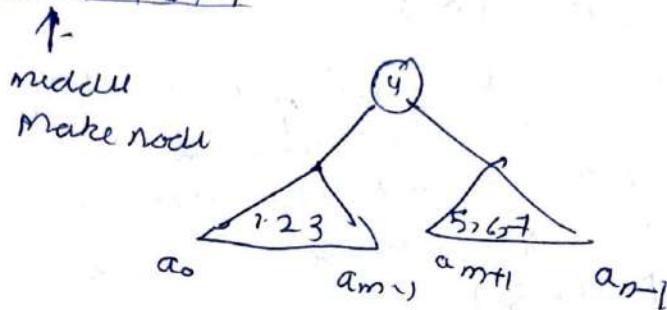
```
+ else if (p.balanced = false) {
    p.balanced = false
}
```

```
+ return p;
```

```
}
```

(VIB)

Build Balanced Tree from Array
a (Array) 1 2 3 4 5 6 7



```
{ if (size) createNode(a[mid]);  
    return NULL; }  
else { // Recursion case  
    int mid = (s + e) / 2;  
    node *root = new Node(a[mid]);  
    root->left = buildTreefromArray(a[s, mid - 1], fn(a, s, mid - 1));  
    root->right = buildTreefromArray(a[mid + 1, e], fn(a, mid + 1, e));  
    return root; }
```

Preorder + Inorder build is same structure
as above

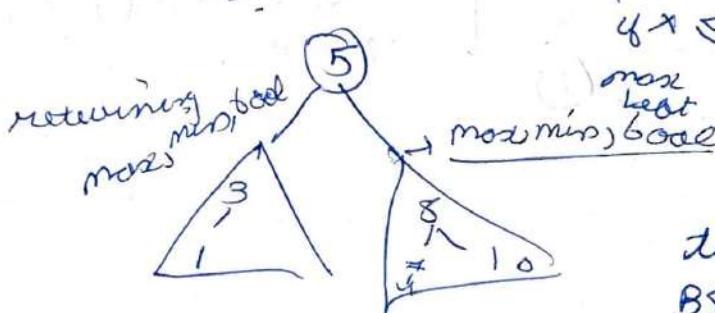
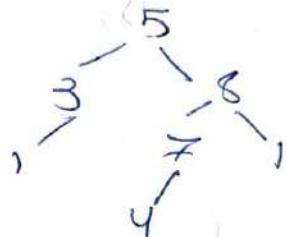
(10)

* Check for BST

v. IMP

Checking left & right child & concluding
 if $\text{left_child} < \text{root_data} < \text{right_child}$ it will be
WRONG

ex:

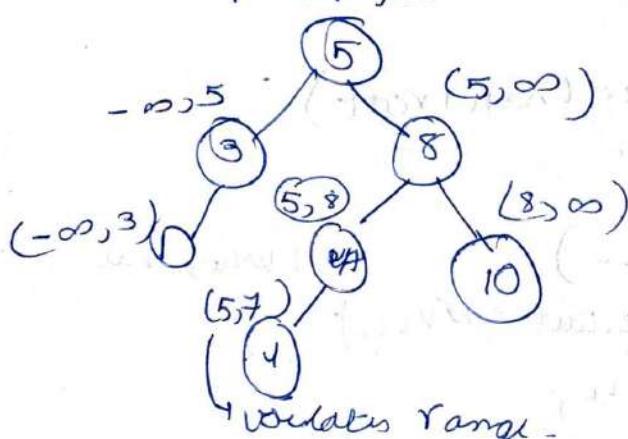


$x \leq \min \text{ and } x \geq \max$
 if $x \leq \min$ and $x \geq \max$,
 both children are BST

thus we conclude its
BST

\therefore this is Bottom Up Approach.

Other One is Top Down Approach
 setting min & max range of nodes



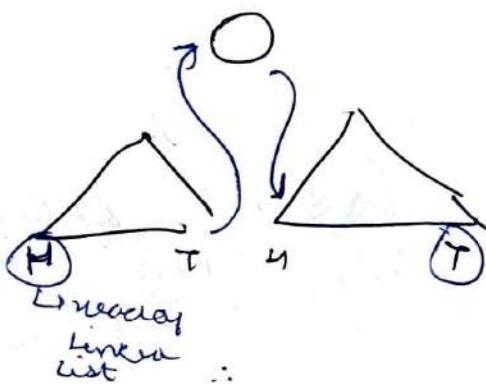
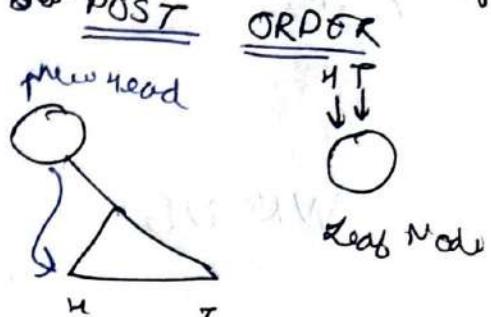
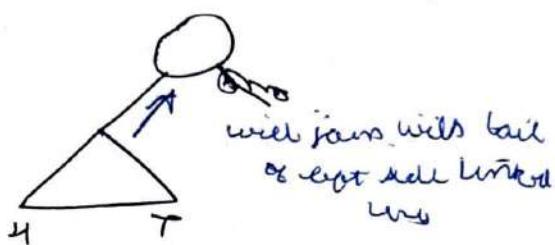
Validates Range

Goal is BST (node * root, int minV = INT_MIN, int maxV = INT_MAX)
 if (root == NULL) {
 } returns true;
 if (root->data >= minV & root->data <= maxV & isBST(root->left))
 {
 } minV < root->data) & isBST(root->right), root->data <= maxV
 true; else false;

(x)

Flattening of BST into a sorted linked

List : Not INORDER BUT PRE-POST ORDER
LOC-IC cases Recursively



[∴ Recursive & call will
recurs head & tail
separately]

Code

```

class Node {
    data;
    Node *left;
    Node *right;
};

Node *flatten(Node *root) {
    Node *head, *tail;
}

```

Linked list flattens (node+root)

```

{
    linkedlist l;
    if (root == NULL)
        return l; // when tail is empty
    if (l.head == l.tail == NULL)
        l.head = l.tail = root;
    else
        l.tail = NULL;
    }
    // Leaf node
    if (root->left == NULL & root->right == NULL)
        l.head = l.tail = root;
    else
        l.tail = root;
    return l;
}

```

// left is not null

if (root + left == NULL & root + right == NULL)
 ↳ linkedlist leftll = flatten (root + left)
 $\text{left} \ll \text{tail} \rightarrow \text{right} = \text{root};$
 l. head = left & head
 l. tail = root;
 returns l;

}

// right is not null

if (root + left == NULL & root + right != NULL)

↳ linkedlist rightll = flatten (root + right)
 $\text{root} + \text{right} = \text{right} \ll \text{head}.$
 l. head = root;
 l. tail = rightll. tail;
 returns l;

}

// Both sides are not null

first
order
form [linkedlist leftll = flatten (root + left)
 linkedlist rightll = flatten (root + right)
 $\text{left} \ll \text{tail} \rightarrow \text{right} = \text{root};$
 $\text{root} + \text{right} = \text{right} \ll \text{head};$
 l. head = left & head;
 l. tail = rightll. tail;
 returns l;

}

(Xvi) Right View of Binary tree

i) using Queue

~~EASY APPROACH~~ (use NULL as characters.)

~~THOUGH~~ [we need the last node of every level]
∴ By using queue.

```
void PrintRightView(Node* root)
```

{ if (!root)

return;

queue<Node*> q;

q.push(root);

while (!q.empty())

{ // no. of nodes at current level
 int n = q.size();

// Traverse all nodes of current level.

```
for (int i=1; i<=n; i++)
```

{ Node* temp = q.front();

q.pop();

(I) if (i==n)

cout << temp->data << " ",

// Add left node to queue

if (temp->left != NULL)

q.push(temp->left);

// Add right node to queue

if (temp->right != NULL)

q.push(temp->right);

= = =

II Recursive Approach

we will take level & max level
max level will be telling upto which level we have found right view.

```

void rightViewUtil(struct Node* root, int level,
                    int *max-level)
{
    // Base case
    if (root == NULL)
        return;
    // If this is the last node of its level
    if (*max-level < level)
    {
        cout << root->data << endl;
        *max-level = level;
    }
    // Recursion for right subtree first
    // then left subtree.
    // A wrapper over util.
    void rightViewUtil(struct Node* root)
    {
        int max-level = 0
        rightViewUtil(root, 1, &max-level);
    }
}

```

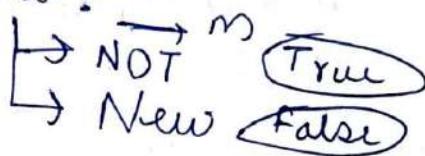
CONCEPT:
 : max-level is storing upto which level we have printed right view.
 & level is denoting the current level of the tree.

① Trie Data Structures

Problem: List of strings

"No", "Not", "Necas", "Hello", "Apple" ... \xrightarrow{N} (n)

Queries:

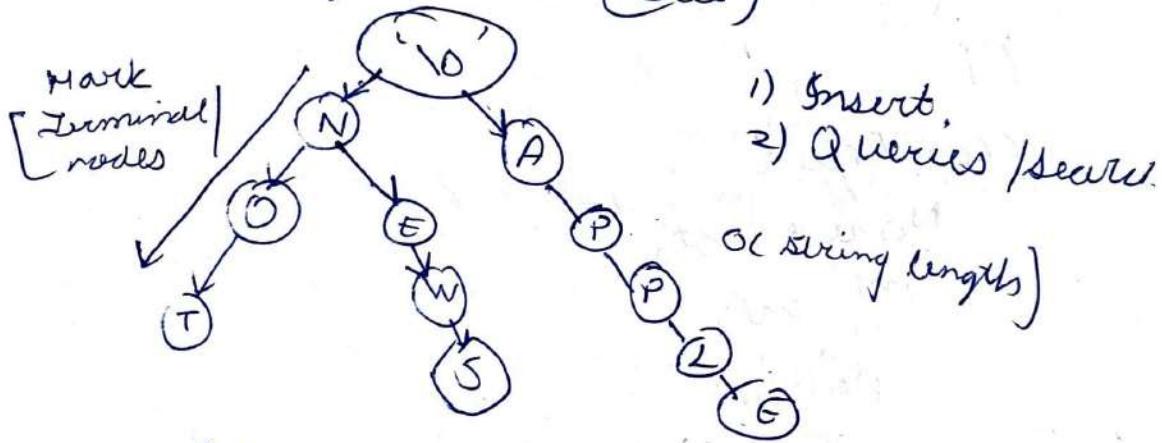


① Brute Force

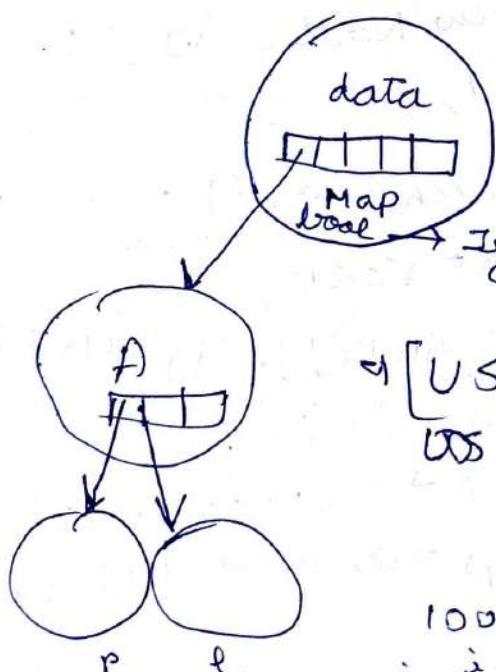
$O(NM)$

② Hashing $O(N)$

③ Prefix Tree (Trie)



\therefore Node can be or



$map < \text{char, node*} \rangle$
like char a have this
node as its child.
Terminal
(whether its terminal or not)

④ [USEFUL APPLICATION]

In tree calls we have multiple
phone no with some prefix.

1000 contacts \rightarrow hashmap
is faster than Trie.

(2)

Code:

```

#include <bits/stdc++.h>
using namespace std;

class Node {
public:
    char data;
    unordered_map<char, Node*> children;
    bool terminal;
};

Node(char d) {
    data = d;
    terminal = false;
}

class Trie {
private:
    Node* root;
    int cnt;
public:
    Trie() {
        root = new Node('\0');
        cnt = 0;
    }

    void insert(char* w) {
        Node* temp = root;
        for (int i = 0; w[i] != '\0'; i++) {
            char ch = w[i];
            if (temp->children.count(ch)) {
                temp = temp->children[ch];
            }
        }
    }
}

```

(3)

else {

```

Node * n = new Node(ch);
temp -> children[ch] = n;
temp = n;
    }
```

```

} temp -> terminal = true;
    }
```

bool find (char * w) {

```

Node * temp = root;
for ( int i = 0; w[i] != '\0'; i++ ) {
    char ch = w[i];
    if ( temp -> children . count ( ch ) == 0 ) {
        return false;
    }
    else {
        temp = temp -> children [ ch ];
    }
}
return temp -> terminal;
}
```

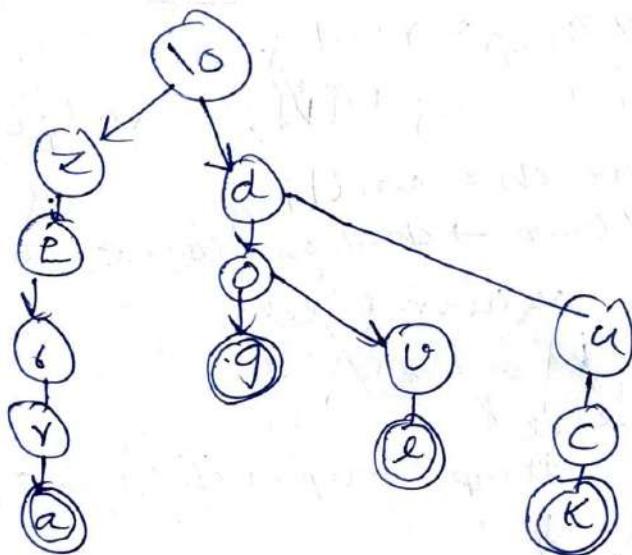
Time Complexity $O(\text{strlen}) \approx O(1)$

(4)

Unique Prefix Array

Array \rightarrow ["zebra", "dog", "dove", "duke"]
 output \rightarrow ["z", "dog", "dov", "du"]
 BruteForce : $O(n^2 \times \text{avg. len})$

Since we are dealing with prefix
 so we should think of TRIE Data
 structure



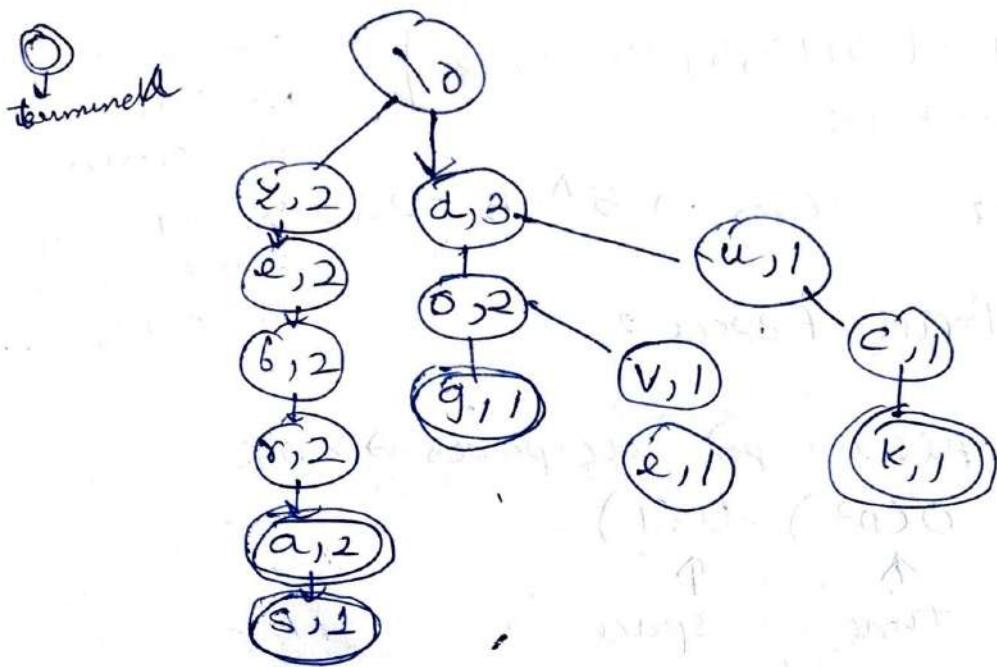
\therefore For unique prefix we observe that
 when we have no f withs splits
 that would give us unique prefix
 But not always.

To ex: if we have zebra & zebras
 both then there would be no split
 at z but then too it cannot be
 unique prefix.

(5)

Now on thing we can add in Node class

- count is a maintains ~~count~~ count of no of times the node is visited with insertion.



Now we see that as soon we get the count 1 till that point we have to take for unique prefix.

For Zebra,

there is no Node with till a $\therefore \underline{-1}$ $\text{count} = 1$

Zebra will have "zebras" as unique prefix.

Time complexity : $O(n \times \max_{\substack{\uparrow \\ \text{word}}})$

we are traversing n-elements of array.

Ques: Maximum XOR Pair Value in Array

I/P - [3, 10, 5, 25, 2, 8]

O/P → 28

Explanation → $5 \wedge 25 = 28$

3 - 00011

10 - 01010

5 - 00101

25 - 11001

2 - 00010

8 - 01000

Brute Force?

All the possible pairs → XOR

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

↑
Time

↑
Space:

~~Maximum XOR Pair Problem~~

By Trie's we have to compare bit of each no. at a time.

How we calculate?

3
10
5

 → suppose we known ans of this

25 → now we know 25 so we need to check all pairs of 25, 5 25, 10 25, 3 and see if the XOR of any > 28

If yes then we will set value of 1
How we will check?

ns
1 1 0 0 1

we will see in Trie that do we have 0 at MS if yes then go on that part of Trie -

7

```

#include <bits/stdc++.h>
using namespace std;
class trieNode {
public:
    trieNode *left;
    trieNode *right;
};

void insert(int n, trieNode *head) {
    for (int i = 31; i >= 0; i--) {
        int bit = (n >> i) & 1;
        if (bit == 0) {
            if (curr->left == NULL)
                curr->left = new trieNode();
            curr = curr->left;
        } else {
            if (curr->right == NULL)
                curr->right = new trieNode();
            curr = curr->right;
        }
    }
}

int findMaxXorPair(trieNode *head, int *arr, int n, int el) {
    int maxScore = INT_MIN;
    trieNode *curr = head;
    int value = el;
    int currXor = 0;
}

```

⑧
 for (int j = 3; j >= 0; j--) {
 int b = (value > j) & 1;
 if (b == 0) {
 if (curr->right != NULL)
 curr = curr->right;
 curr-xor = (int) pow(2, j);
 }
 else
 curr = curr->left;
 }
 else {
 // current bit is 1
 if (curr->left != NULL)
 curr = curr->left;
 curr-xor += (int) pow(2, j);
 }
 else
 curr = curr->right;
 }
 if (curr-xor > max-xor)
 max-xor = curr-xor;
 return max-xor;
 }
 int main() {
 int n;
 cin >> n;
 int* arr = new int[n]();
 for (int i = 0; i < n; i++) {
 cin >> arr[i];
 }
 trieNode* head = new trieNode();
 int result = INT-MIN

⑨

```

foo ( int i=0; i<n; i++ ){
    insert ( arr[i] , head );
    int x = findMaxOrPair ( head , arr , n , arr[i] );
    result = ( result < x ) ? x : result;
}
cout << result << endl;
}

```

Subarray with Maximum XOR

{ 3, 10, 5, 25, 2, 8 } { 3, 3¹, 10, 8¹, 10¹, 5, 3², 10², 5², 25,
 $f(L, R) = f(0, R) \text{ XOR } f(0, L-1)$ }

{ a, b, c, d, e, f } ↘

↳ Max XOR pair
 (converted into two)

232

AVL

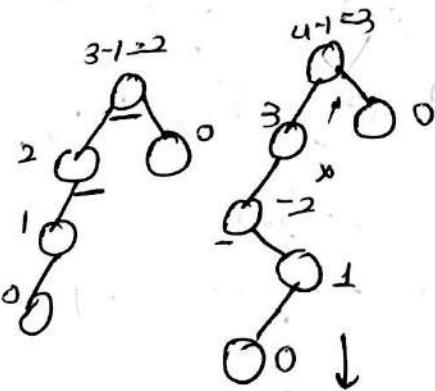
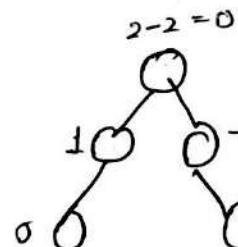
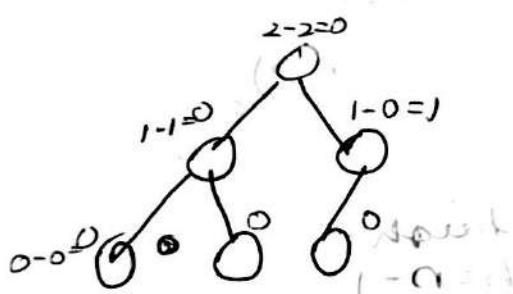
$b_f = \text{height of left subtree} - \text{height of right subtree}$

$$b_f = h_L - h_R = \{-1, 0, 1\}$$

$$|b_f| = |h_L - h_R| \leq 1 \quad \begin{array}{l} \text{balanced} \\ \text{unbalanced} \end{array}$$

$$|b_f| = |h_L - h_R| > 1$$

If any tree has unbalanced node then that tree will be unbalanced tree.



$|b_f| = 3$ is not possible invalid example of b_f can be 2 or 3 or large.

Inserting in AVL with Rotations

Rotation is always applied on 3 nodes

Rotation for Insertion

I LL
critical

After inserting 10

Perform LR-rot.

right

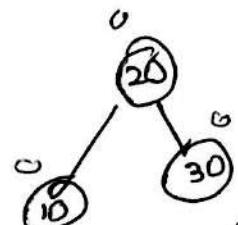
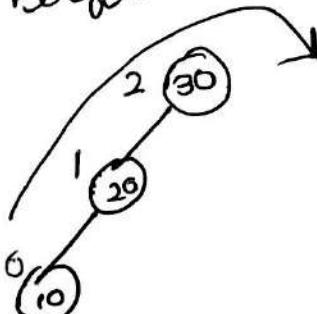
after LR-rot.



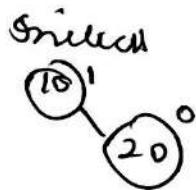
LL imbalance
(left-left)

since we
have inserted
10 on root of 20

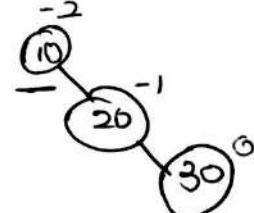
at 30



II RR Rotation

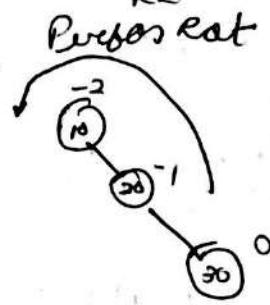


After Inserting 30

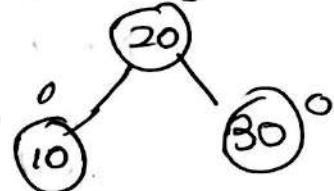


RR Imbalance

Left RE



After RR Rotation



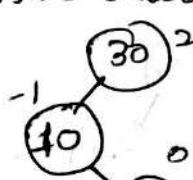
233

III LR Rotation

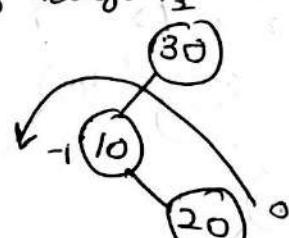
Initial



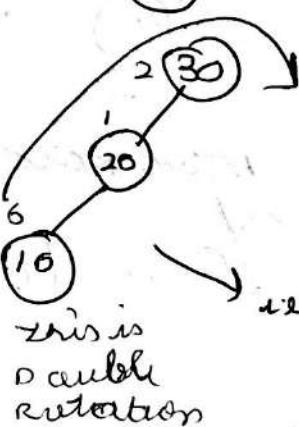
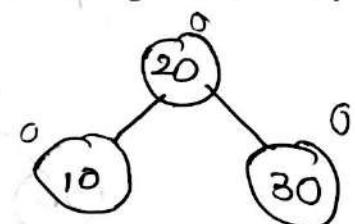
After Inserting 20 Perform Rot.



LR Imbalance

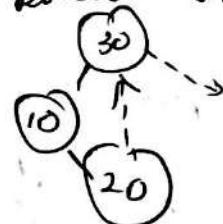


After LR-Rot.



this is
double
rotation

Right Method

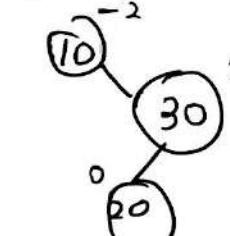


IV RL Rotation

Initial



After Inserting 20

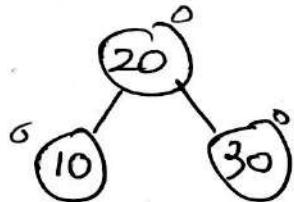


RL Imbalance

Perform RL rot



After RL Rd



Double
Rotation

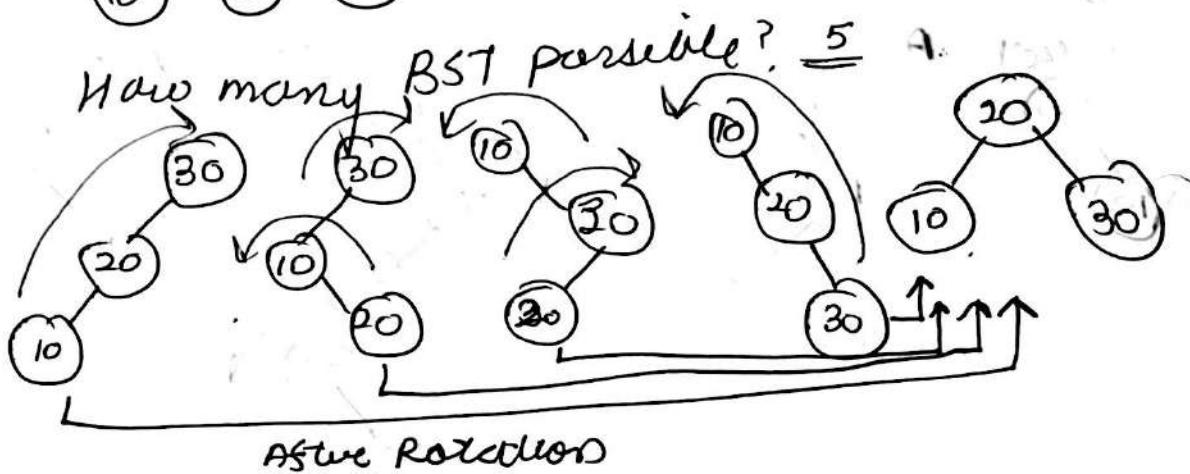
22

234

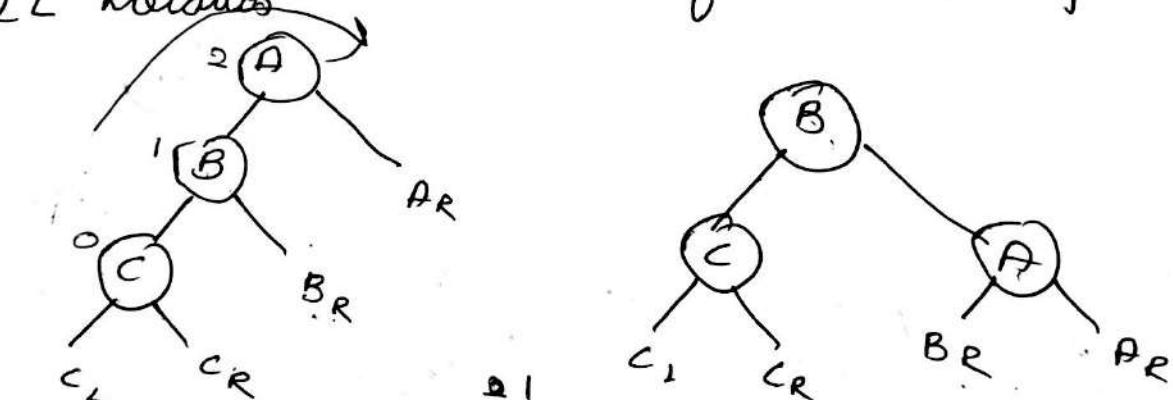
LL Rotations \rightarrow single rotation
 RR Rotations
 \leftarrow L.R. rotations \rightarrow double rotation
 R.L. Rotations (but not 2 rotations)

11

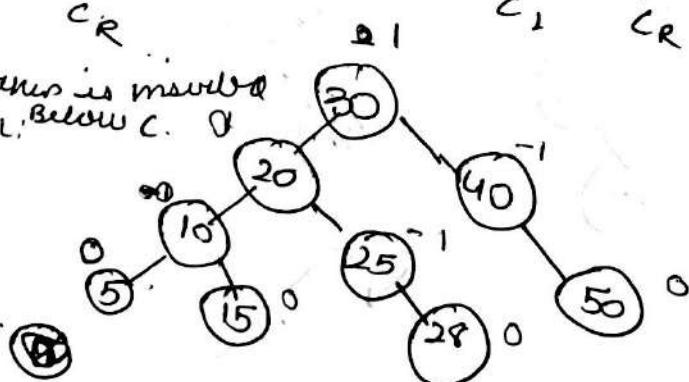
Idea about AVL



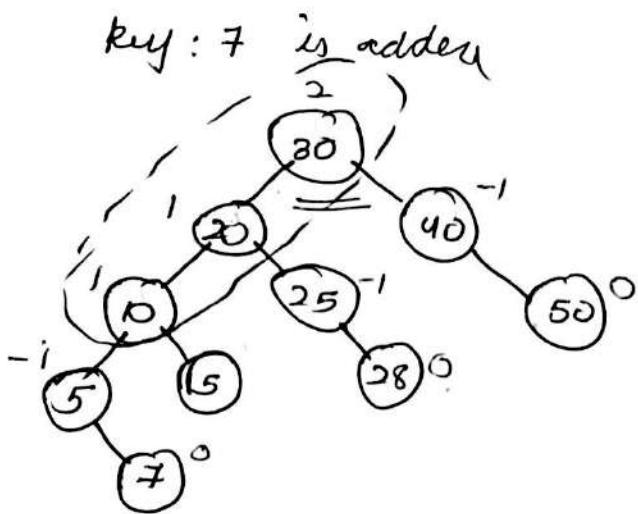
\pm LL Rotations Formula of Rotation for Insertion



If insertion is involved
new node is inserted
below C.



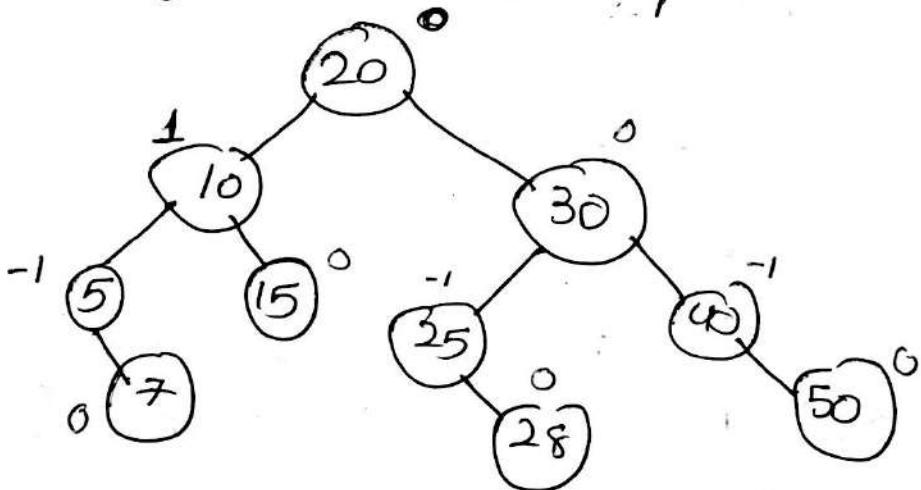
Now



Now by adding 7 i.e. LLR of 30 should we call LLR imbalance? No

we only see three nodes ∵ from 30 it is added at Left of left of 30 ∴ LL imbalance

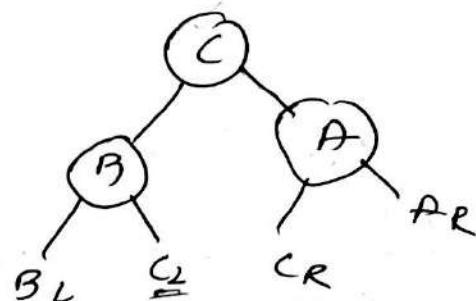
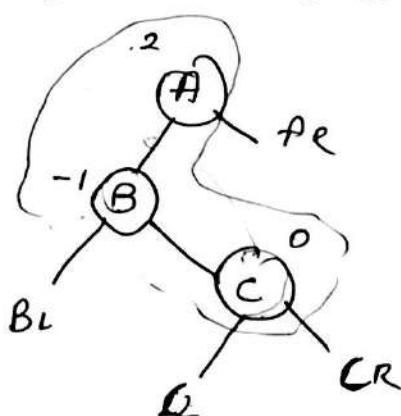
∴ After Rotation



∴ Balanced Tree

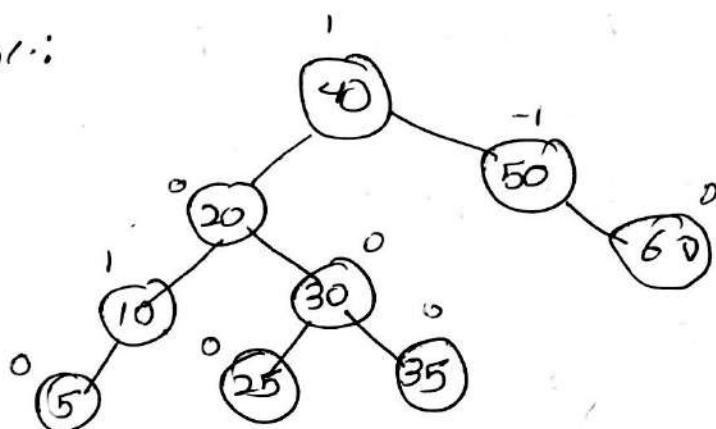
235

LR Rotation

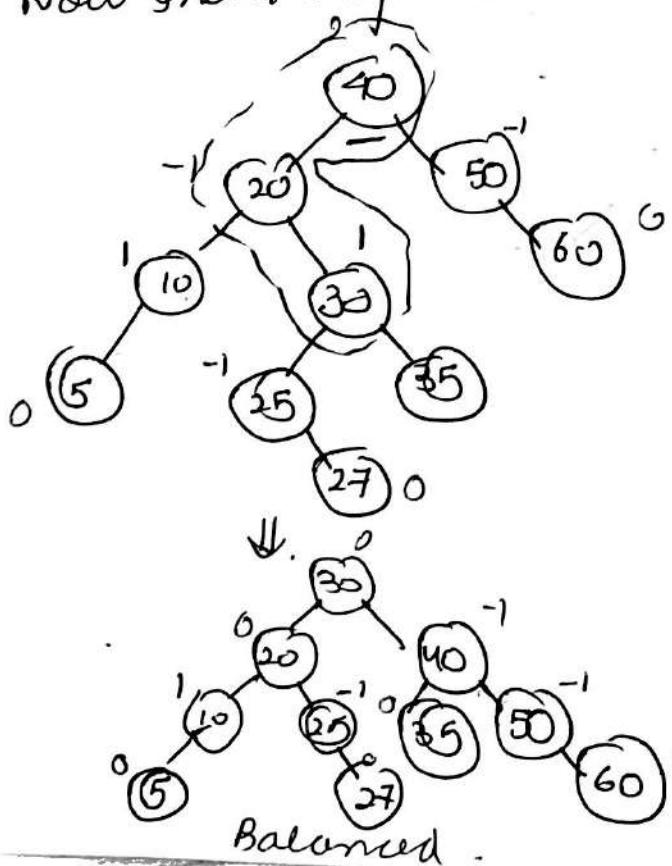


for C's children
 C_L will be on left of C
 C_R will be on right of C

201:



Now insert key = 27



\therefore Node is inserted on L R L R but we see only 3 nodes
 \therefore LR

Let's Code LL & LR ~~Tree~~ Rotations

we have modified `insert` func. of BST

```
#include <stdio.h>
```

```
struct Node
```

```
{
    struct Node *lchild;
    int data;
    int height;
    struct Node *rchild;
}
```

```
px root = NULL;
```

```
int NodeHeight(struct Node *p)
```

```
{
    int hl, hr
```

```
hl = p && p->lchild ? p->lchild->height : 0;
```

```
hr = p && p->rchild ? p->rchild->height : 0;
```

```
return hl > hr ? hl + 1 : hr + 1;
```

```
}
```

```
int BalanceFactor(struct Node *p)
```

```
{
    int hl, hr
```

```
hl = p && p->lchild ? p->lchild->height : 0
```

```
hr = p && p->rchild ? p->rchild->height : 0
```

```
return hl - hr;
```

```
}
```

```
struct Node * LLrotation (struct Node *p)
```

```
{
    struct Node * pl = p->lchild;
```

```
struct Node * plr = pl->rchild;
```

```
pl->rchild = p;
```

```
p->lchild = plr;
```

```
p->height = NodeHeight(p);
```

```
pl->height = NodeHeight(pl);
```

```
if (root == p)
```

```
root = pl; return pl;
```

235

struct Node * LR_Rotation (struct Node * p)

{
 struct Node * pl = p->lchild;
 struct Node * pr = pl->rchild;
 pl->rchild = pr->lchild;
 pr->lchild = pl->rchild
 pl->lchild = pl
 pl->rchild = p

} Modifications

pl->height = NodeHeight(pl);
 p->height = NodeHeight(p);
 pl->height = NodeHeight(pl);
 if (root == p)
 root = pl;
 return pl;

struct Node * RR_Rotation (struct Node * p)

struct Node * RR_Rotation (struct Node * p)
 { returns NULL; "Blank Function"

y
struct Node * RL_Rotation (struct Node * p)

{ returns NULL;

y
struct Node * RInsert (struct Node * p, int key)

{ struct Node * t = NULL;
 if (p == NULL)

{ t = (struct Node *) malloc (sizeof (struct Node));
 t->data = key;
 t->height = 1;
 t->lchild = t->rchild = NULL;
 return t;

y

if (key < p->data)

p->lchild = RInsert (p->lchild, key);

else if (key > p->data)

p->rchild = RInsert (p->rchild, key);

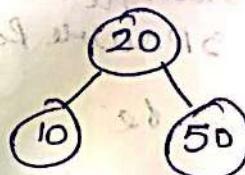
11. These task are done at returning time & Add to insert of BS ?.

P → height = NodeHeight(p);
 if (BalanceFactor(p) == 2 && BalanceFactor(p → lchild) == 1)
 return LLRotation(p);
 else if (BalanceFactor(p) == 2 && BalanceFactor(p → rchild) == -1)
 return LRRotation(p);
 else if (BalanceFactor(p) == -2 && BalanceFactor(p → lchild) == -1)
 return RRRotation(p);
 else if (BalanceFactor(p) == -2 && BalanceFactor(p → rchild) == 1)
 return R2Rotation(p);
 returns(p)

```

int main()
{
    root = RInsert(xroot, 50);
    RInsert(root, 10);
    RInsert(root, 20);
    return 0;
}
  
```

result:



AVL Tree

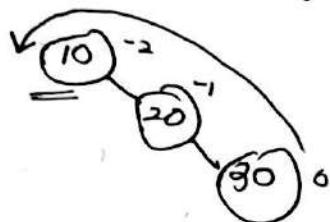
Note: Always tree is balanced as soon as it becomes imbalance ie $|bf| \neq 2$

240

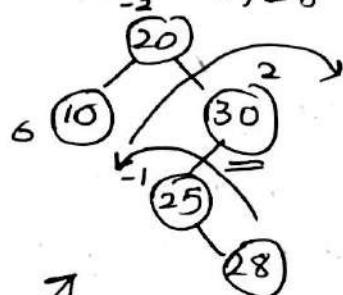
Creating AVL Tree

keys: 10, 20, 30, 25, 28, 27, 5

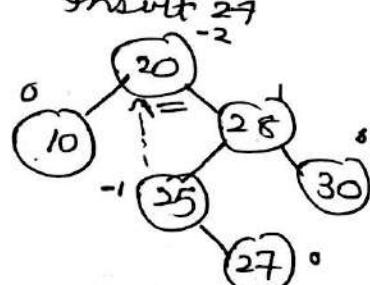
Insert 10, 20, 30



Insert 25, 28



Insert 27

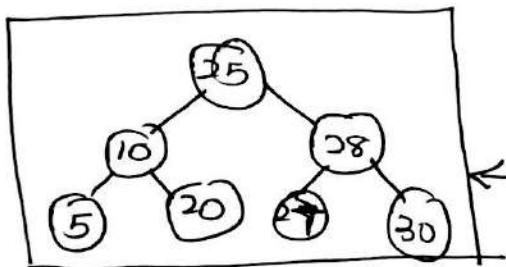


RR Rotation

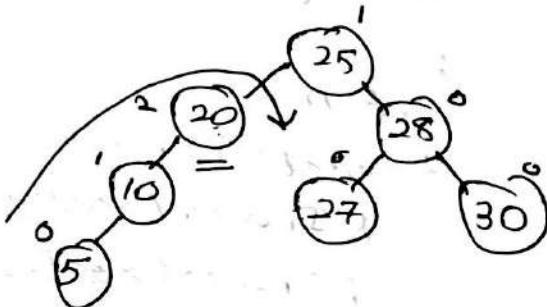


L-R Rotation

R-L Rotation



Insert 5



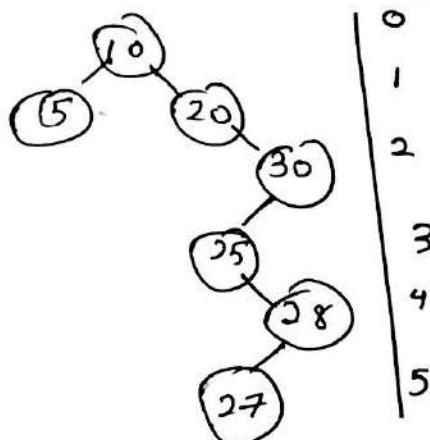
LL Rotation

∴ 4 Rotations

→ single rotations

→ Double rotations

Since BST formed would be



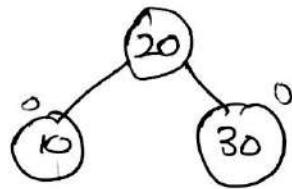
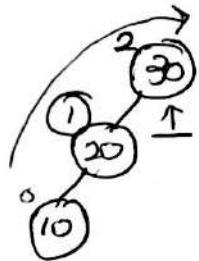
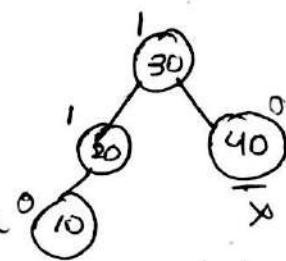
∴ In BST if we are searching for 27 we have to search for 6 times
But in AVL $\log(n+1) + 1$ times
ie 3 times

∴ Balanced

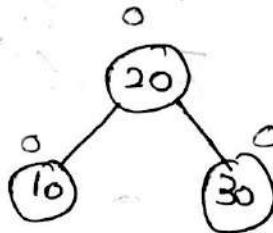
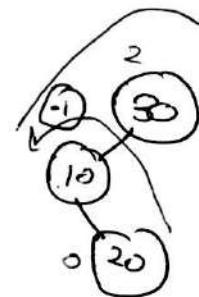
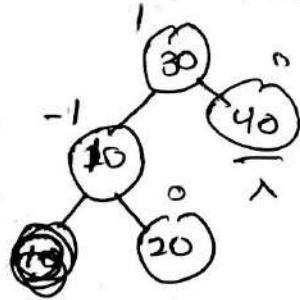
Deletions from AVL tree with Rotations

- 1) L(+) Rotation (LL rotation)
- 2) L(-) Rotation (LR rotation)
- 3) L(0) Rotation (LRS or LRU rotation you choose)
- 4) R(+)-Rotation (RL rotation)
- 5) R(-) Rotation (RR rotation)
- 6) R(0)-Rotation (RRO or RL rotation)

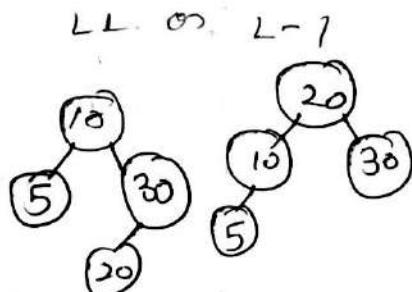
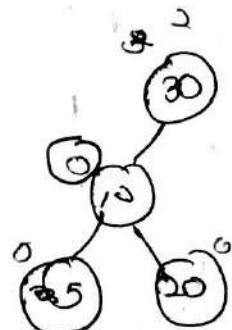
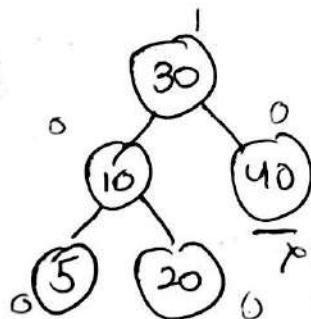
L(1)
 after removing
 40 left side
 became
 Imbalanced



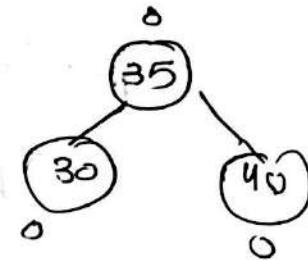
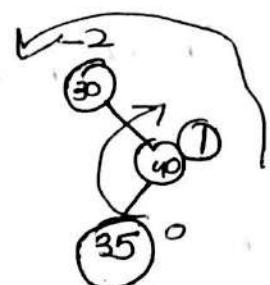
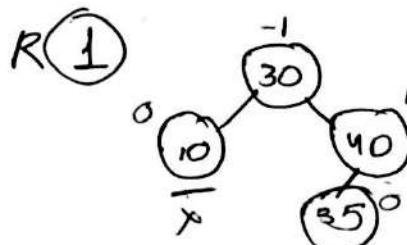
L(-1)



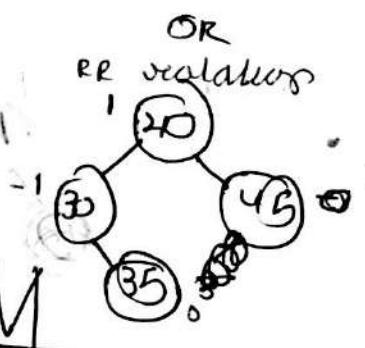
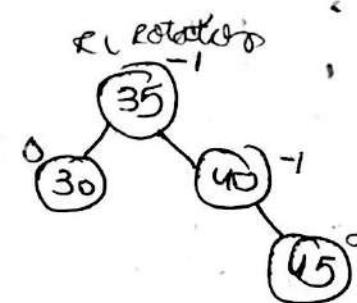
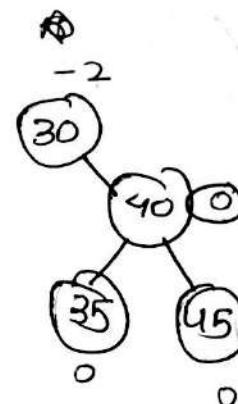
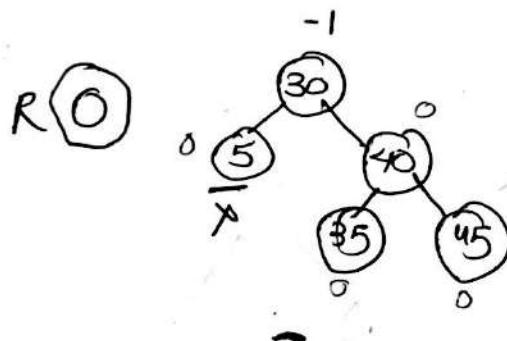
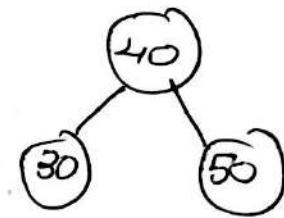
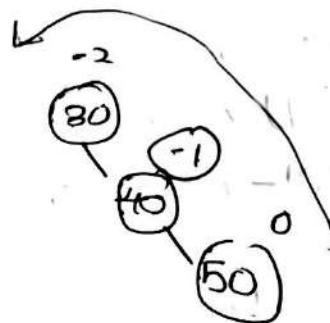
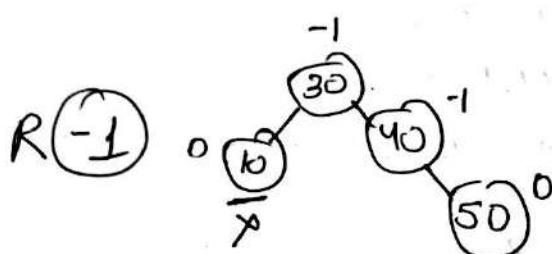
L(0)



242



Right side
is unbalanced.



Height vs Node of AVL

If Height is given find

$$\text{Max Nodes } n = 2^{h-1}$$

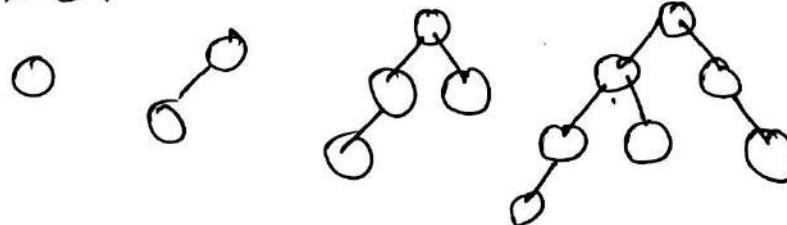
h is starting from 1

Min nodes : look Table

We will see by observations,

$$h=1 \quad h=2 \quad h=3 \quad h=4$$

$$n=1 \quad n=2 \quad n=5 \quad n=7$$



h	0	1	2	3	4	5	6	7
n	1	2	4	7	12	20	33	

$$N(h) = \begin{cases} 1 & h=0 \\ N(h-2) + N(h-1) & \text{otherwise} \end{cases}$$

min nodes

$$\therefore \text{Max Nodes} \approx n^{2^{h-1}}$$

\therefore Min height h would be

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

\equiv

Max Height h : look Tally
 \therefore If we have n terms 12 to 19 will have max height 5

& 20 to 32 will have max $h = 6$

Since min node when h is given
 $= N(h-2) + N(h-1) + 1$

It is like Fibonacci series
 which is balanced series.

$$\frac{\phi^{i+1}}{\phi^i} = 1.6 \text{ (approx)} \rightarrow \text{approx} \quad \text{for } i=1, 2, 3, \dots$$

\therefore Approximately if we get Max h from $\frac{\phi^{i+1}}{\phi^i}$

$$\therefore h = 1.44 \log_2(n+2)$$

Search Tree

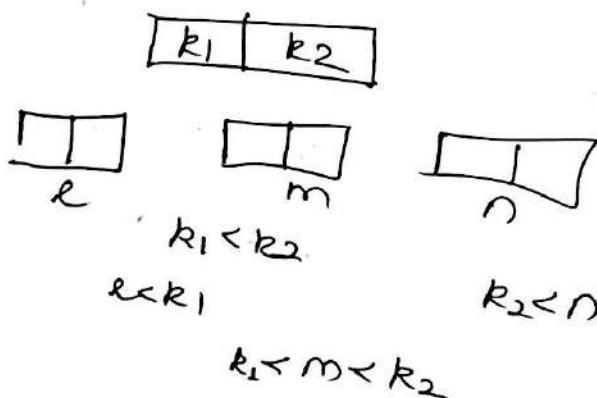
(2-3 tree)

- What are 2-3 Trees
- Insert / Create
- Delete
- Analysis
- Why 2-3 Trees

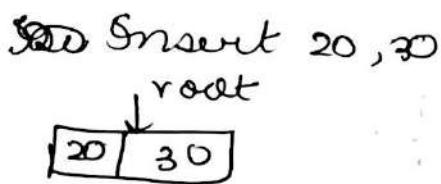
What are 2-3 Trees (Search Tree)

4-ways

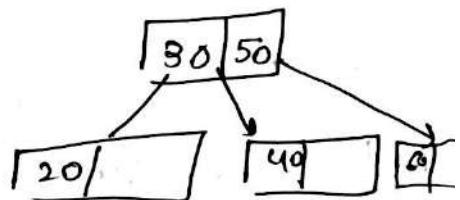
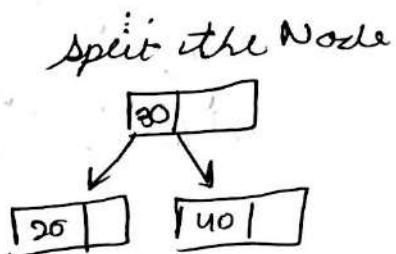
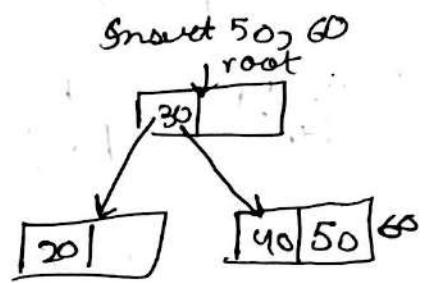
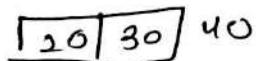
- Multicay Search Tree
- Degree 3
- Their height balanced tree is called B-Tree
- 2-3 Trees ~~are~~ B Trees of Degree 3
- All Leaf nodes at Same Level
- Every nodes must have $\left\lceil \frac{n}{2} \right\rceil = \left\lceil \frac{3}{2} \right\rceil = 2$
- Cannot have duplicate element as it is Search Tree



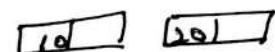
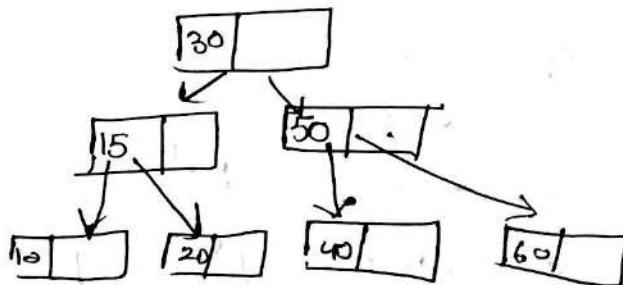
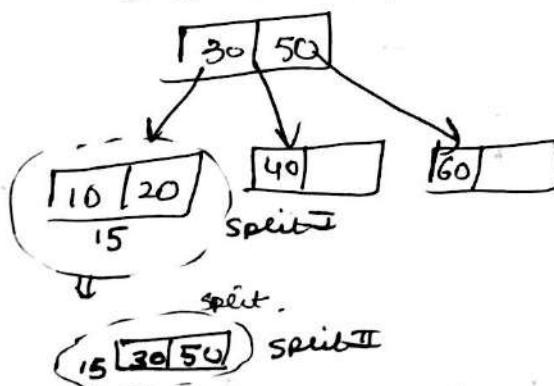
Creating 2-3 Trees
 In tree, tree is creating in upside direction
 $\text{keys} \rightarrow 20, 30, 40, 50, 60, 10, 15, 70, 80$



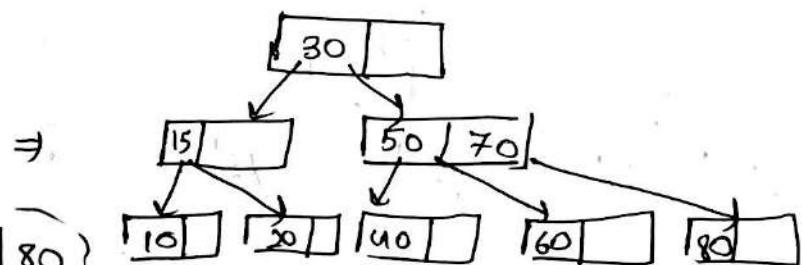
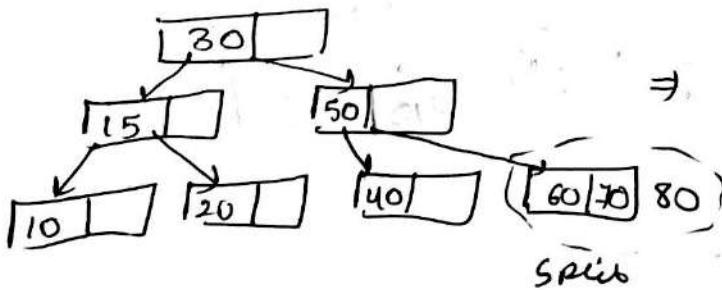
Insert 40
 No space in this node
 is degree 2
 but key 2
 can be stored
 in one node



Insert 10, 15



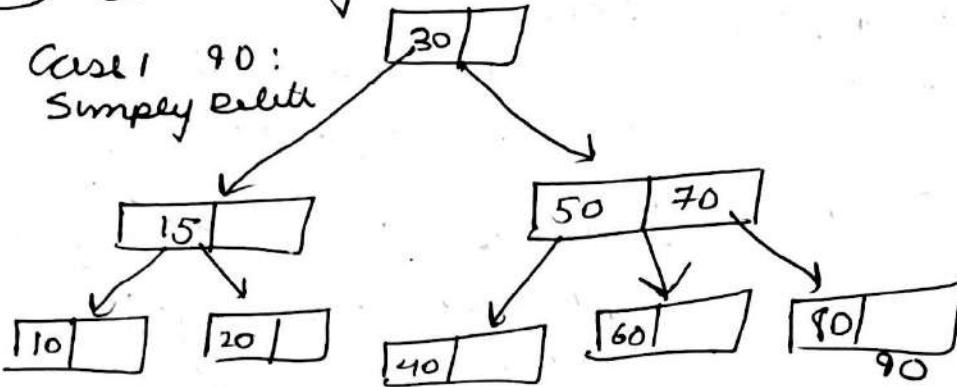
Insert 70, 80



2418

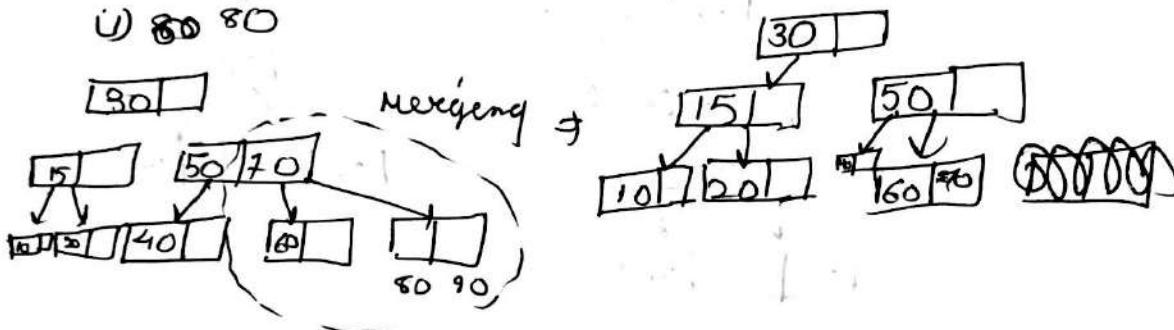
Deleting

case 2 90

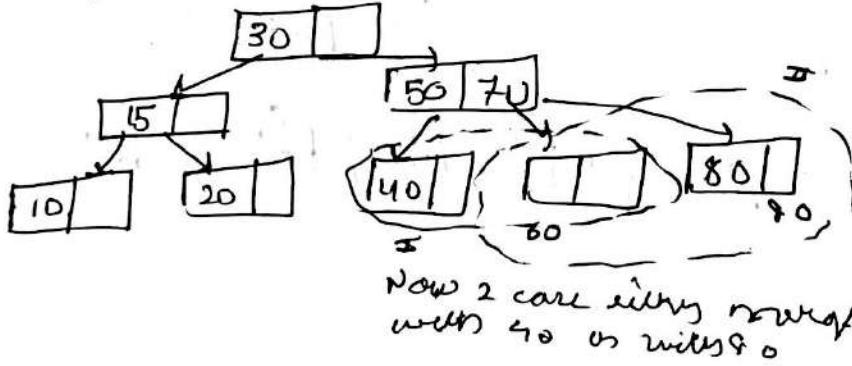
Case 1 90:
Simply delete

case 2: Delete and Merge -

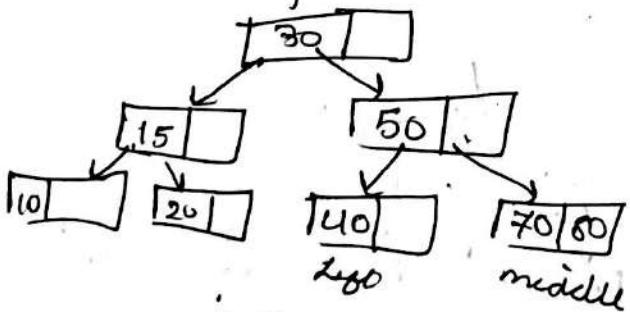
(i) 80 80



(ii) If we delete 60

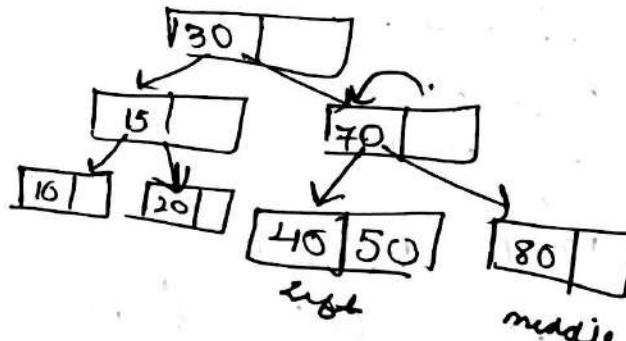


Reperforming II

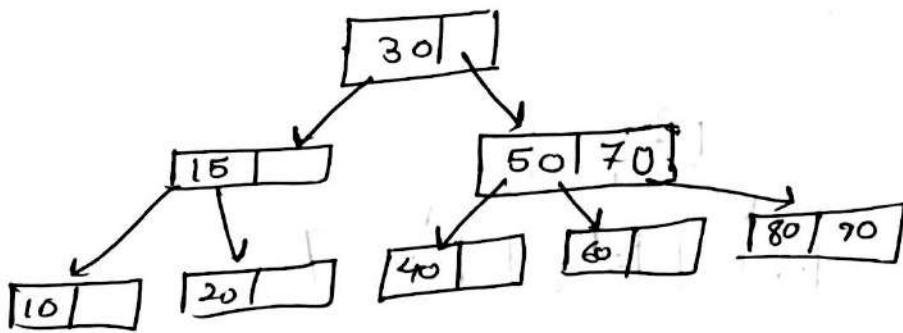


- changes done in I
- merging left & middle
- shifting 70 to front
- making 80 as middle child

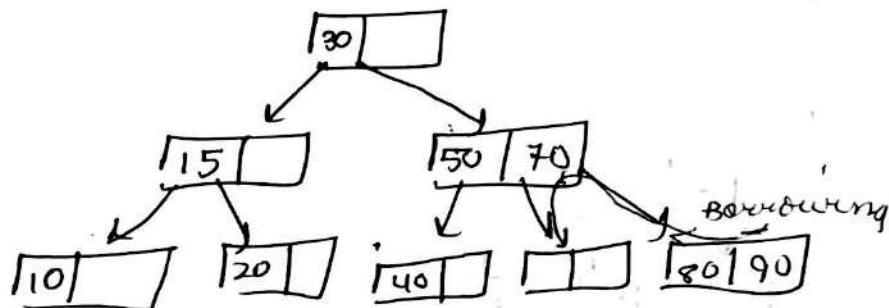
Reperforming I



Case 3 : Borrow



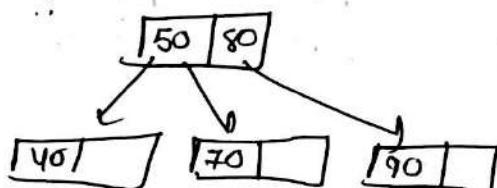
Delete 60



Now before merging there is option of Borrowing

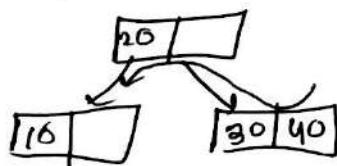
- ∴ left is not has only one key & right [80/90] has 2 key.. we can borrow.

Borrowing process happens through Parent.

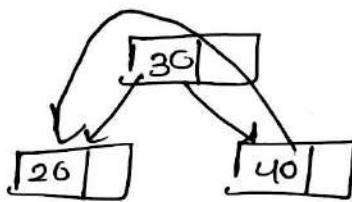


∴ First we see Borrowing. If Borrowing is Not Possible we Merge.

Ex of Borrowing



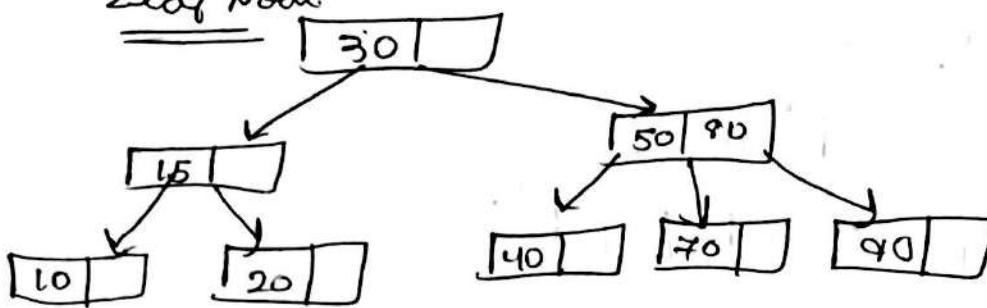
Delete 10



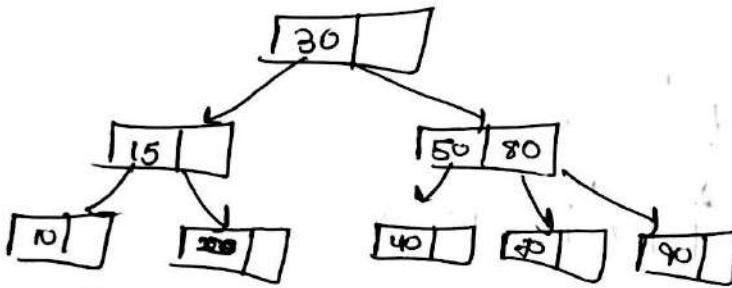
Case 4 Multiple Operations

244

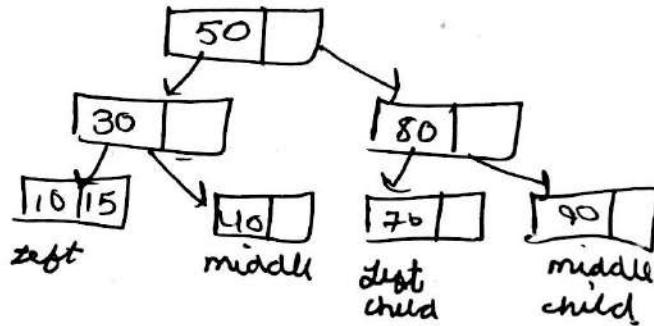
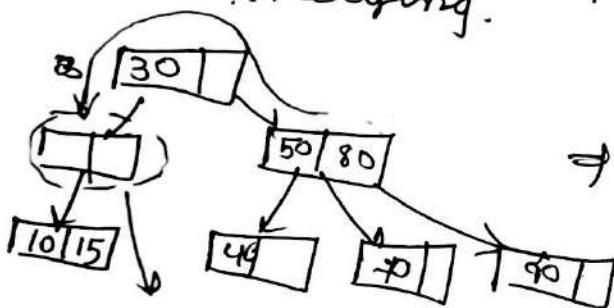
Leaf Node:



* Delete: 20

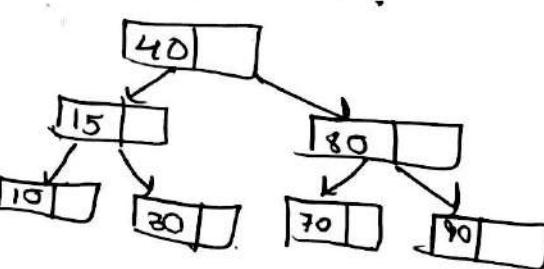
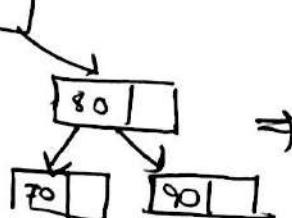
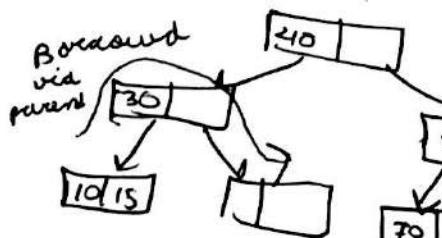


Non Borrowing: not possible.
∴ Merging.



∴ Non Borrowing is Possible
∴ Non Borrowing is done
∴ Left child of 50 is 40
will be at Preorder Precessor

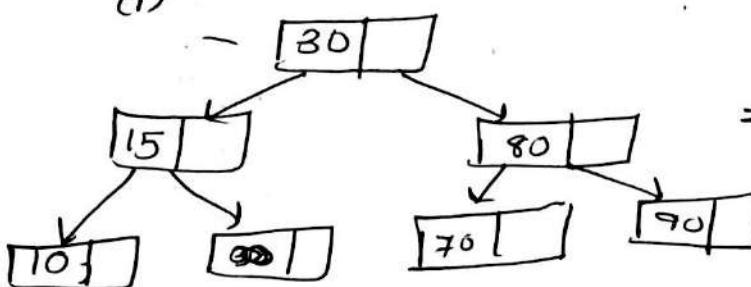
Now if we want to delete 50 (Non Leaf Node)
inorder predecessor will take its place
∴ either 40 or 70 Ans let take 40.



Let Delete 40

∴ Inorder Precesses: 30

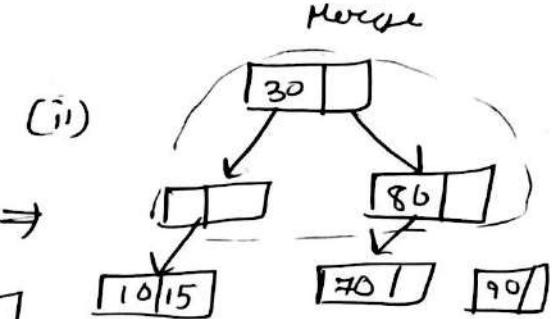
(i)



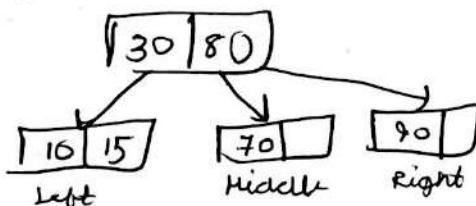
Borrowing not possible

∴ Merging

(ii)



(iii)



Why & How 2-3 Trees are used

Min Key & Min Nodes

examples
3 Degree Tree

Min Nodes

$$n = 1 + 2 + 2^2 + \dots + 2^h - 1 = 2^{h+1} - 1$$

$$\text{Max Keys} h = \log_2(n-1) - 1$$

$$\text{Max Keys} O(\log_2 n)$$

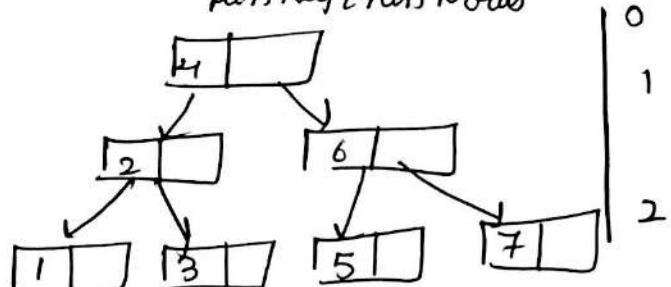
$$\text{Max Node} n = 1 + 3 + 3^2 + \dots + 3^h$$

$$\text{Max Node} = \frac{3^{h+1} - 1}{3 - 1}$$

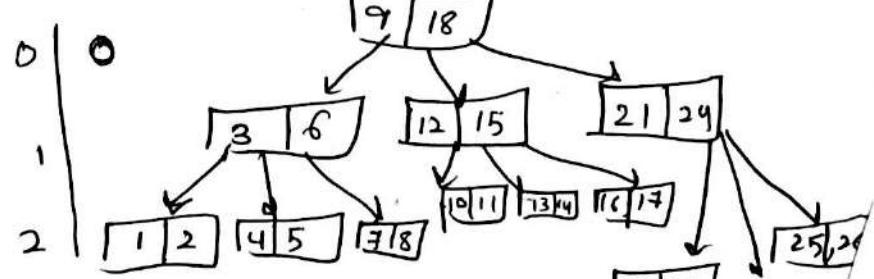
$$n = 3^{\frac{h+1}{2}} - 1$$

$$\text{Min } n = \log_3(n(3-1)+1) - 1$$

$$O(\log_3 n)$$



Max Keys and Max Node
at $h = 2$



Where it is used?

B-Tree or B⁺-Tree are used in DBMS internally

2-3 Tree is a type of B-Tree

∴ In searching when we move to a node

we can get more than one value in 2-3 tree. But in BST we get one value at one node

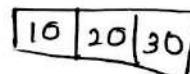
2-3-4 Trees

- B-Tree of degree = 4

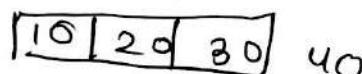
- Every Node must have $\left\lfloor \frac{4}{2} \right\rfloor = 2$ children

- All leaf at same level.

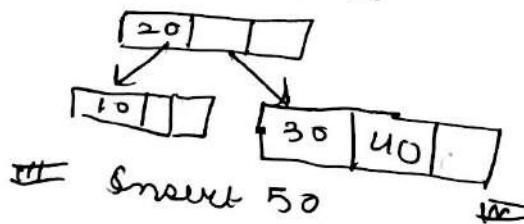
Keys $\rightarrow 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110$
 (i) Insert 10, 20, 30



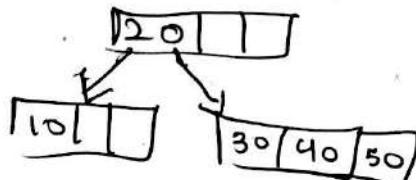
(ii) Insert 40



If we make right biased

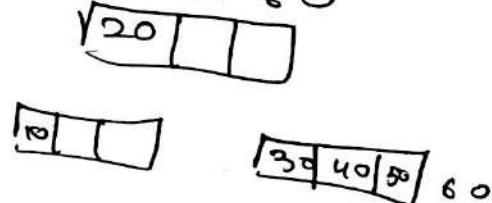


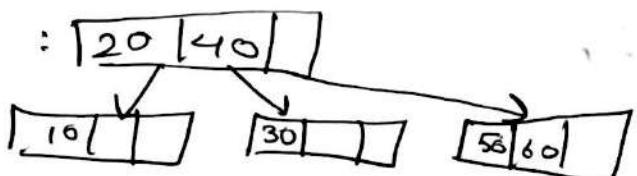
iii) Insert 50



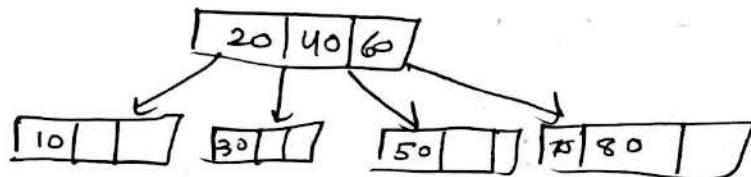
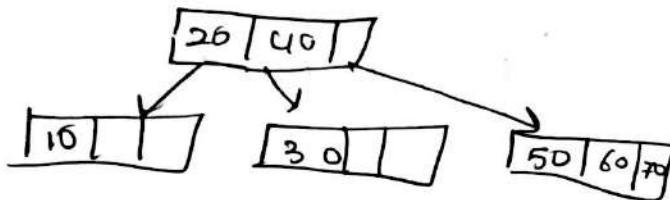
Now we will discuss
 We have even no of keys
 ∴ we have to be left or right biased

Insert 60

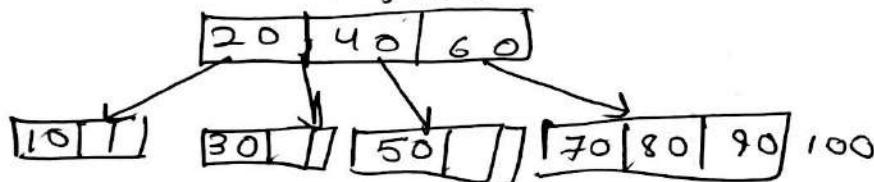




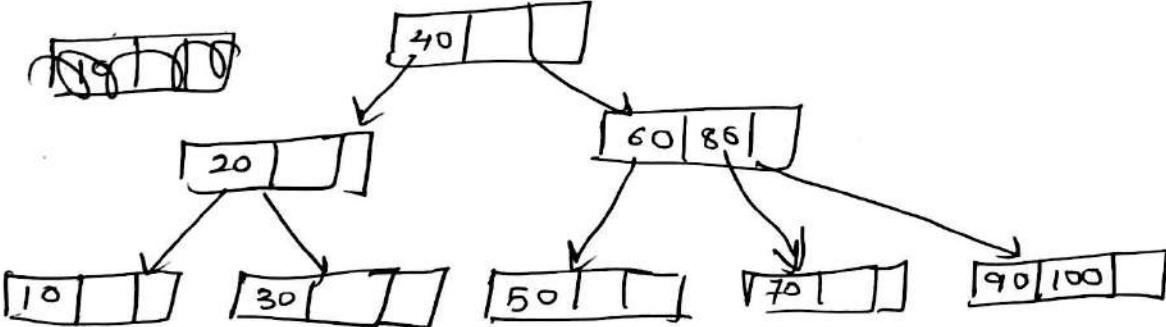
Insert 70, 80



Insert 90, 100

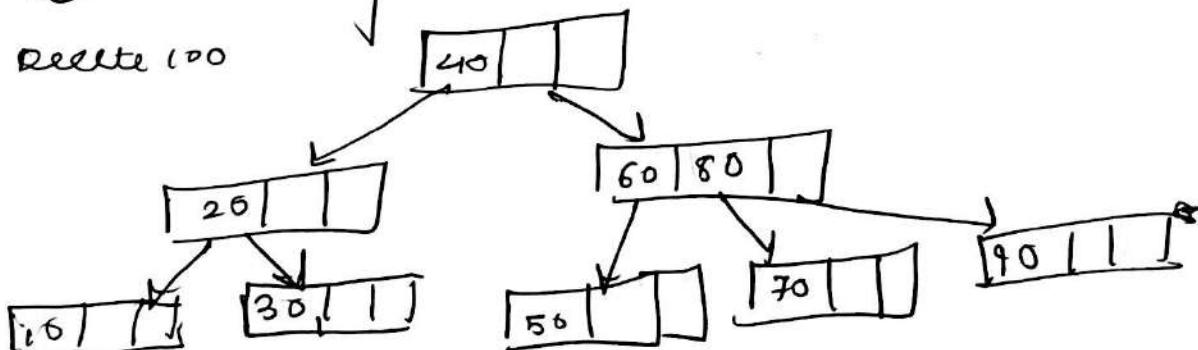


2 Levels of Splitting is Done.



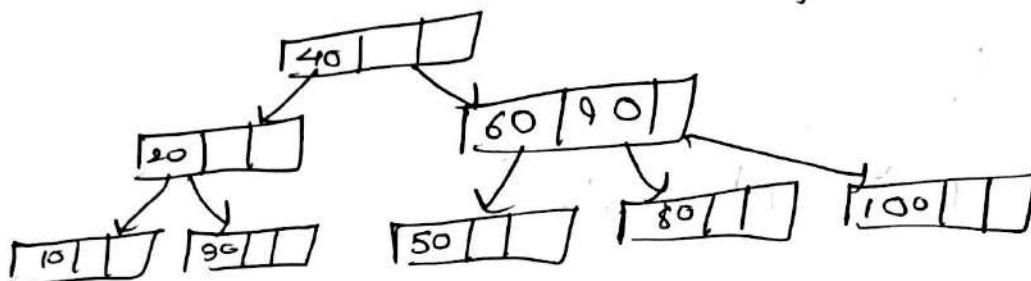
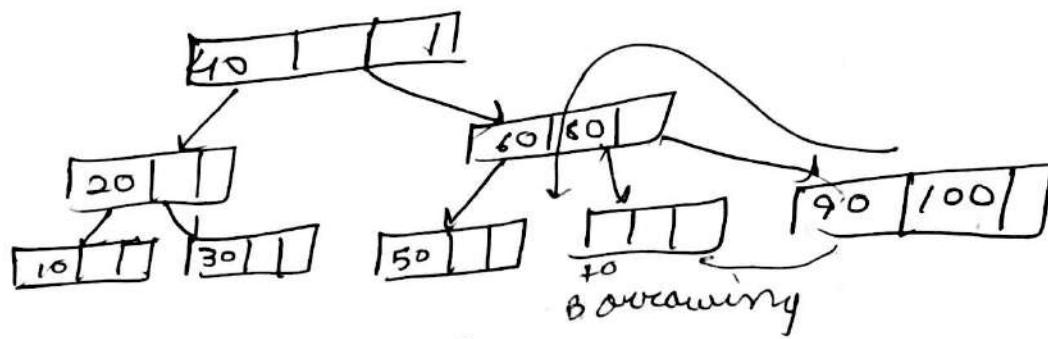
Deleting.

Delete 100

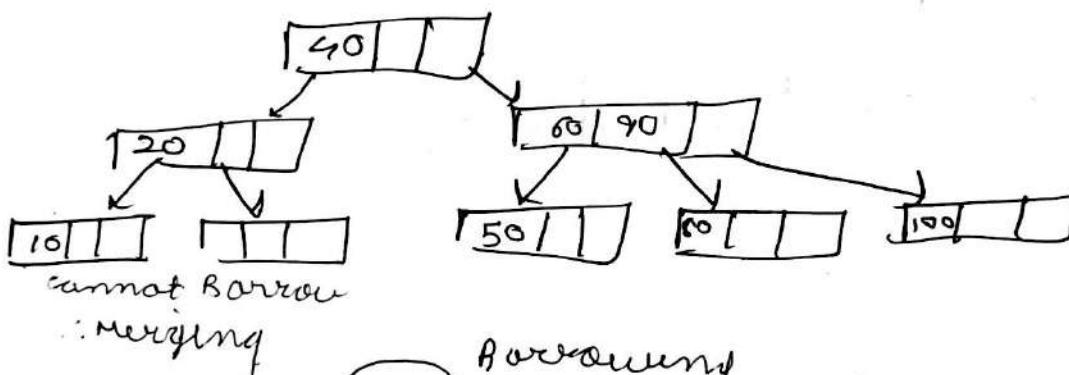


254

If we delete 70 in place of 10^6



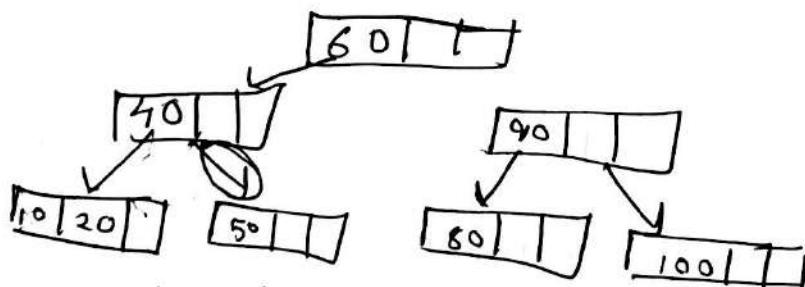
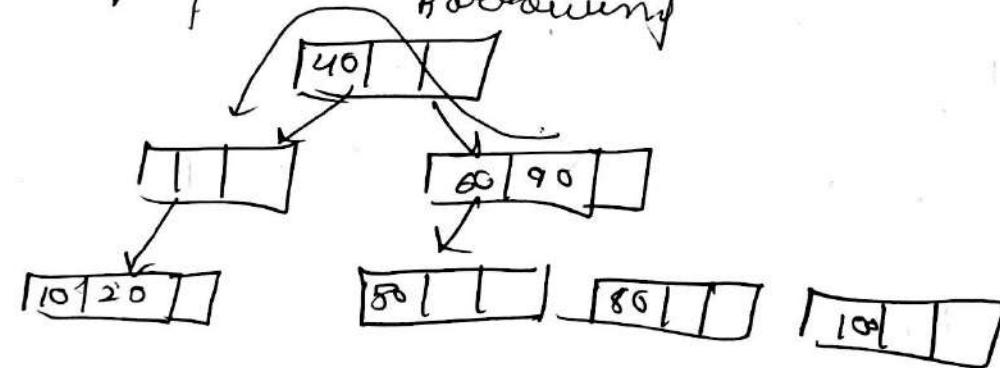
Delete 30



Merging

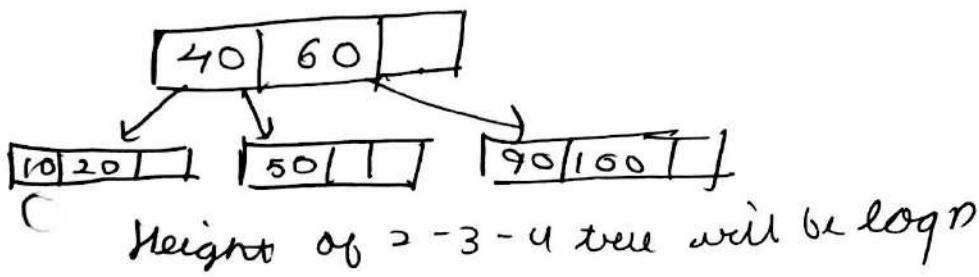
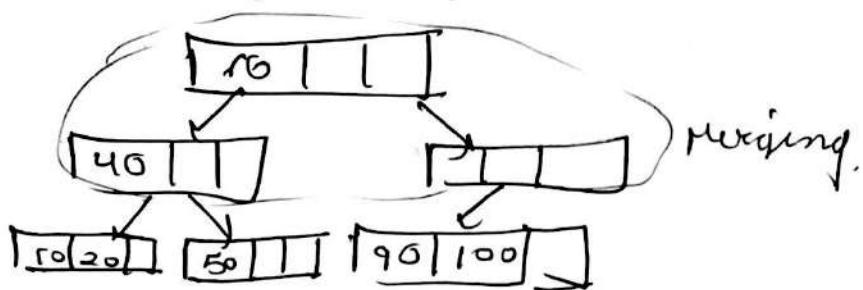
Arcs are shown between the bins of sizes 10, 20, 50, 60, 90, 80, and 100.

Borrowing



60's left child now became
middle child of 50's

Delete 80



Red - Black Trees

→ Properties & Creating Red Black Tree

- (i) It's a Height Balanced Binary Search Tree, similar to 2-3-4 Tree.
- (ii) Every Node is either Red or Black.
- (iii) Root of Tree is always Black.
- (iv) No 2 consecutive Red, Parent and children of Red are Black.
- (v) Number of Black on Paths from Root to Leaf are same.
- (vi) New Inserted Node is Red.
- (vii) Height is $\log n \leq h \leq 2\log n$
AVL trees are more strict.
 $\log n \leq h \leq 1.44 \log n$
- (viii) Red Black Tree is less strict than AVL

256

Creating Red - Black Tree

(more similar to BST + AVL) ... relations are also true

Case I

Uncle is Red \Rightarrow Resolution

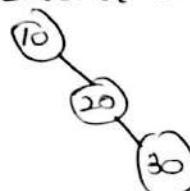
Insert 10



Insert 20



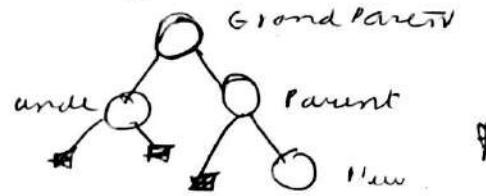
Insert 30



when there is Colours
Conflict 2 cases:

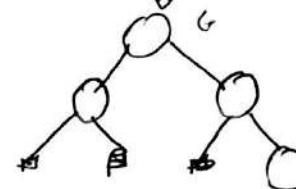
Case I

Uncle is Red \Rightarrow Resolution



Here Parent &
uncle both are
or Red

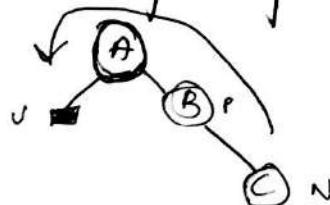
\Downarrow converted into



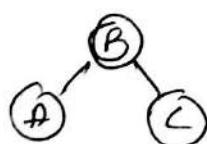
* if G is Node
then it
will be
converted
into Black

Both uncle & Parent
are converted into Black

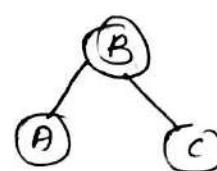
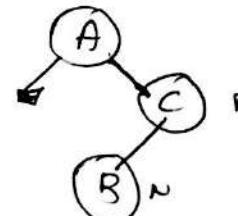
LR/RR zig-zig



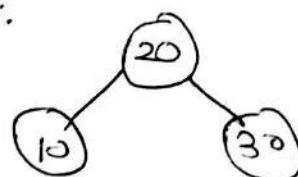
parent Red Uncle Black
thus perform rotation



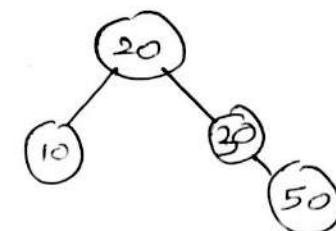
LR/RL zig-zig



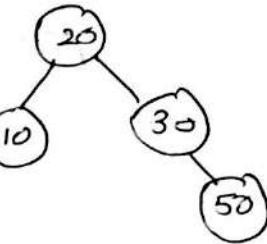
: On Insert 30
we will do rotation
 \therefore



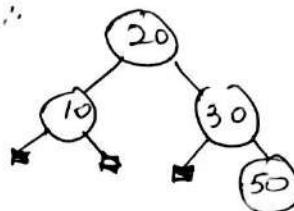
Insert 50



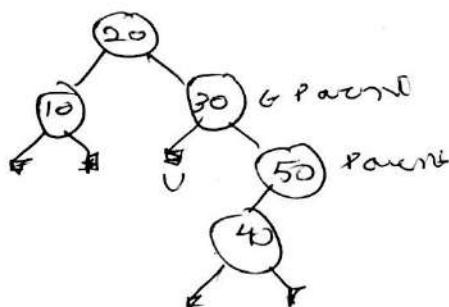
\therefore colour change \Rightarrow



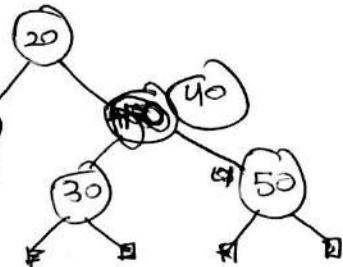
\because Node should be black



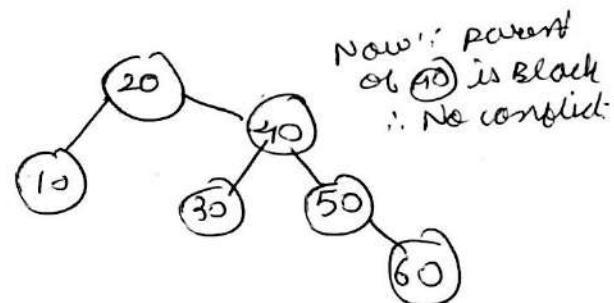
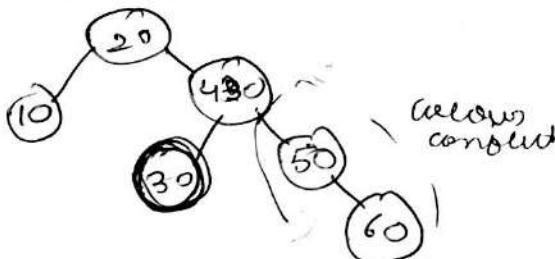
Insert 40



\therefore Rotation RL \Rightarrow

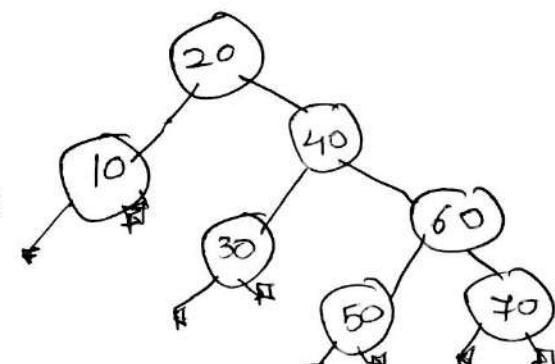
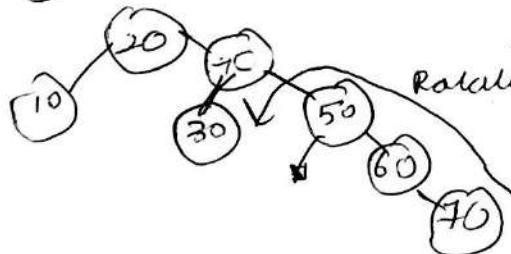


Insert 60



Red-Red \therefore Recolor

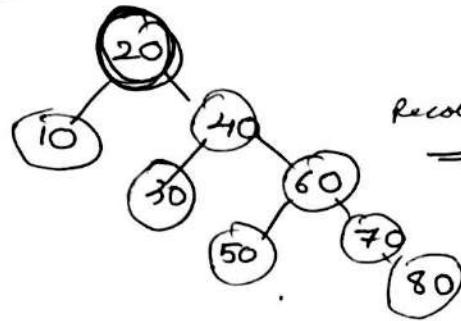
Insert 70



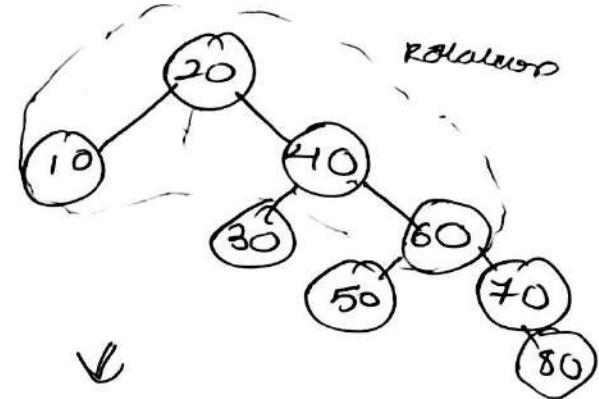
No of Black in every pair is same i.e. 3

258

now Insert 80

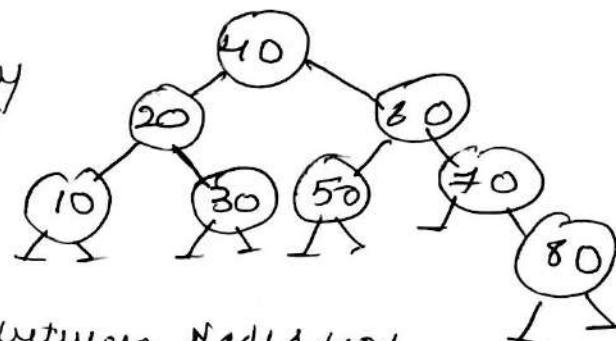


recoloring →



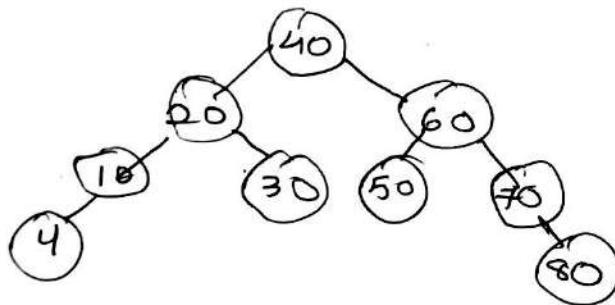
rotation

∴ we did Recoloring
then Rotation.

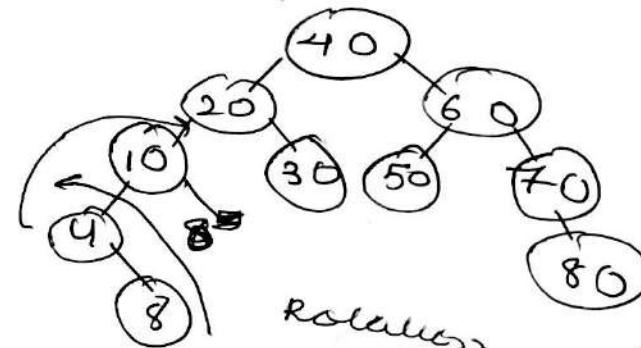


3 Black's between Nodes way
Balanced.

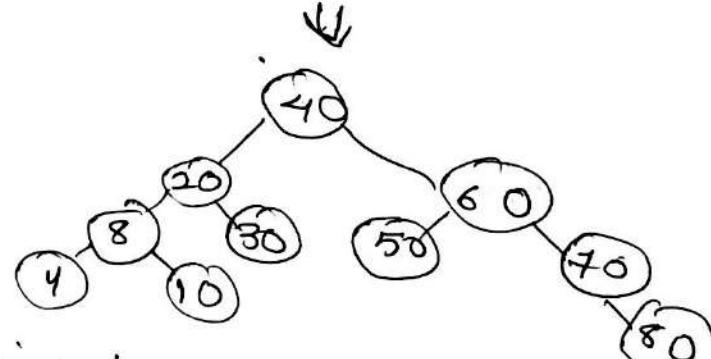
Insert 4



Insert 8



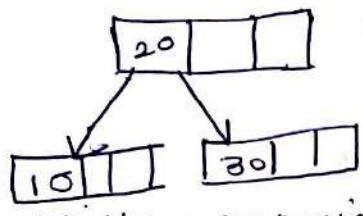
rotation



Height:

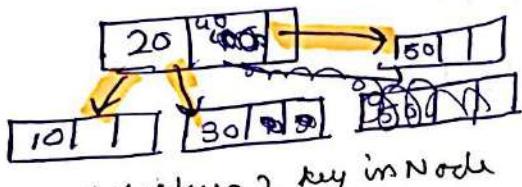
Removing Red Logn
Including Red Logn
 $\approx \log n$

2-3-4 Tree

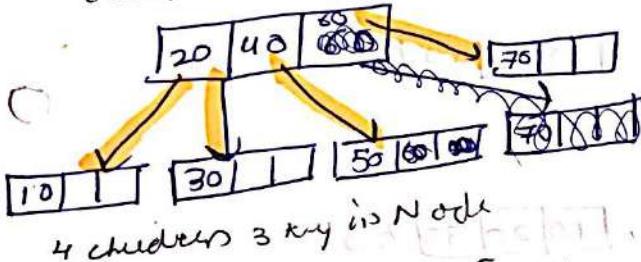


We are
not inserting,
just comparing
RHS & LHS -

2 children one key in Node

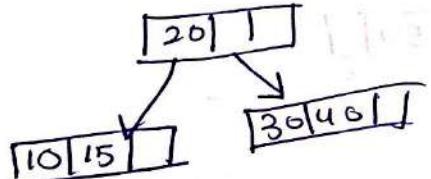


3 children 2 key in Node



4 children 3 key in Node
2-3-4 children are passed

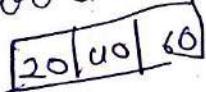
∴ 2-3-4 Tree



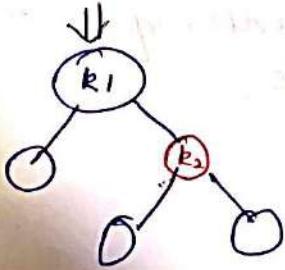
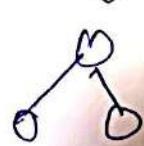
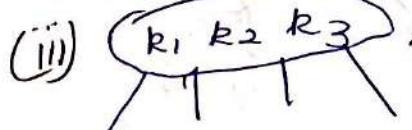
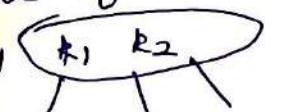
related to ~~stand~~ its parent black colour node
∴ 15 means $\boxed{10 \ 15 \ 1}$

∴ it means red colour and
black colour node

means

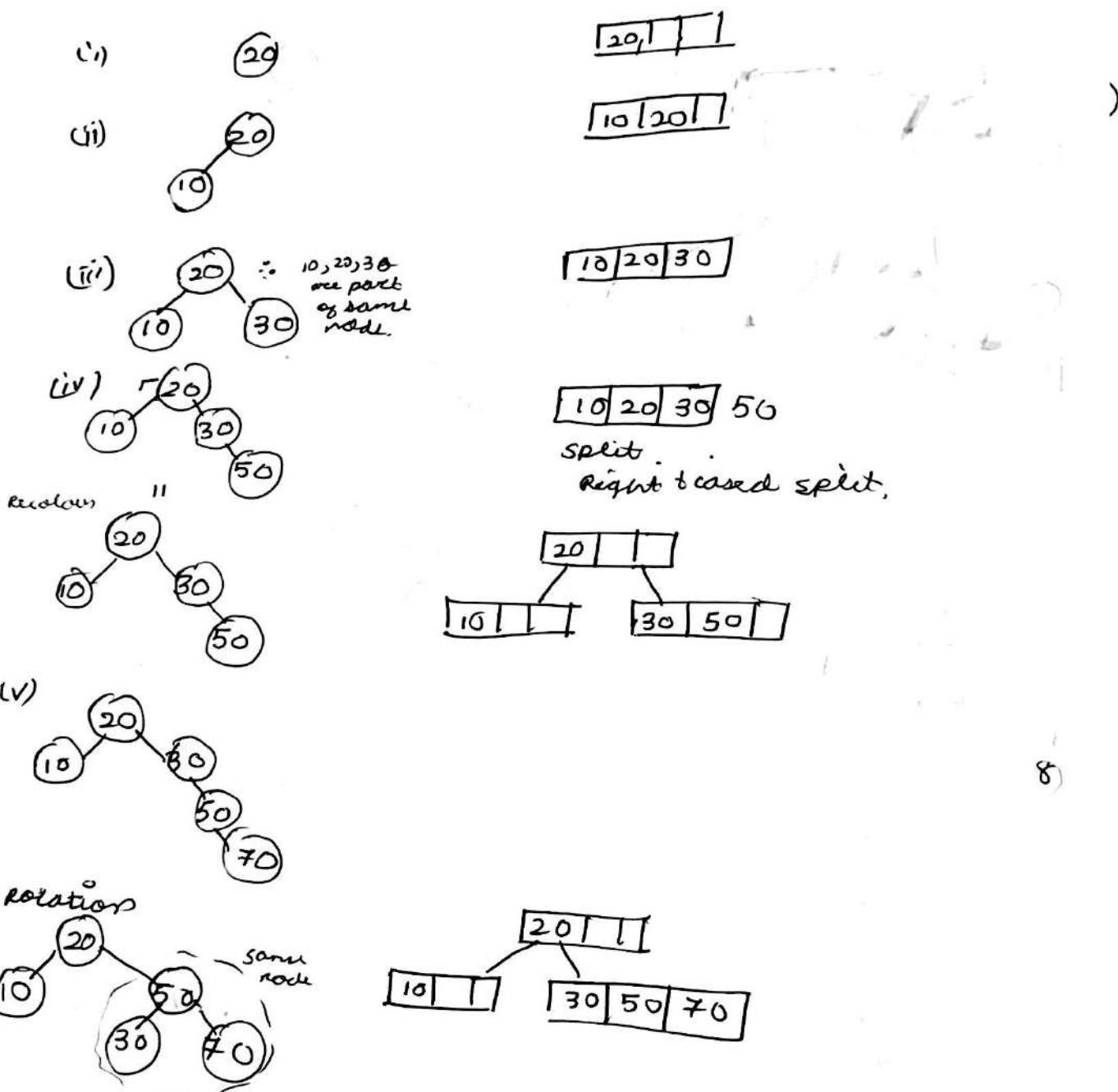


i.e like 2-3 Nodis
General formula



260 Creating Red Black Tree similar to
Creating 2-3-4 Tree

keys : 10, 20, 30, 50, 70, 60, 80, 4, 8



∴ we can see creating 2-3-4 Tree & Red Black Tree is similar.

Red Black Tree Deletion

on Binary Search Tree

so we want to delete any node

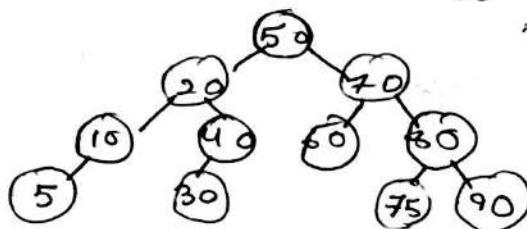
ex 30 we simply Delete

if 20 then we will replace

the node value at that node

by Inorder Precessor or

Inorder Successor & will
thus delete the inorder Pre-
cessor or Successor node



so 30

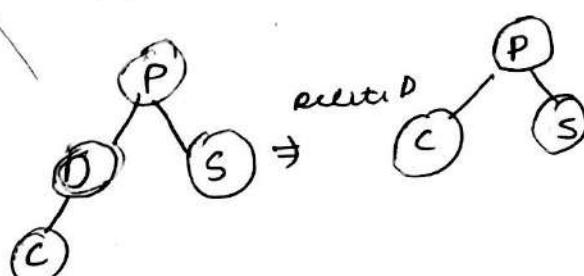
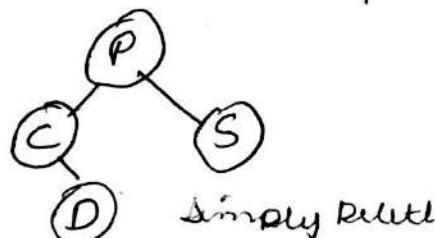
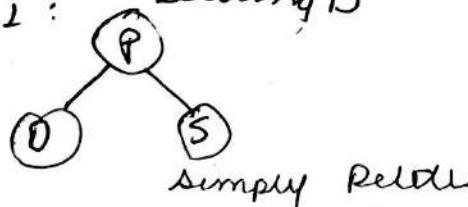
if we delete 20, 10 will take its place
& node of 10 will be deleted :-.

Node Deleted is always either Leaf
Node or Node having 1 children.

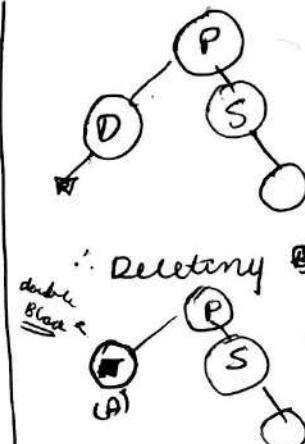
we focus about Deletion of that
Node is Red Black.

Deleting Red Node

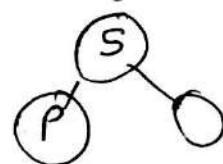
Case 1 :



Deleting Black Node



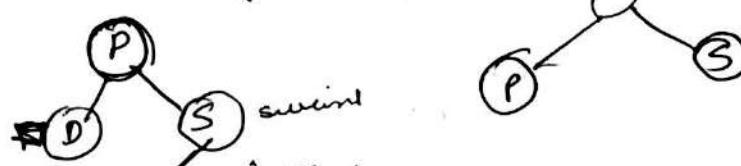
Now when it is Double
Black see Sibling
of Red
Perform Rotations.



It will wrong
combination

262

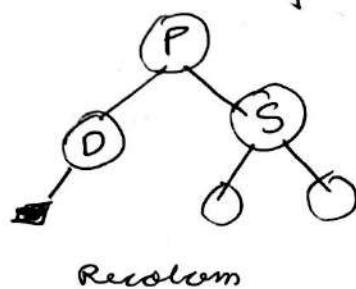
cases : Siving Black, children red



swin

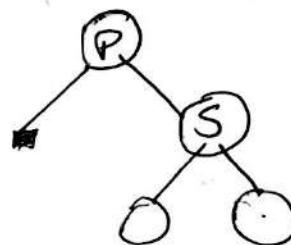
zig zag
rotation

Siving Black & this chickens are also Blad.

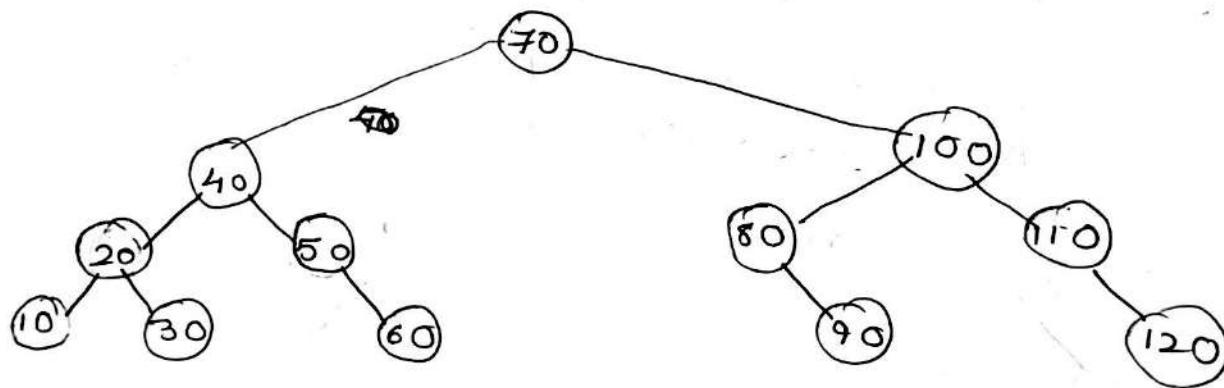


Recolor

make swin
red &
Parent
Black
& delete
D.



Red Black Tree Deletions Examples



Delete 90

Leaf Node & Red :: delete simply

Delete 100

Grands Precessor : 90

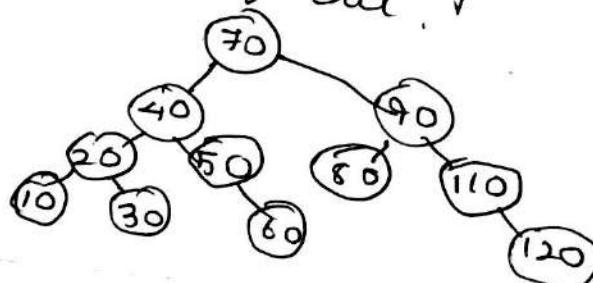
Grande Successor : 110

Let us take Grands Precessor

90 is shifted at 100 places

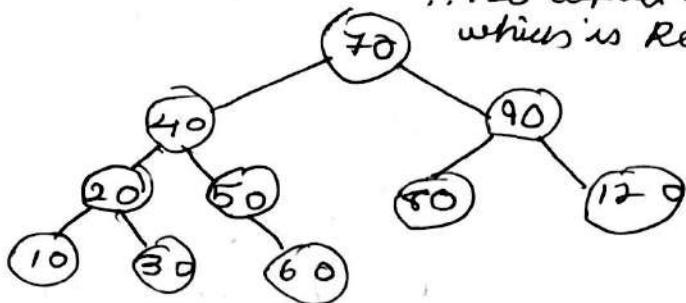
& 90 is simply Deleted :: red colors

& Leaf Node.



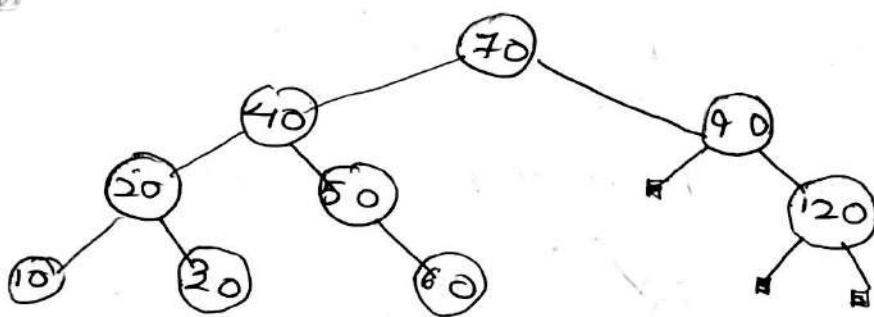
Delete 110

263
120 will be in order successor
∴ 120 copied to 110 & delete 120 Node
which is Red ∴ can be deleted Directly



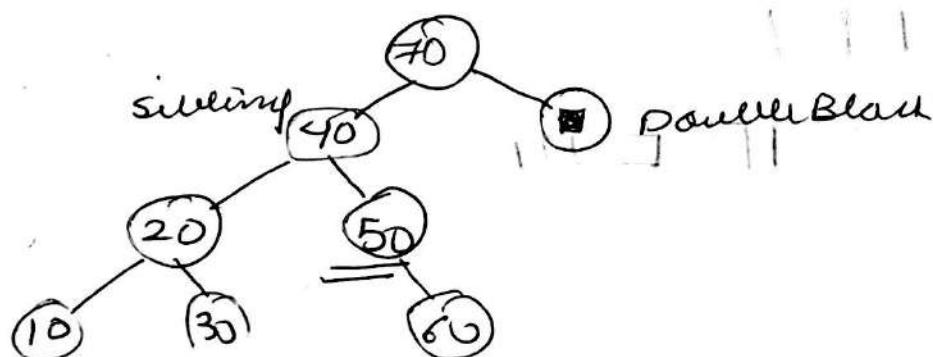
Now Delete 80

∴ No one is there to take place of 80 ∴ Double Black
will be generated. Now sibling of 80 is 120
& its colour is Black & its children are Black
(i.e. null)

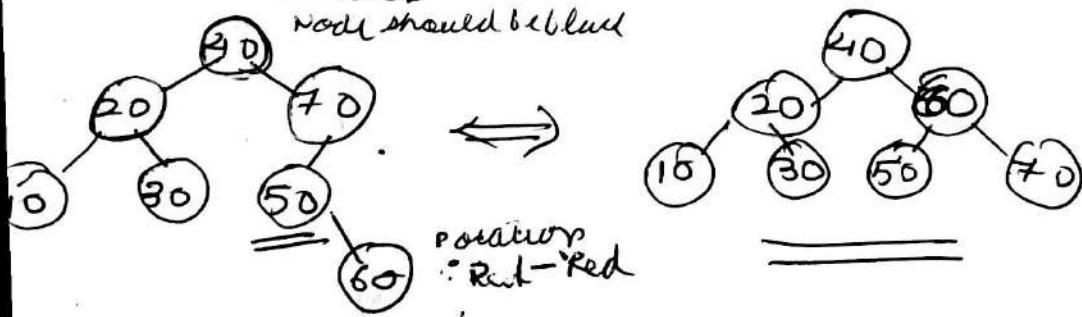


Delete 120: It will be directly deleted.

Now if we delete 90



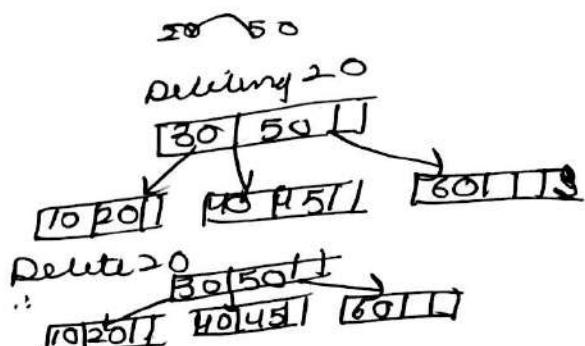
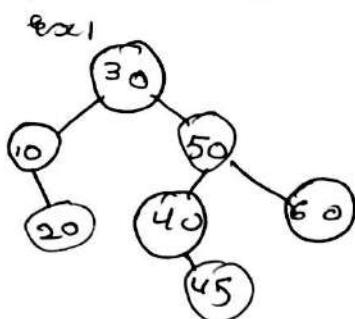
rotation
node should be black



No colour of Node is changed except right child of 40
i.e. 50 colour changes to red. i.e. child which
went to other side has changed colour to Red.

Red Black Tree vs 2-3-4 Tree

Deletions

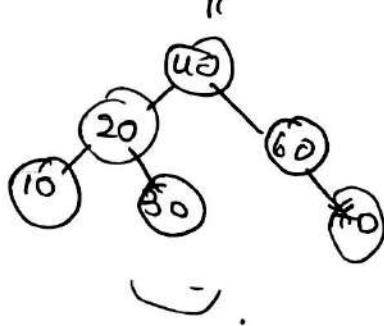
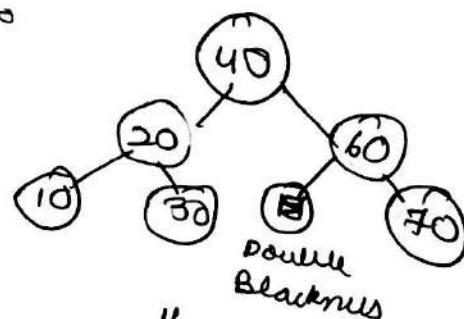
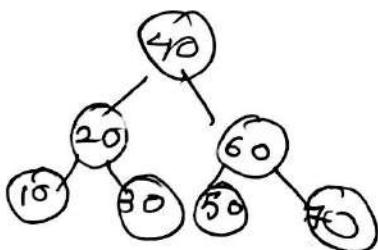
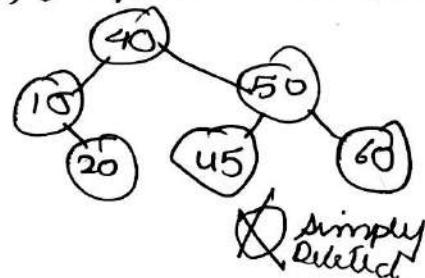
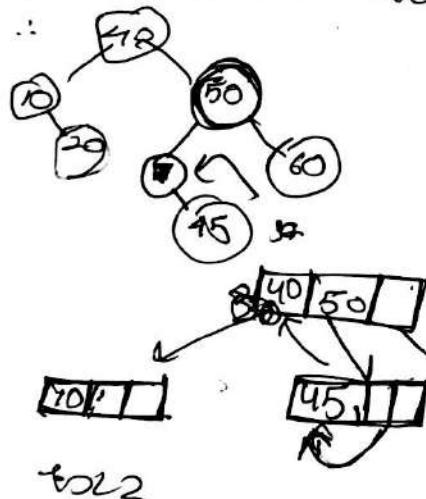


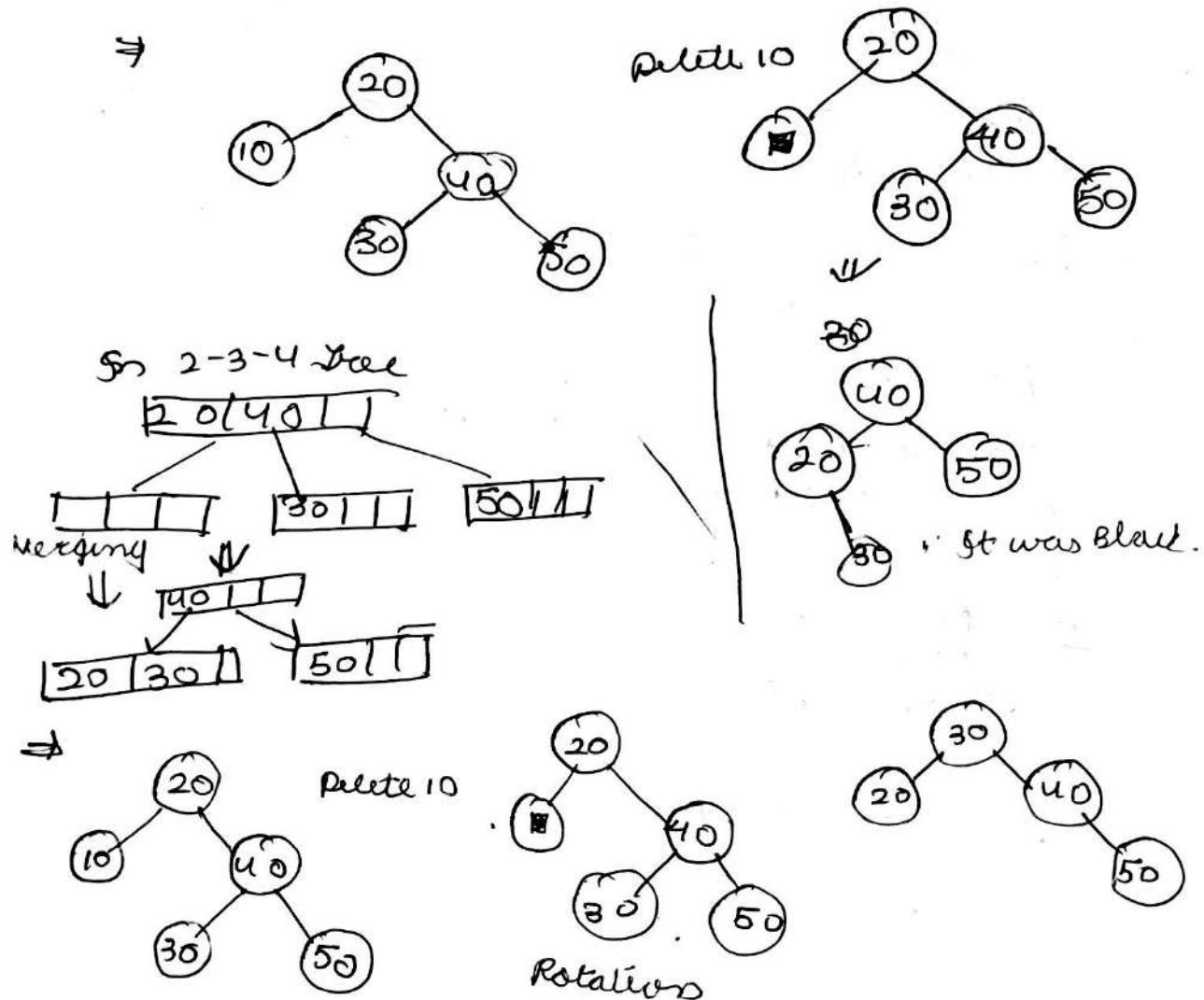
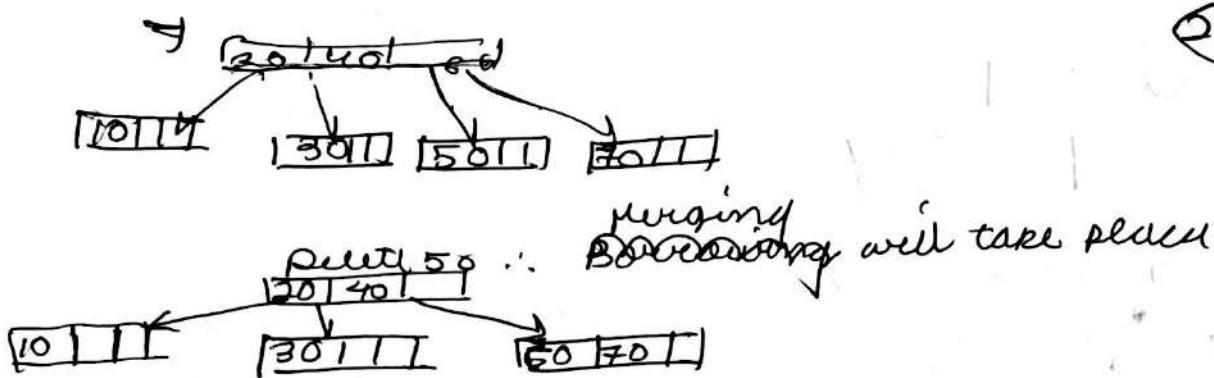
Delete 20
directly will be deleted.

if we want to delete

30
it needs promotion will take its place i.e 20 & 20 value
node will be directly deleted.

→ So we take it needs promotion success than 40



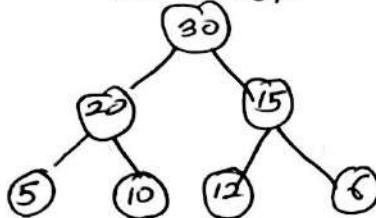


Heap

What is Heap?

I It is complete Binary tree (or Almost Complete)

Max Heap



T	30	20	15	5	10	12	6
	1	2	3	4	5	6	7

Acc'ng to:

Node at index i

L child at $2 \times i$

R child at $2 \times i + 1$

This is complete i.e no gap b/w array.

here index is starting
from ~~0~~ 1

II Max Heaps

Possibility I:

every N

Max Heap: every Node is greater or equal to its descendants.

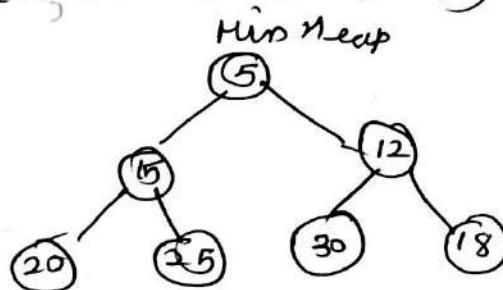
Possibility II:

Min Heap: every Node is smaller or has its descendants.

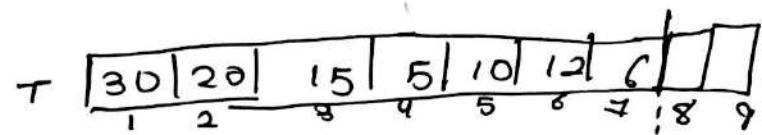
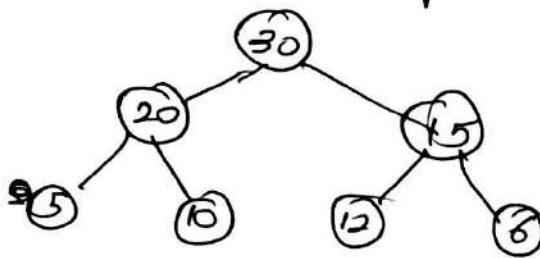
∴ Heap should satisfy either Max or Min Heap.

III Height of tree will be always $\log n$

∴ height of complete tree is also $\log n$.

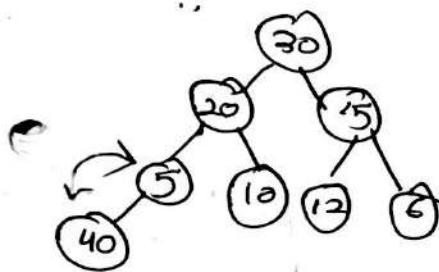


Inserting in a Heap

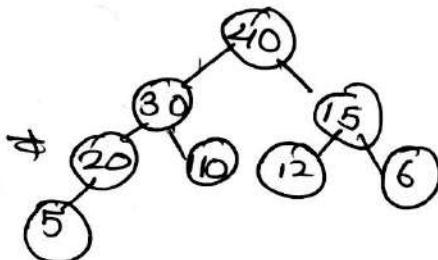
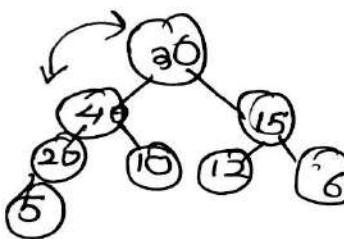
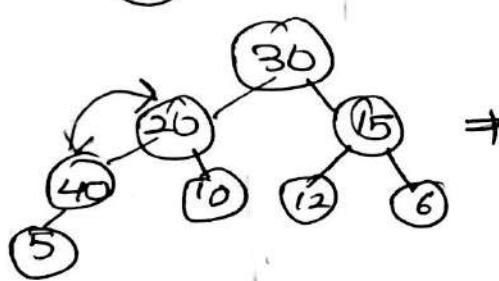


Insert 40

30	20	15	5	10	12	6	40
----	----	----	---	----	----	---	----



Now 40 will be compared with its parents if greater then swap



∴ Now Heap satisfies Max Heap condition

In array we understand as:

30	20	15	5	10	12	6	40
1	2	3	4	5	6	7	8

comp. with parent
∴ parent is smaller, swap

30	20	15	40	10	12	6	5
1	2	3	4	5	6	7	8

30	40	15	20	10	12	6	5
1	2	3	4	5	6	7	8

40	30	15	20	10	12	6	5
1	2	3	4	5	6	7	8

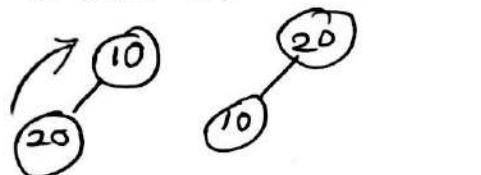
268

Now insert 35 same way.

Creating Heap from given Array

A	1						
#	10	20	80	25	5	40	35
	, 1	, 2	, 3	, 4	, 5	, 6	, 7

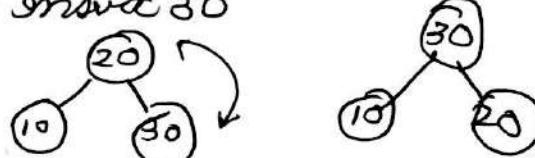
② Insert 20



B	20	10	30	25	5	40	35
	, 1	, 2	, 3	, 4	, 5	, 6	, 7

size of heap

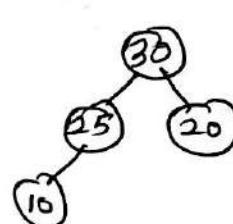
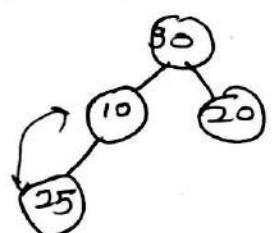
③ Insert 30



C	30	10	20	25	5	40	35
	, 1	, 2	, 3	, 4	, 5	, 6	, 7

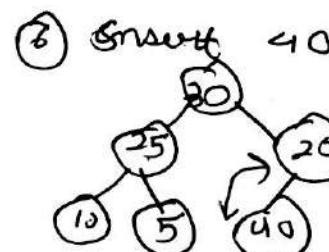
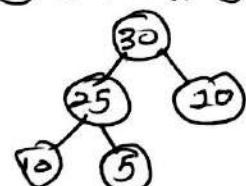
size

④ Insert 25

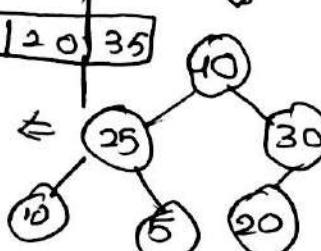


D	30	25	20	10	5	40	35
	, 1	, 2	, 3	, 4	, 5	, 6	, 7

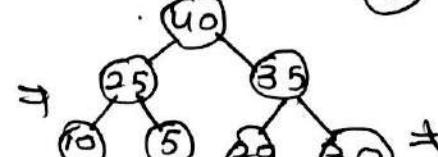
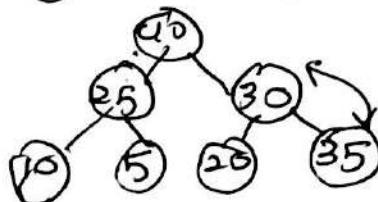
⑤ Insert 5



E	40	25	30	10	5	20	35
	, 1	, 2	, 3	, 4	, 5	, 6	, 7



⑥ Insert 40



F	40	25	30	10	5	20	35
	, 1	, 2	, 3	, 4	, 5	, 6	, 7

Functions for insert

```
void insert ( int A[], int n )
{
    int temp, i = n;
    temp = A[n];
    while ( i > 1 && temp > A[i/2] )
    {
        A[i] = A[i/2];
        i = i/2;
    }
    A[i] = temp;
}
```

void CreateHeap()

```
{ int A[] = { 0, 10, 20, 30, 25, 5, 40, 35 };
```

0 1 2 3 4 5 6 7
int i;

```
for ( i = 2; i < 7; i++ )
```

```
{ insert ( A[i] ); }
```

```
}
```

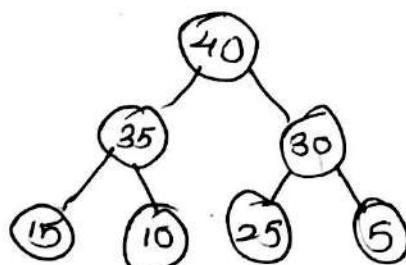
~~For one element insertion
in heap takes $\log n$~~

$\therefore n$ elements insertion = $n \log n$
 $\therefore O(n \log n)$

For min Heap we just have to replace
condition $temp < A[i/2]$

Deleting From Heap & Heap Sort.

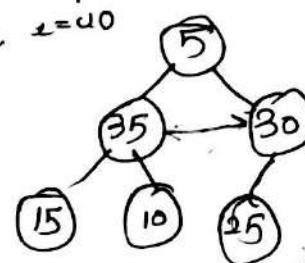
In Heap we can only delete largest element.



0 Delete 40

This last element of heap
will be shifted to the
top

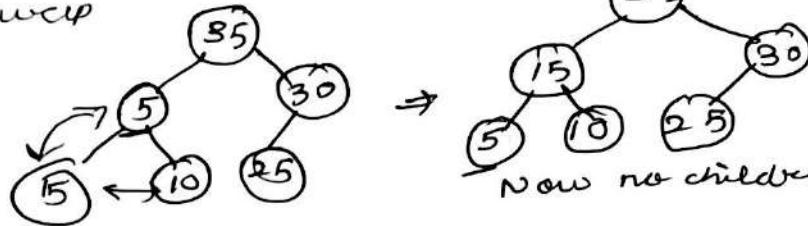
$\therefore z = 40$



Comparing which
is greater
& thus comparing
greater among them with
its parent

270

now 35 is greater & also greater than 5
 ... swap



In Array

5	35	30	15	10	25
1	2	3	4	5	6

35	15	30	15	10	25
1	2	3	4	5	6

child Parent

of 4 & 5

compare

no children

Heap Sort

as we delete 10 and store at last empty place of array.

i.e

35	15	30	5	10	25	40
----	----	----	---	----	----	----

Now Delete 35

30	-	-	-	35	40
----	---	---	---	----	----

Heap size

Heap size.

Now Delete 30

25	-	-	30	35	40
----	---	---	----	----	----

Heap size.

∴ as we see on deleting from
 heap the rest of array outside heap
 is becoming sorted array.

∴ Heap Sort:

- 1) Create Heap of 'n' elements
- 2) Delete n element 1-by-1

∴ Heap Sort: $O(n \log n)$

#include <stdio.h>

void insert(int A[], int n)

```

{ int i=n, temp;
  temp = A[i];
  while (i > 1 && temp > A[i/2])
  {
    A[i] = A[i/2];
    i = i/2;
  }
  A[i] = temp;
}

```

int delete (int A[], int n)

```

{ int i, j, x, temp, val;
  val = A[1];
  x = A[n];
  A[1] = A[n];
  i = 1; j = i * 2;
  A[n] = val;
  while (j <= n-1)
  {
    if (A[j] > A[i])
      if (j < n-1) // New added
        if (A[j+1] > A[i])
          i = j + 1;
    else
      break;
  }
}

```

if (A[i] < A[j])

```

{ temp = A[i];
  A[i] = A[j];
  A[j] = temp;
  i = j;
  j = 2 * i;
}

```

else
break;

(270) returns val;

```

}
int main()
{
    int H[] = { 0, 10, 20, 30, 25, 5, 40, 35 };
    // 40, 25, 35, 10, 5, 20, 30

    int i;
    for (i=2; i<=7; i++)
        Insert(H, i);

    return for (i=7; i>1; i--)
    {
        Delete(H, i);
        for (i=1; i<=7; i++)
            printf("v.d", H[i]);
        printf("\n");
    }
    return 0;
}

```

sorted array

⇒ Heaps - Faster Method for Creating heap.

Create Heap

we go from elements to its parent i.e move in upward direction for swapping

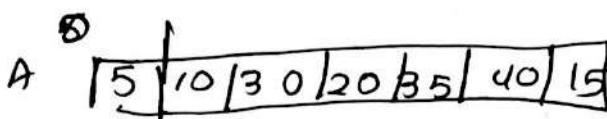
Delete Heap.

we delete top most and then move downwards and check whether and swap.

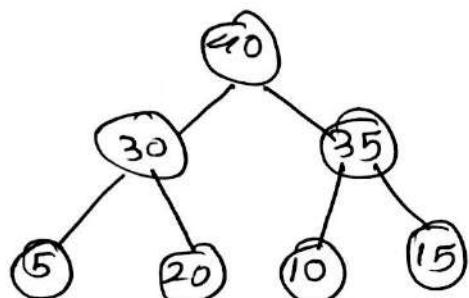
In Heapsify

we will go & check in downward direction
ex:

Create Heap

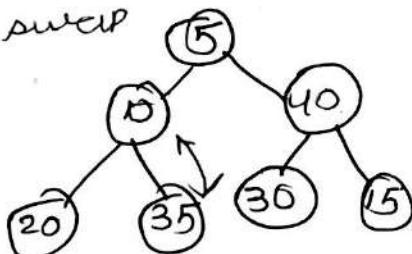


now A is as creating heap

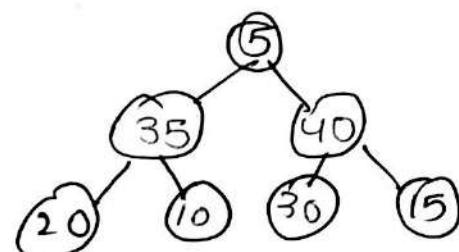


check child of 30 :

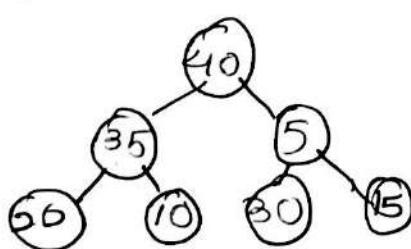
: swap



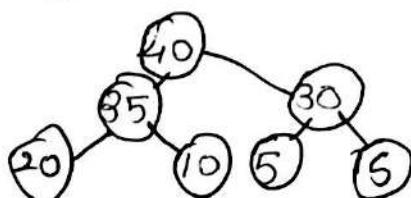
check child of 10
rchild > lchild
 $10 < 20$
: swap



check child of 5
rchild > lchild.
 $5 < 10$
: swap



check child of 5
lchild > rchild
 $5 < 10$
: swap



Max Heap

(It is different from what is created by CreateHeap fun.)

this is $O(n)$ Fastest Method

Binary Heap & Priority Queue

elements $\rightarrow 4, 9, 5, 10, 6, 20, 8, 15, 2, 18$

Larger the element
larger the priority

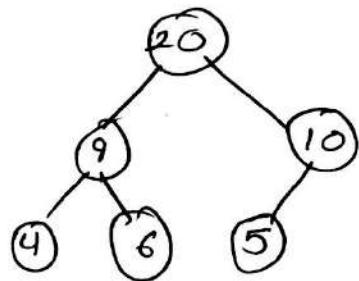
4	9	5	10	6	20	8	15	2	18
---	---	---	----	---	----	---	----	---	----

now: for a element

Insert: $O(1)$

Delete: $n + n$ search shifing $O(n)$

Make Max Heap for the elements.



insert: $\log(n) \Rightarrow O(\log n)$

Delete: Root element is deleted

\because max priority is less of Root

$\therefore \log n$

\therefore Delete: $O(\log n)$

\therefore Heap is the best suitable data structure for implementing ~~the~~ Priority Queues.

If we take small elements & higher priority then use MIN. HEAP

Sorting Techniques :

Criteria for Analysis

- 1) No of comparisons
- 2) No. of swaps
- 3) Adaptive

Comparison based sort.

when an array is sorted & we apply sorting algorithm
if it takes min time.
then we call it Adaptive Algorithms.

1) Bubble	O(n ²)
2) Insertion	
3) Selection	
4) Heap	O(n log n)
5) Merge	
6) Quick	
7) Tree	
8) Shell	O(n ^{3/2})
9) Count	
10) Bucket/Bins	Space based sort.
11) Radix	

- 4) Extra Memory

(extra memory required for sorting)

- 5) Stable

Name → A B C D E F G	Name is sorted
Marks → 5 8 6 4 6 7 10	

When we sort wrt Marks

Name → D	A	C	E	F	B	G
Marks → 4	5	6	6	7	8	10

∴ Stable ~~Means~~ Means if there are any duplicate elements in original list then in sorted list their order must be preserved

i.e. In sorted list based on marks,

C is ahead of E as in duplicate list.

(276)

Bubble Sort:

$$n = 5$$

4	8	5	7	3	2
0	1	2	3	4	

2nd Pass1st Pass

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

4 comp
4 swap

5	1	5	5	5
7	7	7	3	3
3	3	7	7	2
2	2	2	2	7
8	8	8	8	8

3 comp.
3 swap (max)3rd Pass

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

2 comp.
2 swap (max)4th Pass

3	2
2	3
5	5
7	7
8	8

1 comp
1 swap (max)No of Passes : $4 - (n - 1)$ passesNo of comp : $1 + 2 + 3 + 4 \rightarrow$

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

Max No of swaps : $4 + 3 + 2 + 1 = \frac{n(n-1)}{2} = O(n^2)$ In first pass we get $\frac{n-1}{2}$ largest elementsIn 2nd Pass we get 2 largest elementsIn kth Pass we get k largest elements.

```

void BubbleSort(int A[], int n)
{
    int flag;
    for (i = 0; i < n - 1; i++)
    {
        flag = 0;
        for (j = 0; j < n - 1 - i; j++)
        {
            if (A[j] > A[j + 1])
            {
                swap(A[j], A[j + 1]);
                flag = 1;
            }
        }
        if (flag == 0) break;
    }
}

```

Adaptive

1st Pass

i → 2	2	2	2	2
3] j → 3	3	3	3
5	5] j → 5	5	
7	7	7] j → 7	7
8	8	8	8	

comp = $n - 1$
 swap = 0
 loopbreak.
 $O(n)$

$O(n)$

∴ Bubble sort time $O(n^2)$

∴ we made Bubble Sort Adaptive by introducing flag variable

BY NATURE BUBBLE SORT IS NON ADAPTIVE
 WE MADE IT ADAPTIVE & it is known as
 Adaptive Algorithms

Stable ↗

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

∴ ⑧ will below ⑧

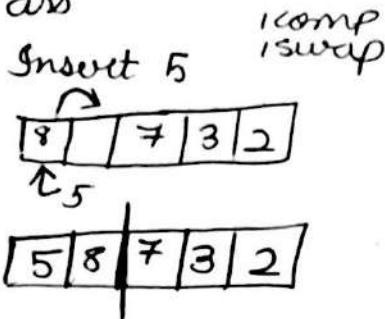
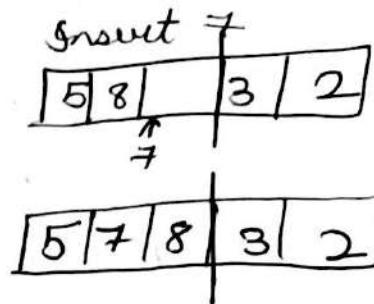
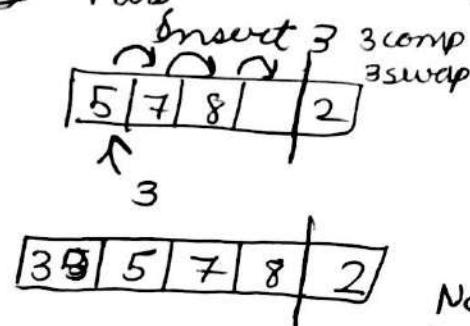
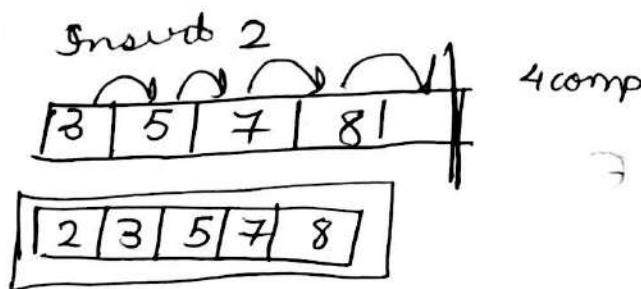
∴ order is maintained

∴ Stable

278

INSERTION SORT

A	8	5	7	3	2
---	---	---	---	---	---

Ist PassIInd PassIIIrd PassIVth PassNo. of Passes $\rightarrow (n-1)$ passesNo. of comp $\rightarrow 1+2+\dots+(n-1) = n(n-1)/2 = O(n^2)$ No. of swap $\rightarrow 1+2+\dots+(n-1) = \frac{n(n-1)}{2} = O(n^2)$

Insertion Sort is better in Linked List than Array : (Insertion sort is made for sorting linked list)

Program

```

void insertionSort(int A[], int n)
{
    for(i=1; i<n; i++)
    {
        j = i-1;
        x = A[i];
        while(j > -1 && A[j] > x)
        {
            A[i+1] = A[i];
            i--;
        }
        A[j+1] = x;
    }
}

```

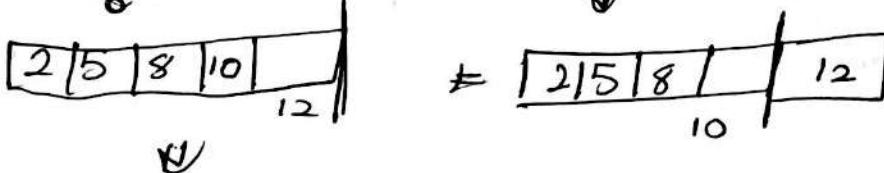
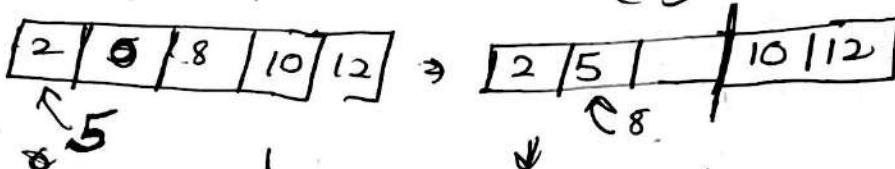
(7)

Adaptive or Not.

12 | 5 | 8 | 10 | 12

No of comp $\rightarrow n-1 = O(n)$

No of swaps $\rightarrow 0 = O(1)$



12 | 5 | 8 | 10 | 12

No of comp $\rightarrow n-1 = O(n)$

No of swap $\rightarrow 0 = O(1)$

Min

Max

Min $O(n)$

$O(n^2)$

swap $O(1)$

$O(n^2)$

Stable

~~Stable~~

3 | 5 | 8 | 10 | 12 | 5



3 | 5 | 8 | 10 | 12 |



3 | 5 | 5 | 8 | 10 | 12

\therefore Stable

\therefore ADAPTIVE AS WELL AS STABLE

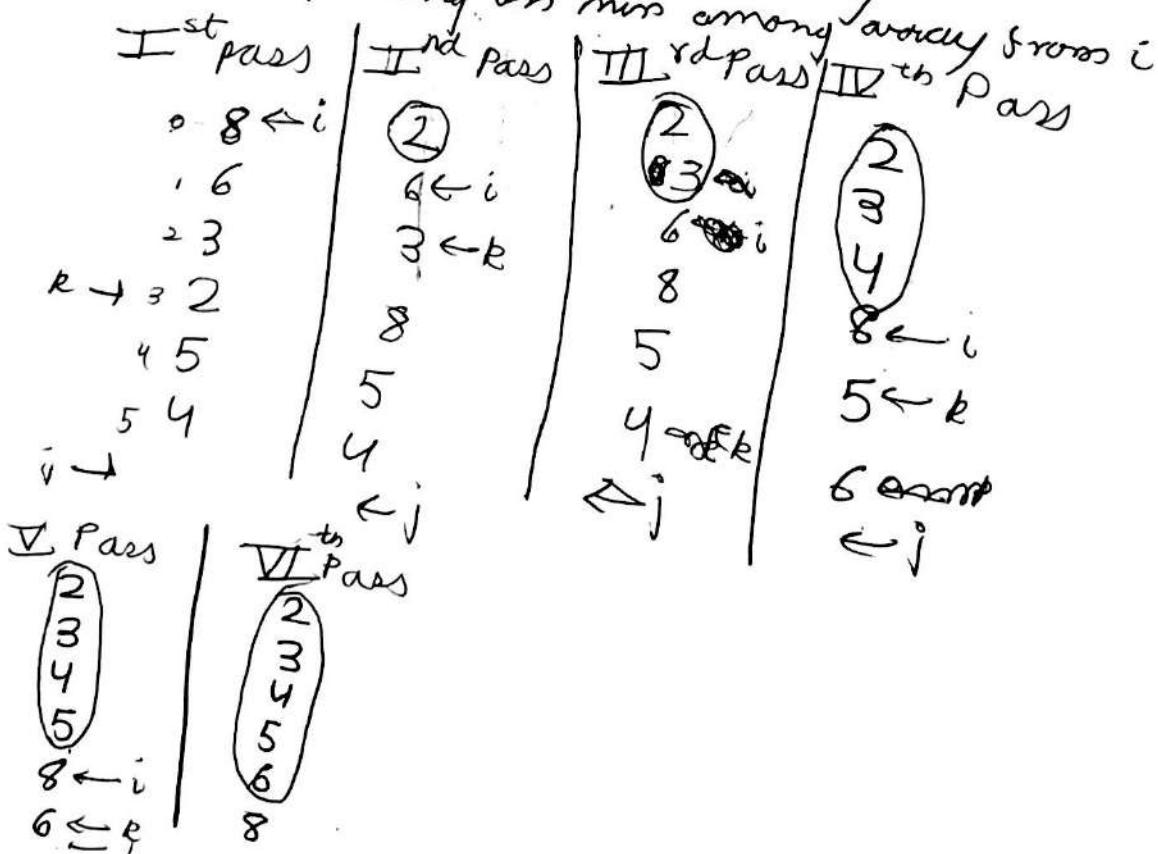
	Bubble Sort	Insertion	
Min comp.	$O(n)$	$O(n)$	already in Ascending
Max comp.	$O(n^2)$	$O(n^2)$	Descending
Min swap	$O(1)$	$O(1)$	Ascending
Max swap	$O(n^2)$	$O(n^2)$	Descending
Adaptive	✓	✓	
Stable	✓	✓	
Linked list	No	Yes	
p. passes significance	Yes	No	

ONLY BUBBLE AND INSERTION SORT ARE ADAPTIVE

SELECTION SORT

We go to particular place and then find the best suitable element at that place

- i : points to the particular place
- j : scanning through the array
- k : pointing on max among array from i



Ist Pass: 5 No of comp: 5
No of swap: 1

IInd Pass: No of comp: 4
No of swap: 1

IIIrd Pass: No of comp: 3
No of swap: 1

IVth Pass: No of comp: 2
No of swap: 1

Vth Pass: No of comp: 1
No of comp

No of comp 28)
 $1 + 2 + 3 + \dots + (n-1)$
 $\frac{n(n-1)}{2} = O(n^2)$

No of swaps = n - 1

If we perform k Passes we will get k smallest Pass

Adaptive
Program:

i	A[i]
0	8
1	6
2	3
3	10
4	9
5	4
6	12
7	5
8	2
9	7

Adaptive or Not?

2 $\leftarrow i$
4
5
10
12
16

When we check whether any swap done on 1st pass we cannot say its sorted or not.
 \therefore NOT ADAPTIVE

void selectionSort (int A[10])

{ int i;

for (i=0, i < n-1, i++)

{ for (j=R=i; j < n; j++)
 if (A[i] > A[j])
 R = j;

 swap(A[i], A[R]);

 stable

0	8 $\leftarrow i$	2
1	3	3
2	5	5
3	8	8
4	4	4
5	2 $\leftarrow R$	7
6	7	7

- Order of initially 8 have been change
- NOT STABLE

282

IMP: POINTS OF SELECTION

→ IT FORMS MIN NO SWAP

→ K PASSES GIVE TO SMALLEST NO.

Quick Sort

50 70 60 90 40 80 10 20 30 ∞
i j

i will be looking for any element greater than pivot

j will be looking for element smaller than or equal to pivot

pivot
50 70 60 90 40 80 10 20 30 ∞
i ← → j

pivot
50 30 60 90 40 80 10 20 70 ∞
i ← → j

pivot
50 30 20 90 40 80 10 60 70 ∞
i ← → j

pivot
50 30 20 10 40 80 90 60 70 ∞

Now $j < i \therefore$ Now swap 50 & 40

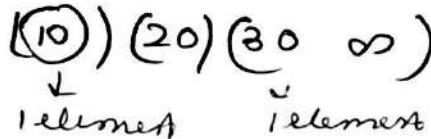
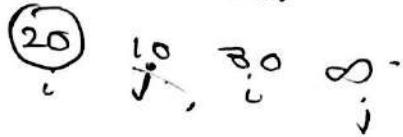
$\therefore (40 \ 30 \ 20 \ 10) \underline{50} (80 \ 90 \ 60 \ 70 \ \infty)$

50 is at its correct position
partitioning
process.

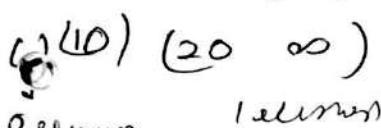
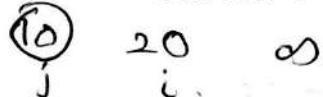
Now Apply quick sort on I & II & do the
for recursively

Quick sort will work for min 2 elements
Quick sort USE PARTITION POSITIONING ALGORITHM

For 3 element

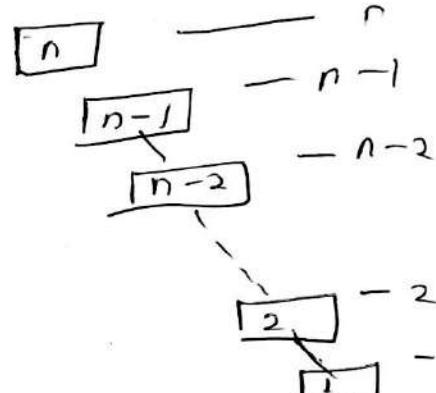
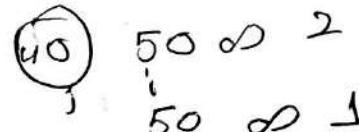
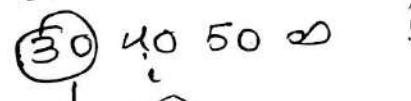
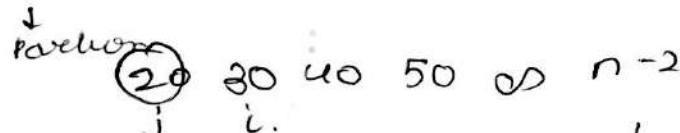
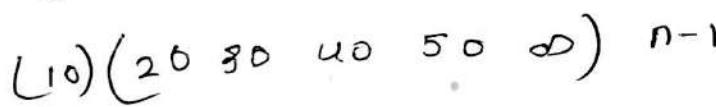
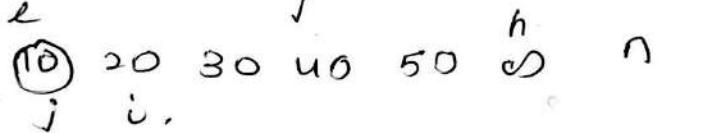


For 2 elements.



No Need to sort both

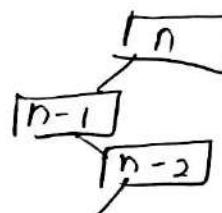
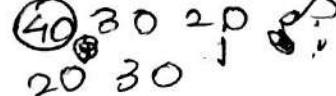
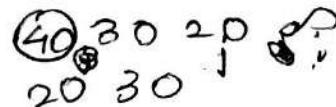
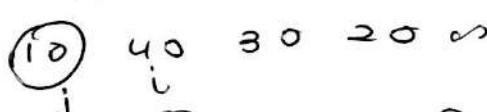
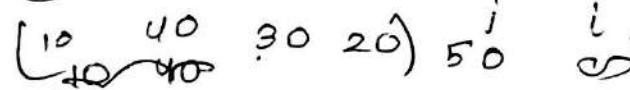
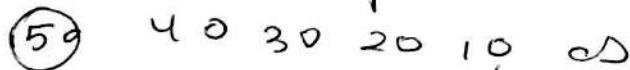
for Ascending order



$$1+2+\dots+n = \frac{n(n+1)}{2}$$

IT IS
Worst case

for Descending order



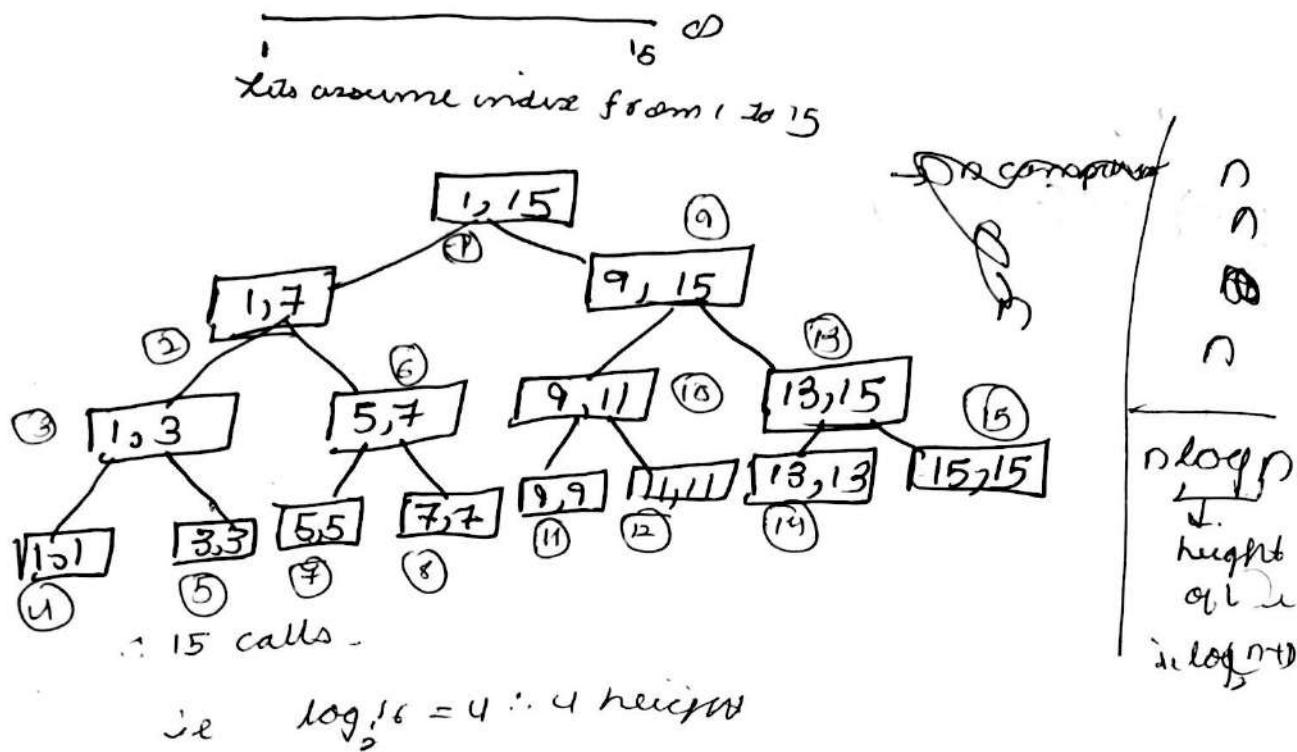
1 2 j

$$1+2+3+4+\dots+n = \frac{n(n+1)}{2}$$

$$O(n^2)$$

Best Case

(284)



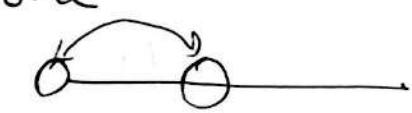
Best Case \rightarrow if partitioning is in middle.

Time : $O(n \log n)$

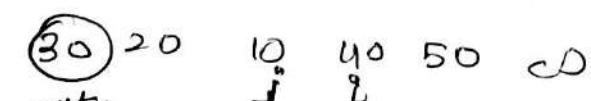
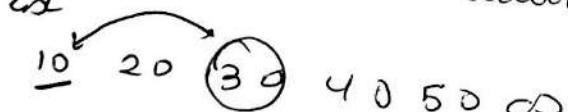
Worst Case \rightarrow if partitioning is on any end.
Time : $O(n^2)$ (already sorted)

NOTE: Avg Case Time $\rightarrow O(n \log n)$

\Leftrightarrow we want sorted list to have Best Case time so we will make pivot to middle



swap middle with first
and then do the same procedure



partition at middle

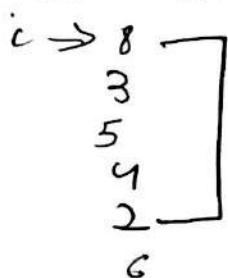
117

Then,

Best Case \rightarrow sorted list $O(n \log n)$

Worst Case \rightarrow partitions at any end $O(n^2)$

Selection Sort



In selection sort we select the positions and find elements for that positions

Quick Sort:

(50) 70 90 60 10 20 30 40

But in quick sort we select element and find out its position
Other Names: Of Quick Sort:
+ Selection Exchange Sort
+ Partition Exchange Sort

Code Quick Sort:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
void swap(int *x, int *y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}
int partition(int A[], int l, int h)
{
    int pivot = A[l];
    int i = l; j = h;
    do
    {
        do {i++;} while (A[i] <= pivot);
        do {j--;} while (A[j] > pivot);
    }
```

(286)

```
if ( $i < j$ ) swap(&A[i], &A[j]),  
    } while ( $i < j$ ),  
    swap(&A[l], &A[j]),  
    return j;  
}  
void Quicksort(int A[], int l, int h)  
{  
    int j;  
    if ( $l < h$ )  
    {  
        j = partition(A, l, h);  
        Quicksort(A, l, j);  
        Quicksort(A, j + 1, h);  
    }  
}  
int main()  
{  
    int A[] = {11, 13, 7, 12, 16, 9, 24, 5, 8, 10, 3, INT32_MAX},  
        n = 10, i, j;  
    Quicksort(A, n);  
    for (i = 0; i < 10; i++)  
        printf("%d", A[i]);  
    printf("\n");  
    return 0;  
}
```

Merging 2 Array

	A	B
i →	0	0
0	2	4
1	10	9
2	18	19
3	20	25
4	23	

I	$i=0$	$j=0$	$A[i] < B[j] \therefore i++$
II	$i=1$	$j=0$	$A[i] > B[j] \therefore j++$
III	$i=1$	$j=1$	$A[i] > B[j] \therefore j++$
IV	$i=1$	$j=2$	$A[i] < B[j] \therefore i++$
V	$i=2$	$j=2$	$A[i] < B[j] \therefore i++$
VI	$i=3$	$j=2$	$A[i] > B[j] \therefore j++$
VII	$i=3$	$j=3$	$A[j] < B[j] \therefore j++$
VIII	$i=4$	$j=3$	$A[i] < B[j] \therefore i++$
X	$i=5$ (out of A)	$j=3$	print all elements of B

C	2
	4
	9
	10
	18
	19
	20
	23
	25

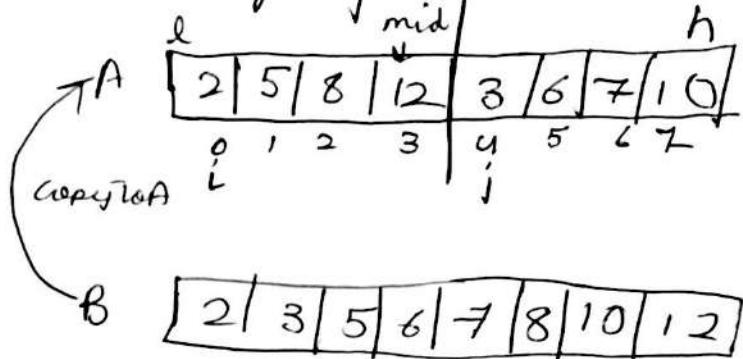
```

void Merge (int A[], int B[], int m, int n)
{
    int i, j, k;
    i = j = k = 0;
    while (i < m && j < n)
    {
        if (A[i] < B[j])
            C[k++] = A[i++];
        else
            C[k++] = B[j++];
    }
    for ( ; i < m, i++)
        C[k++] = A[i];
    for ( ; j < n, j++)
        C[k++] = B[j];
}

```

Time complexity $O(m+n)$

Merging Without Array



```

void Merge(int A[], int l, int m, int h)
{
    int i, j, k;
    int B[h+1];
    int i = l; j = mid + 1; k = l
    while (i <= mid & & j <= h)
    {
        if (A[i] < A[j])
            B[k++] = A[i++];
        else
            B[k++] = A[j++];
        if (j == mid)
            i++;
        for (j; i <= h; j++)
            B[k++] = A[i++];
    }
    if (i <= l; i <= h; j++)
        A[i] = B[i];
}

```

Merging Multiple List

(284)

4-way Merging. (M-way Merging)

A	B	C	D
2	3	5	8
5	6	7	16
15	18	12	20
i	j	k	l

Method I:

$i=0 \ j=0 \ k=0 \ l=0 \ A[i]$ is min.

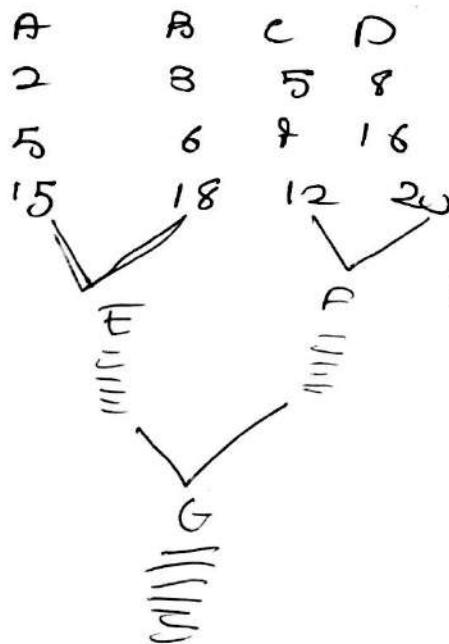
$\therefore i=1 \ j=0 \ k=0 \ l=0 \ B[j]$ is min

$i=1 \ j=1 \ k=0 \ l=0 \ C[k]$ is min.

$i=1 \ j=1 \ k=1 \ l=0 \ D[l]$ is min

so on.

Method II



(2-way Merging)

Iterative Merge Sort

problem statement : Sort an array list of 8 elements by Merge Sort.

\because each element is a list.

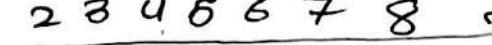
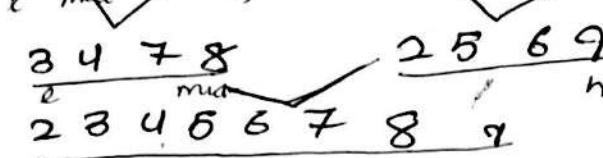
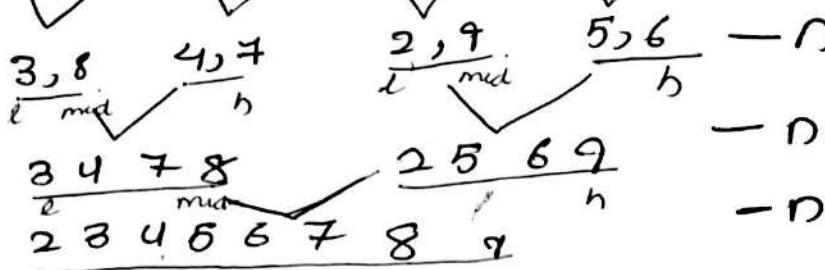
\therefore Sort an array containing 8 sorted lists where each list contains 1 element.

290

A

$i = 0$	$i = 1$	$i = 2$	$i = 3$	$i = 4$	$i = 5$	$i = 6$	$i = 7$
8	3	7	4	9	2	6	5

Ist Pass $p=2$
IInd Pass $p=4$
IIIrd Pass $p=8$



$\log n$
time
size of
tree.

Ques. ans

I Pass means traversing through all elements
∴ Greater Versions has Passes & not in Reversal

$p=2$ means merging of 2 list to result in size of 2

$p=4$ means merging of list having 2 elements to result into list having 4 elements

Code:

void I_MergeSort(int A[], int n)

{ int p, i, l, mid, h;

= for ($p=2$; $p \leq n$; $p=p*2$)

{ for ($i=0$; $i+p-1 < n$; $i=i+p$)

{ $l=i$;

$h=i+p-1$;

mid $\lceil (l+h)/2 \rceil$

Merge(A, l, mid, h);

} if ($p/2 < n$) // for cases having n
not being power

. // Merge(A, 0, p/2, n-1); $\frac{p}{2}$ gives
many cases

$l=i-p$

mid = $i+1$; $h=n-1$;

if ($i+1 < n$)

Merge(A, l, mid, h);

impl

for case, n in
Not having power
of 2

297
291

Ex Case for if ($p/2 < n$)

$n = 10$

I pass
 $p=2$

8	3	7	4	9	2	6	1	5	1	3

II pass
 $p=4$

3	8	4	7	2	9	5	6	1	3	

III pass
 $p=8$

3	9	7	8	2	5	6	9	1	3

(if $\frac{16}{2} = 8 < 10$)

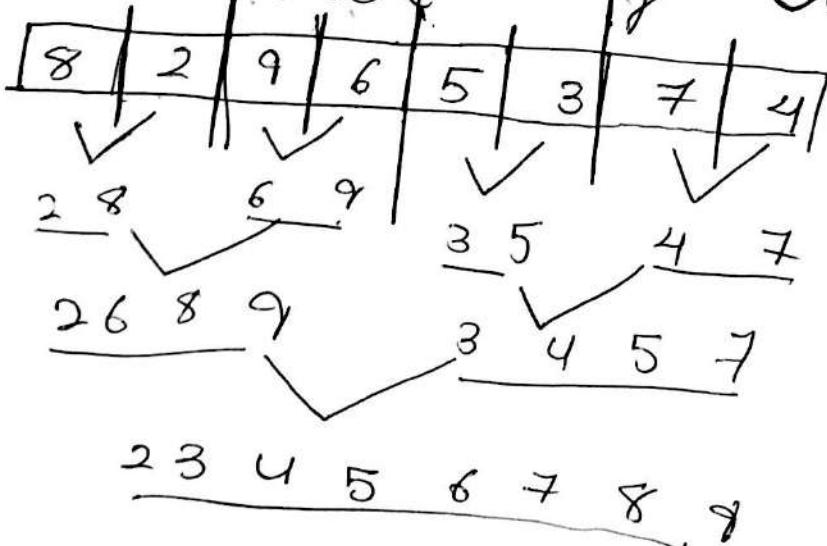
$\therefore \text{mid} = 8$

Sorted \Rightarrow 1 2 3 3 4 5 6 7 8 9

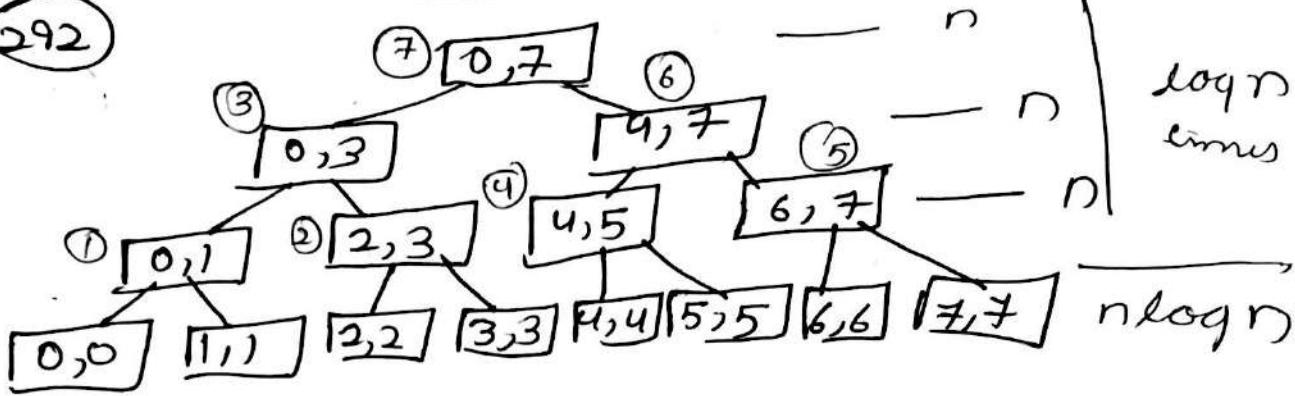
\therefore Time Complexity : $n \log n$

This is called 2-Way Merge Sort.

Recursive Merge Sort



5 | 6
9



① order of merging
Space A - n
 B - n

$$\frac{\text{stack} - \log n}{2n + \log n}$$

∴ In Comparison based Sort only Merge sort take extra space

void Mergesort (int A[], int l, int h)
{ if ($l < h$)

mid = $(l+h)/2$;
Mergesort(A, l, mid);
Mergesort(A, mid+1, h);
Merge(A, l, mid, h);

}

Count Sort

(Index based sorting)

fastest but consumes lots of memory

A	6	3	9	10	15	6	8	12	3	6
---	---	---	---	----	----	---	---	----	---	---

count	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	

size of arr should be ~~max~~ equal to max. elements of A

count	0	0	0	2	0	3	0	1	1	1	0	1	0	0	1	
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

& then copy to A & from back from 0 to 15

3	3	6	6	6	8	9	10	10	12
---	---	---	---	---	---	---	----	----	----

Time complexity : $n+m = O(n)$

THOUGHT: when we have elements not large thus only we use CountSort so that count array doesn't take large space.

void CountSort (int A[], int n)

{ int max, i;

int *c;

max = findMax(A, n)

c = new int [max+1];

for (i=0; i<max+1; i++)

c[i] = 0;

for (i=0; i<n; i++)

{ c[A[i]]++; }

i=0; j=0;

6
8

294

6

```

while (i < maxn)
{
    if (c[i] / 20)
        { A[i + 1] = i
          c[i] -= 20;
        }
    else
        i++;
}

```

四

$\sigma(n)$

Bin / Bucket Sort

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

Says through A

at 0, 6 go to index 6 of Bins and drop into the 6 bins & similarly for all.

Bins

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

↓

3	1
---	---

↓

3	1
---	---

↓

6	1
---	---

↓

6	1
---	---

↓

8	1
---	---

↓

9	1
---	---

↓

10	1
----	---

↓

12	1
----	---

↓

15	1
----	---

↓

1100	
------	--

Now for D 2015 ~~for~~ of Bin copy to R till Bin becomes NULL again.

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

Time Complexity
 $O(n)$

```

void BinSort (int A[], int n)
{
    int max, i, j;
    Node **Bins;
    max = findMax(A, n);
    Bins = new Node * [max + 1];
    for (i = 0; i < max + 1; i++)
        Bins[i] = NULL;
    for (i = 0; i < n; i++)
    {
        insert(Bins[A[i]], A[i]);
    }
    i = 0, j = 0;
    while (i < max + 1)
    {
        while (Bins[i] != NULL)
        {
            A[j++] = delete(Bins[i]);
            i++;
        }
    }
}

```

$\frac{n}{3n} \cdot O(n)$

* Radix Sort

(similar to Bin Sort)

10 size array is required \because No is in decimal
 \therefore digit can be 0 - 9.

A	237	146	258	348	152	163	235	48	36	6
	0	1	2	3	4	5	6	7	8	9

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

596

Ist Pass

$$(A[i]/11) \% 10 = \text{last digit}$$

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

152 62 163 235 146 36 237 348 48 259

↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓

62 36 163 235 146 237 348 48 259

Now from 0 to 9 in Bins coping back to A and deleting from Bins

A 152 62 163 235 146 36 237 348 48 259

IInd Pass

$$(A[i]/110) \% 10 = \text{second last digit}$$

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

235 146 152 62
↓ ↓ ↓ ↓
36 348 259 163
↓ ↓
237 48

A	0	1	2	3	4	5	6	7	8	9
	235	036	237	146	348	048	152	259	062	163

IIIrd Pass

$$(A[i]/1100) \% 10 = \text{third last digit}$$

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

36 146 235 348
↓ ↓ ↓ ↓
48 152 237
↓ ↓
62 163 259

A	0	1	2	3	4	5	6	7	8	9
	36	48	62	146	152	163	235	237	259	348

∴ Array is Sorted.

Shell Sort

useful for very large size list.
(extension of insertion sort)

Video: 043

concept

- If array is not sorted due to few elements then it require very few shifting.

∴ We apply sorting in Gaps

$$\text{I}^{\text{st}} \text{ Pass gap} = \left\lceil \frac{n}{2} \right\rceil$$

then

$$\text{II}^{\text{nd}} \text{ Pass gap} = \left\lceil \frac{n}{4} \right\rceil$$

∴ till gap = 1 work at gap 1 it will be same as insertion sort.

```
ex: #include < stdio.h >
    #include < stdlib.h >
```

```
void swap( int *x, int *y )
```

```
{ int temp = *x;
  *x = *y;
  *y = temp }
```

```
void shellsort( int A[], int n )
```

```
{ int gap, i, j, temp; }
```

```
for( gap = n/2; gap >= 1; gap /= 2 )
```

```
{ for( i = gap; i < n; i++ )
```

```
{ temp = A[i];
```

```
j = i - gap
```

```
while( j >= 0 && A[j] > temp )
```

```
{ A[j + gap] = A[j];
  j = j - gap; }
```

```
,
```

296

$$A[j + \text{gap}] = \text{temp} ;$$

Analysis:

Time Complexity $n \times \log n$

[here we have not consider
shifting]

scans
int

no. of gaps between gaps
no. of passes

Also analysed as

$$O(n^{3/2}) = O(n^{1.5})$$

$$O(n^{7/3}) = O(n^{1.66})$$

sometimes gaps are also taken as prime no's
less than n

15 marks, max ...

xx

Hashing Technique

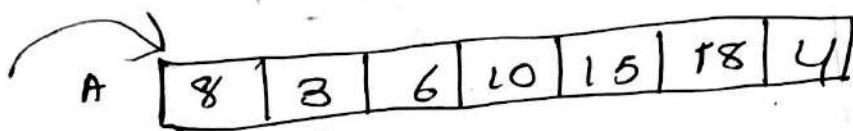
1) Why Hashing

Hashing is used for searching

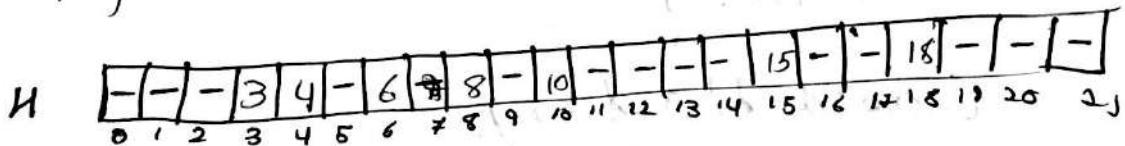
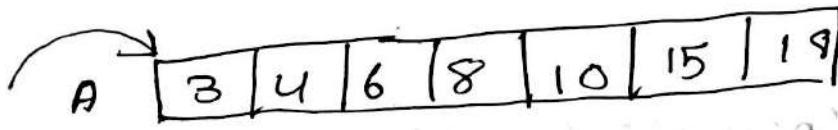
Searching

key : 8, 3, 6, 10, 15, 18, 4

Linear search
 $O(n)$



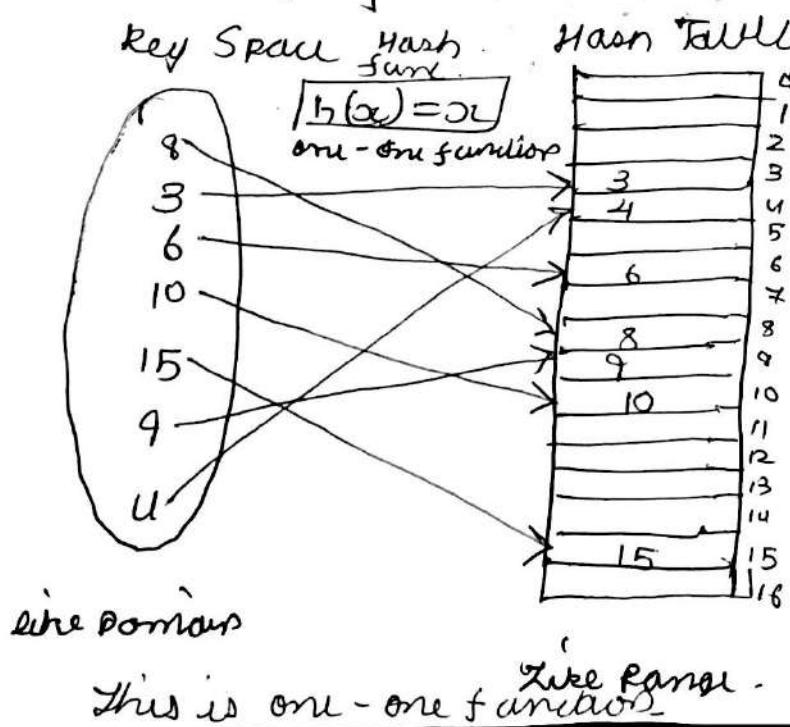
Binary search
 $O(\log n)$



\therefore Search: $O(1)$

search 3 Present at index 3

But it consumes lot of memory :- we do mapping



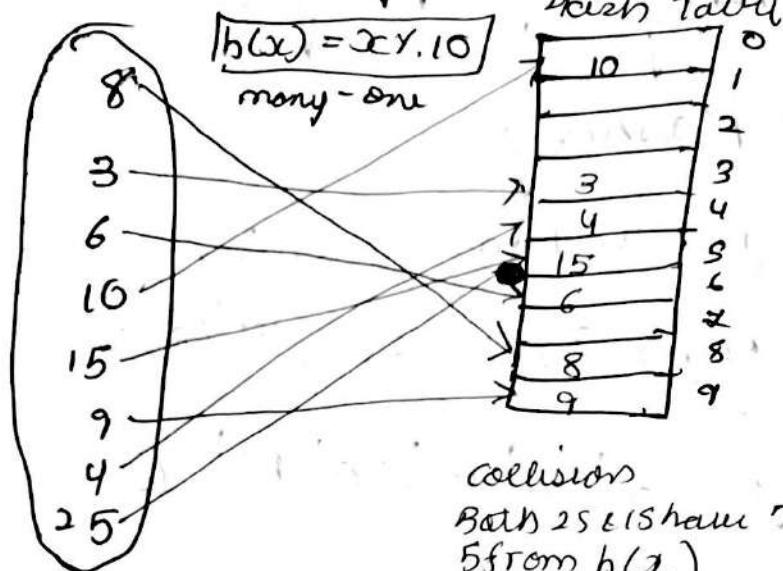
Mapping functions

- one-one
- one-many
- many-one
- many-many

$h(x) = x$ is ideal hash function. It takes constant time -

300

Now to reduce space (By Modulus Hash Function)
we modify Hash Function



Now How to deal Collisions (Methods to resolve)

Open Hashing (we would use extra space)

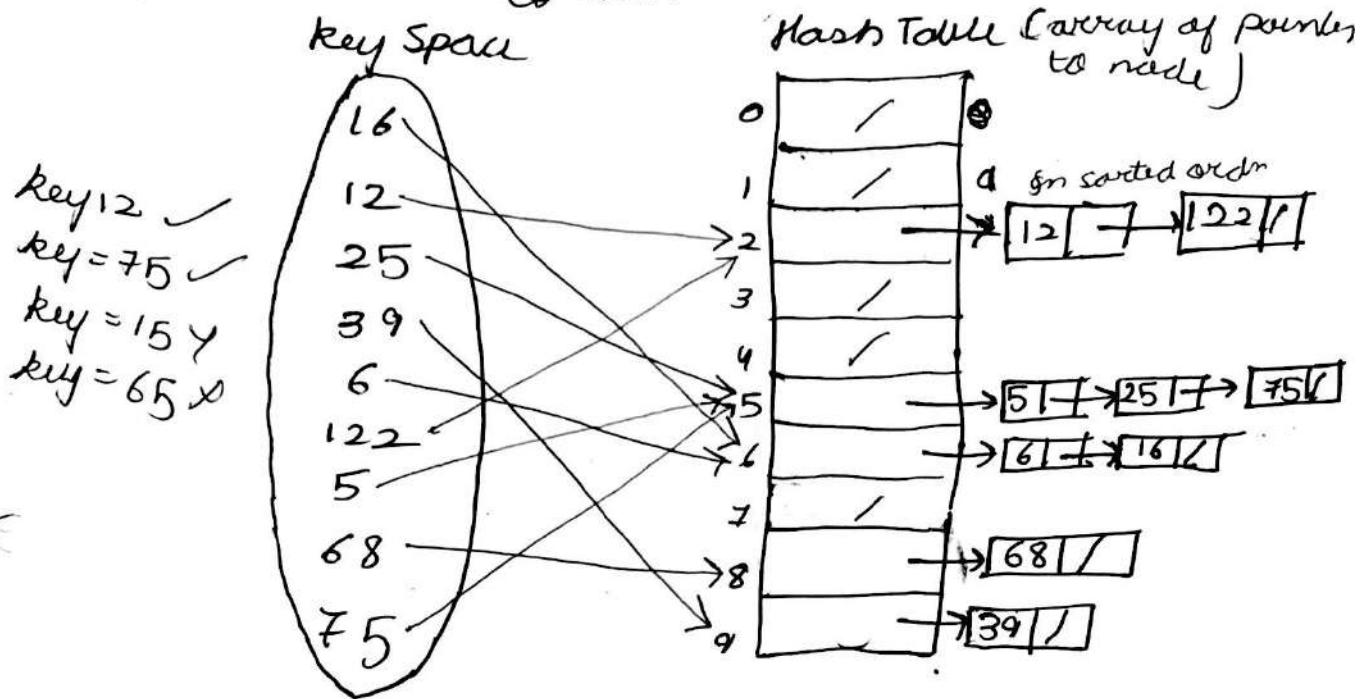
- chaining (more than Hash Table)

Closed Hashing, (no use of extra space)

- Open Addressing
 - 1) Linear Probing
 - 2) Quadratic Probing
 - 3) Double Hashing
- ways to address where to store in hash table
- (we will use space of Hash Table which are left)

(50)

$$h(x) = 2x + 10$$



Analysis

$$n = 100$$

$$\text{size} = 10$$

$$\text{Loading factor} \rightarrow \lambda = \frac{n}{\text{size}}$$

$\therefore \lambda = \frac{100}{10} = 10$ ~~we assume at every location~~ we assume at every location on we have 10 elements.

(Analysis of Hashing is allays based on Loading factor
 $\therefore \lambda = \frac{n}{\text{size}}$)

Avg. successful search

$$t = 1 + \frac{\lambda}{2}$$

Avg unsuccessful search

$$t = 1 + \lambda$$

(302)

In this $h(x) = \frac{G(x)}{10} \% 10$ can be also taken as hash function, i.e. second last digit of x .

Ex: 5, 35, 45, 145, 175, 265, 845, 55, 25

Here last digit will give always 5.
If we will No Hashing by using $\lceil \frac{x}{10} \rceil$ if we use $\lceil \frac{x}{10} \rceil \% 10$ then Hashing Technique fails.

Lets Code Chaining

```
#include < stdlib.h >
```

```
struct Node
```

```
{ int data ;
```

```
 struct Node * next ;
```

```
 } ;
```

```
void SortedInsert( struct Node * &H, int x )
```

```
 { struct Node * t, * q = NULL, * p = H ;
```

```
 t = ( struct Node * ) malloc ( sizeof( struct  
 Node ) );
```

```
 t -> data = x ;
```

```
 t -> next = NULL ;
```

```
 if ( H == NULL )
```

```
 * H = t ;
```

```
 else
```

```
 { while ( p && p -> data < x )
```

```
 { q = p ;
```

```
 p = p -> next ;
```

```
 if ( p == NULL )
```

```
 { t -> next = * H ;
```

```
 * H = t ;
```

```
 else
```

```
 { t -> next = q -> next ;
```

```
 q -> next = t ;
```

```

struct Node * search(struct Node *p, int key)
{
    while (p != NULL)
    {
        if (key == p->data)
            return p;
        p = p->next;
    }
    return NULL;
}

int hash(int key)
{
    return key % 10;
}

void Insert(struct Node **HT[], int key)
{
    int index = hash(key);
    sortedInsert(&HT[index], key);
}

int main()
{
    struct Node * HT[10];
    struct Node * temp;
    int i;
    for (i = 0; i < 10; i++)
        HT[i] = NULL;
    Insert(HT, 12);
    Insert(HT, 22);
    Insert(HT, 42);
    temp = search(HT[hash(21)], 21);
    printf("%d", temp->data);
    return 0;
}

```

3089

Linear Probing

$$h(x) = x \cdot 10$$

26	80	0
30	29	1
45		2
23	23	3
25	43	4
43	45	5
74	26	6
19	25	7
29	74	8
	19	9

Searching

$$\Rightarrow \text{key} = 45$$

$$\begin{aligned} h(x) &= 45 \cdot 1.10 \\ &= 5 \end{aligned}$$

go to 5 found

$$\Rightarrow \text{key} = 74$$

$$\begin{aligned} h(x) &= 74 \cdot 1.10 \\ &= 4 \end{aligned}$$

'among $H[4]$ is not found

$H[5], H[6], H[7], H[8]$ is found

\therefore we will keep on searching till the key is found or we get a Blank Space

$$h'(x) = (h(x) + f(i)) \cdot 1.10 \text{ where } f(i) = i \quad i = 0, 1, 2, \dots$$

$$\begin{aligned} h'(25) &= (h(25) + f(0)) \cdot 1.10 \\ &= (5 + 0) \cdot 1.10 = 5 \\ \text{index } 5 &\text{ is empty} \end{aligned}$$

$$\begin{aligned} h'(25) &= (h(25) + f(1)) \cdot 1.10 \\ &= (5 + 1) \cdot 1.10 = 6 \end{aligned}$$

$$\begin{aligned} h'(25) &= (h(25) + f(2)) \cdot 1.10 \\ &= (5 + 2) \cdot 1.10 = 7 \end{aligned}$$

$$\begin{aligned} h'(25) &= (h(25) + f(0)) \cdot 1.10 \\ &= (5 + 0) \cdot 1.10 = 5 \\ (5+1) \cdot 1.10 &= 6 \\ (5+2) \cdot 1.10 &= 7 \text{ empty} \end{aligned}$$

$$\Rightarrow \text{key} : 40$$

$$h(x) = 0$$

$H[0]$ = not found

$H[1]$ = not found

$H[2]$ = Blank

\therefore STOP after getting null.

1.

10

Analysis:

Loading factor $d = \frac{\rho}{\text{size}}$

$$+ \quad \lambda = \frac{q}{10} = 0.9$$

$$\boxed{d \leq 0.5}$$

as for size = 10 we should not insert $\underline{5}$ more than $\underline{5}$

Avg successful search

$$t = \frac{1}{d} \ln\left(\frac{1}{1-d}\right)$$

Avg unsuccessful search

$$b = \frac{1}{1-d}$$

Drawback

$\because d = 0.5 \therefore$ half space would be empty & wasted.

\Rightarrow it forms primary clusters.

For Deletion we have to do Rehashing time consuming

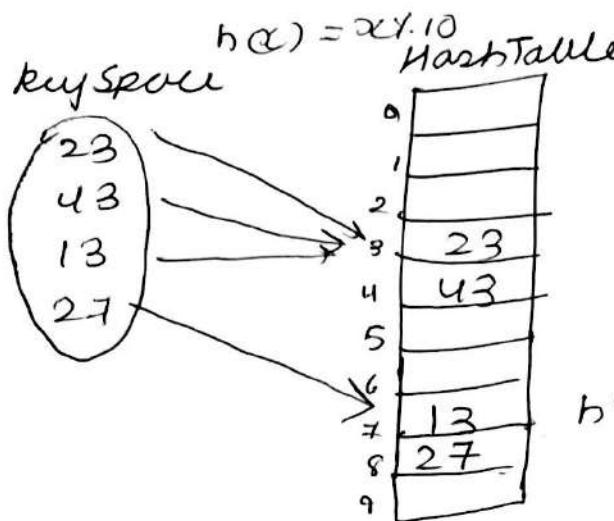
- * Make hash table NULL then insert again
- In Linear Probing we do not suggest deleting. If at α we want to delete
 - * we make a flag array

roll	80	1
delete	29	1
25	23	1
	43	1
	45	1
	26	1
	25	0
	74	1
	19	1
		free

this denotes it's not present

306

Quadratic Probing



$$h'(x) = (h(x) + f(i)) \times 10$$

where $f(i) = i^2$
 $i = 0, 1, 2, \dots$

$$\begin{aligned} h'(13) &= (h(13) + f_0) \times 10 \\ &= (3+0) \times 10 = 3 \\ &= (3+1) \times 10 = 4 \\ &= (3+4) \times 10 = 7 \end{aligned}$$

$$\begin{aligned} h'(27) &= 7+0 = 7 \text{ filled} \\ &7+1 = 8 \text{ empty} \end{aligned}$$

→ Avg successful search
 $- \log_e(1-d)$

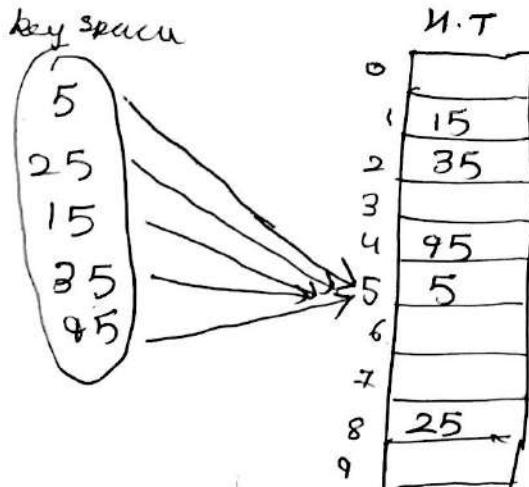
→ Avg unsuccessful search

$$= \frac{1}{1-d}$$

Double Hashing

(to avoid multiple collisions)

$$h_1(x) = x \times 10$$



$$\begin{aligned} h_1(x) &= x \times 10 \\ h_2(x) &= R - (x \% R) \quad (x \% 7) \\ h'(x) &= (h_1(x) + i * h_2(x)) \times 10 \end{aligned}$$

where $i = 0, 1, 2, \dots$

general form of
 $h_2(x) = R - (x \% R)$
 where R is primary no. i.e.
 largest prime no. which is
 less than size of H.T
 \therefore here it is 7

$h'(2)$ has a property that it should not give index empty. 307

$$\Rightarrow h'(2) = (5 + 2 * 2) \% 10 = \underline{\underline{8}} \text{ empty}$$

$$7 - (25 \% 7)$$

$$7 - 4 = \underline{\underline{3}}$$

$$\Rightarrow h'(5) = (5 + 1 * 5) \% 10 = \underline{\underline{1}} \text{ empty}$$

$$7 - (15 \% 7)$$

$$7 - 1 = \underline{\underline{6}}$$

$$\Rightarrow h'(35) = (5 + 1 * 7) \% 10 = \underline{\underline{2}} \text{ empty}$$

$$7 - (35 \% 7)$$

$$7 - 0 = \underline{\underline{7}}$$

$$\Rightarrow h'(75) = (5 + 1 * 3) \% 10 = \underline{\underline{8}} \text{ filled by } 25$$

$$7 - (95 \% 7)$$

$$\therefore i = \underline{\underline{2}}$$

$$(5 + 2 * 3) \% 10 = \underline{\underline{1}} \text{ filled by } 15$$

$$\therefore i = \underline{\underline{3}}$$

$$(5 + 3 * 3) \% 10 = \underline{\underline{4}} \text{ empty.}$$

Let's Code Linear Probing

```
#include <stdio.h>
```

```
#define SIZE 10
```

```
int hash (int key)
```

```
{ return key \% SIZE;
```

```
int probe (int H[], int key)
```

```
{ int index = hash (key);
```

```
int i = 0;
```

```
while (H[(index + i) \% SIZE] != 0)
```

```
i++;
```

```
return (index + i) \% SIZE;
```

```
}
```

```

void Insert(int H[], int key)
{
    int index = hash(key);
    if (H[index] != 0)
        index = probe(H, key);
    H[index] = key;
}

int Search(int H[], int key)
{
    int index = hash(key);
    int i = 0; For Quadratic  
for i ≠ i.
    while (H[(index + i) % SIZE] != key)
        i++;
    return (index + i) % SIZE;
}

int main()
{
    int HT[10] = {0};
    Insert(HT, 12);
    Insert(HT, 25);
    " (HT, 35);
    " (HT, 26);
    printf("Key found at %d\n", Search(HT, 35));
    return 0;
}

```

Types of Hash Functions

309

- 1) Modular (done)
 - 2) Midsquare
 - 3) Folding

IMP: Values in Hash Table should be uniformly distributed

- i) Modulus:

$b(x) = (x^y, \text{size})$

- ① size should be double the no of elements

- 2) MidSquare

$$\Rightarrow \text{key} = 11 \\ = (11)^2 = 121$$

key : - -
- - -

$$\Rightarrow \text{key } 13 \\ = (13)^2 = 169$$

key --
-- $\frac{6}{1}$ --

$$\text{then we take Mod} \\ \text{Ex } 63 \times 10 = \underline{\underline{3}}$$

- ### 3) Folding

$$\begin{array}{r} \text{key} = \boxed{12} \boxed{33} \boxed{47} \\ \hline 12 \\ + 33 \\ + 47 \\ \hline 92 \end{array}$$

use

we can't walk around & 2 as

183 12/24/18 9+2<11

for sort, so we have key = "ABC"

$$\begin{array}{c} A \quad B \quad C \\ \boxed{65 \quad 68 \quad 67} \end{array} \quad : \quad \begin{array}{c} 65 \\ + 66 \\ \hline 131 \end{array}$$

or we can take
48 ~~skip~~

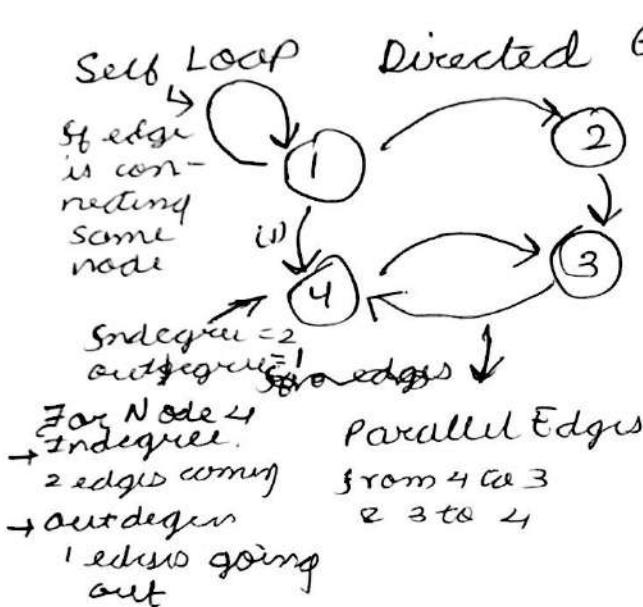
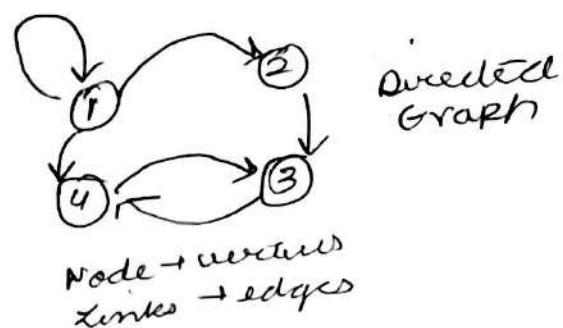
← 198

now we can
do modules
if we want small
size

Graphs
 (connections or vertices)
 ↘ edges

$$G = (V, E)$$

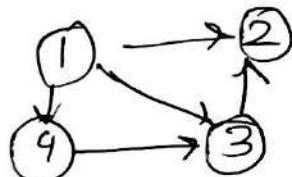
vertices ↓ edges



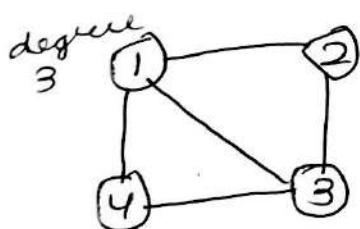
if edge has direction even it is called directed Graph.

edge(1) is outgoing from 1 & incoming to 4

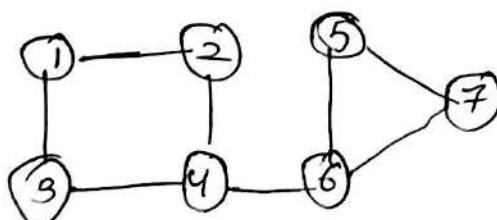
Two vertices connected by an edge are called Adjacent Vertices



Simple Digraph
 (without loops edges & self loop)



Graph / Non-Directed graph
 (No directed edges)
 — is counted as \longleftrightarrow



2 component.

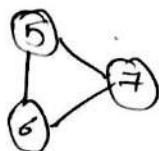
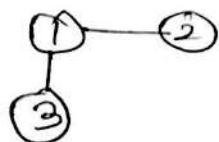
Non-connected

if we connect graph at 4 & 6 & if we remove 6 & 8 edges connected to it then graph would be fragmented in more than one component.

Ex: Remove 6

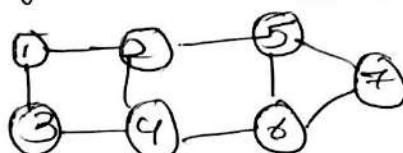


Remove 4



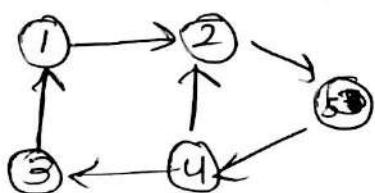
∴ Such Vertices are called Articulation Point
∴ 4 & 6 are articulation point.

If we connect 2 & 5



Biconnected
Graph ~~as~~ i.e. it is
STRONGLY CONNECTED

No articulation point.



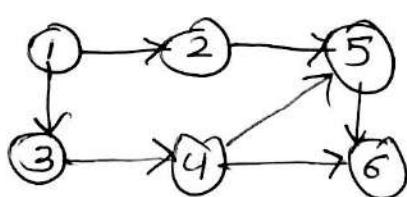
Strongly Connected.

In this from any vertices
you can reach to any
vertices.

Path: Set of all the vertices in between
Pair of vertices

i.e. Path from 1 to 5 is: 1 → 2 → 5

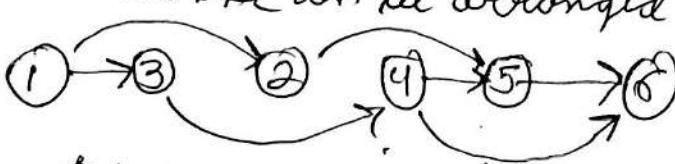
They are CYCLIC



Directed Acyclic
Graph (DAG)

Directed graph with no
cycles. We cannot reach
the same vertex.

These ~~source~~ DAG can be arranged linearly



edges are going in forward direction
only

This is only possible in DAG

Topological
Ordering
of
Vertices

312

Representations of Undirected Graph

1. Represent edges & vertices via matrix.

1) Adjacency Matrix

2) Adjacency List

3) Compact List

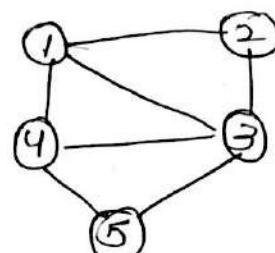
Adjacency Matrix

	1	2	3	4	5	
1	0	1	1	1	0	⑦
2	1	0	1	0	0	to access all sum
3	1	1	0	1	1	$n \times n = n^2$
4	1	0	1	0	1	$O(n^2)$
5	0	0	1	1	0	

5×5

use code indexes from
1

$A[1][2]$ means
edges 1 & 2 are
connected by edges
1 & 5 are not connected.



$$G = (V, E)$$

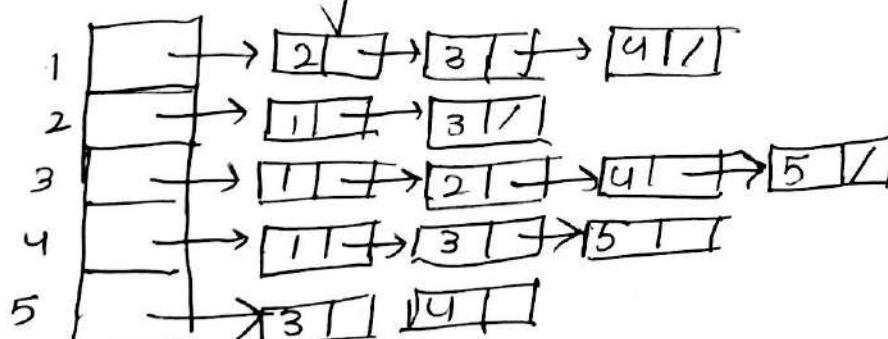
$$|V| = n = 5$$

$$|E| = e = 7$$

\Rightarrow Vertices connected by edge
(i, j)

$$\text{then } A[i][j] = 1$$

Adjacency Matrix List [Array of Link List]



Time to access every element :
 $|V| + 2|E|$

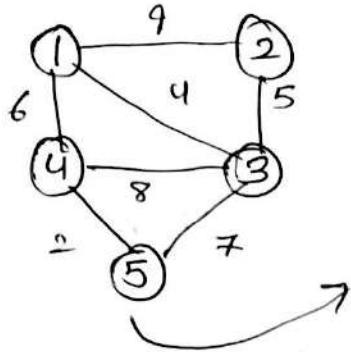
$$O(|V| + 2|E|)$$

5 \times no. of edges
"one edge is counted 2 times."

Space complexity
 $V + E$

If there is weightage to every edges then

(313)

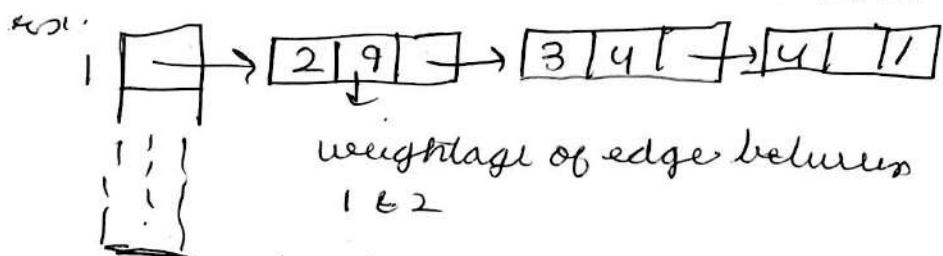
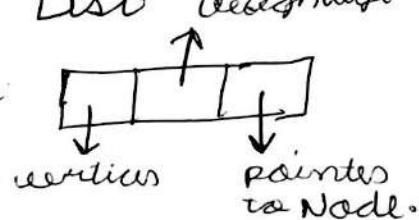


represented as

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 0 & 9 & 4 & 6 & 0 \\ 9 & 0 & 5 & 0 & 0 \\ 4 & 5 & 0 & 8 & 7 \\ 6 & 0 & 8 & 0 & 2 \\ 0 & 0 & 7 & 2 & 0 \end{bmatrix}$$

∴ These matrix is called COST ADJACENCY MATRIX

For cost Adjacency List weightage
we should have Node



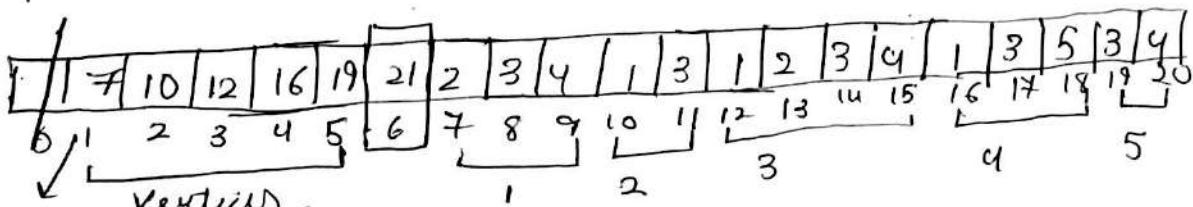
Compact List (single dimensional array)

size : $|V| + 2|E| + 1$ → extra space

For our ex:

$$5 + 2 \times 7 + 1 = 20 \text{ & since we are starting from index } 1 \therefore 20 + 1 = 21$$

from index 1. ∴ 20 + 1 = 21



denotes vertices connected
to ① are stored in array
from index 7 to 9

edges
vertices
connected
with 1.

∴ Space complexity

$$n + 2e \alpha$$

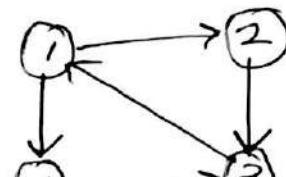
$$\approx n = e$$

$$\therefore n + 2n = \underline{3n} \therefore O(n)$$

Representations of Directed Graph

Adjacency Matrix

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}_{4 \times 4}$$



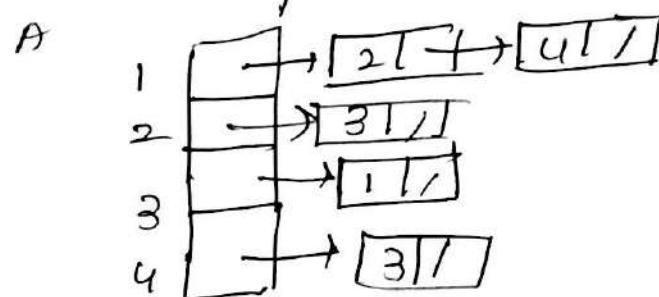
$$G = (V, E)$$

$$\begin{cases} |V| = 4 \\ |E| = 5 \end{cases}$$

$$\text{Here } A[1][3] = 0$$

∴ Node 1 does not direct 3

Adjacency List

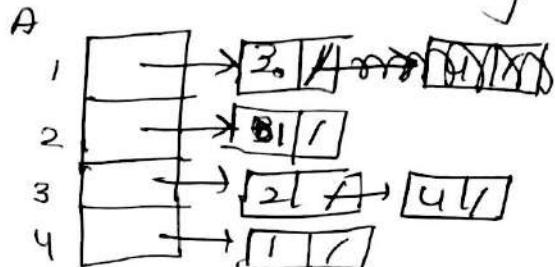


If we want to know ∴ what are edges going out from 1 then we see row 1. If we want to what are edges coming to 1 we see column 1 in Adjacency Matrix

~~In~~ Adjacency list

If we want to know ~~as~~ incoming to ①
 Then we have to Traverse whole list.
 ∴ To ~~know~~ know edges that are coming
 in we make an

Inverse Adjacency list



This gives incoming edges.

∴ Adjacency Matrix

$n \times n = n^2$ if we are accessing every element.

~~In~~ Adjacency list

$$(V+E) = n + e \Rightarrow O(n)$$

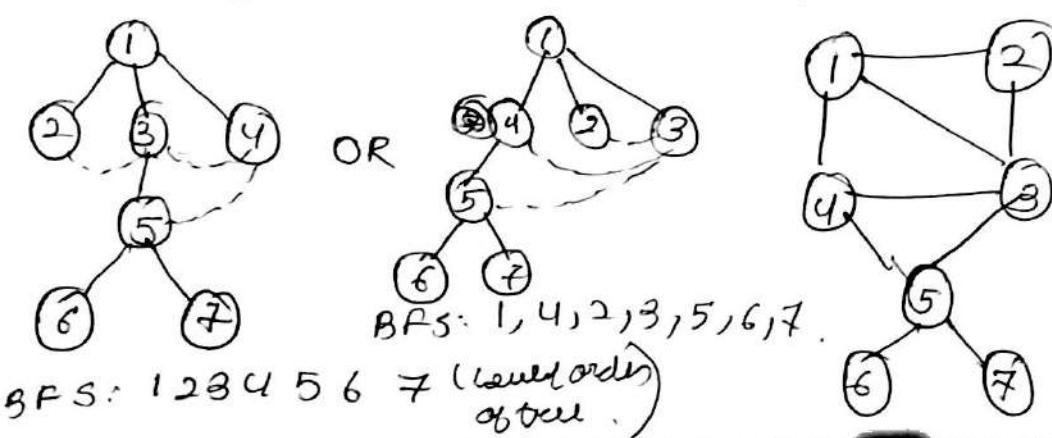
Search Traversal

(I) Breadth First Search

(II) Depth First Search

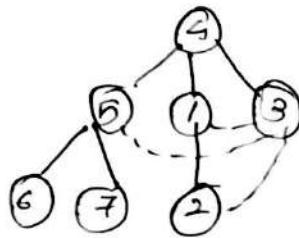
Breadth First Search

[similar as level order]



316

or we can start from vertex 4



BFS: 4, 5, 1, 3, 7, 6, 2

\therefore whenever we select vertex explore all adjacent vertices. You can visit adj. vertex in any order.

BFS

1. visiting (visiting on vertices)

2. exploring (exploring adjacent vertex)

(a) BFS : 1 ^(III) called starting point

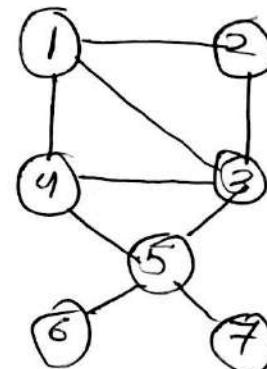
Queue : 1 ^(II) drop in queue

(I) visited 1

IV Now take out from Queue & explore

BFS : 1, 2, 3, 4

Queue : 1, 2, 3, 4



(b) Now visit 2

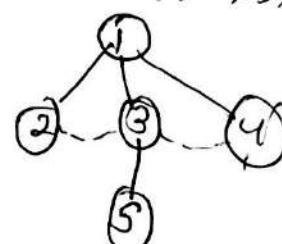
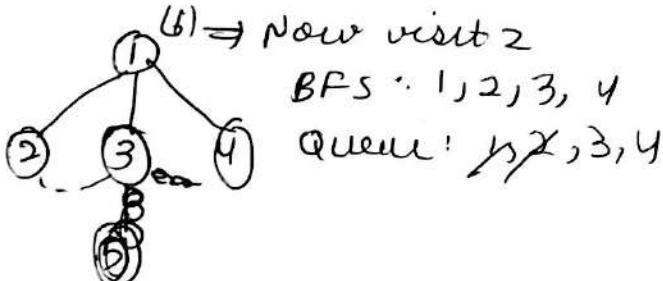
BFS : 1, 2, 3, 4

Queue : 1, 2, 3, 4

(c) Now visit 3

BFS : 1, 2, 3, 4, 5

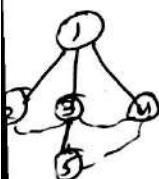
Queue : 1, 2, 3, 4, 5



(d) Now visit 4

BFS : 1, 2, 3, 4, 5

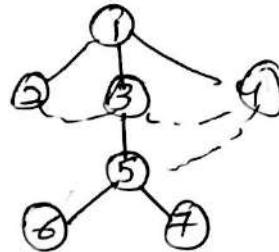
Queue : 1, 2, 3, 4, 5



12

(e) Now visit 5

BFS : 1, 2, 3, 4, 5, 6, 7
Queue : 1, 2, 3, 4, 5, 6, 7



317

Now explore 6

Queue : 1, 2, 3, 4, 5, 6, 7
It is fully explored.

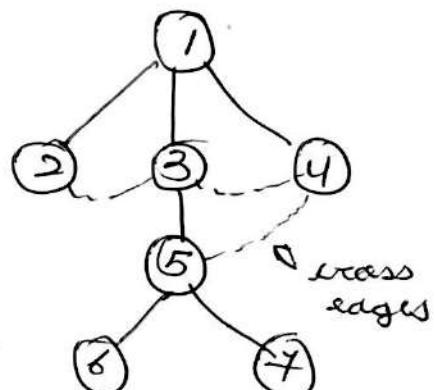
Now explore 7

Queue : 1, 2, 3, 4, 5, 6, 7
Fully explored

Now Queue Empty

∴ BFS Ends

Output 1, 2, 3, 4, 5, 6, 7



BFS spanning Tree

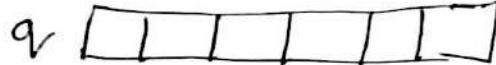
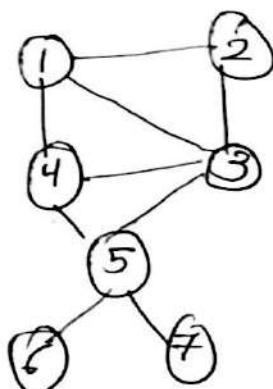
⇒ Cross edges would be connecting with vertices of some level or adjacent level only

Analytically time complexity : $O(n)$
(Deletion & insertion in queue)
(is not taken)

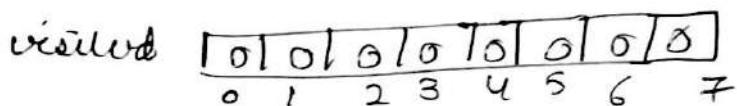
since we are visiting each node

Program for BFS

$u \rightarrow$

$$A = \begin{matrix} & & u \\ & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 2 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 3 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 4 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 5 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 6 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 7 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{matrix}$$


(^{activated}
Bots are assumed
globally)



```

void BFS(int i)
{
    int u;
    printf("v%d.d", i);
    visited[i] = 1;
    enqueue(q, i);
    while (!isEmpty(q))
    {
        u = dequeue(q);
        for (v = 1; v <= n; v++)
        {
            if (A[u][v] == 1 && visited[v] == 0)
            {
                visiting new vertices.
                printf("v%d.d", v);
                visited[v] = 1;
                enqueue(q, v);
            }
        }
    }
}

```

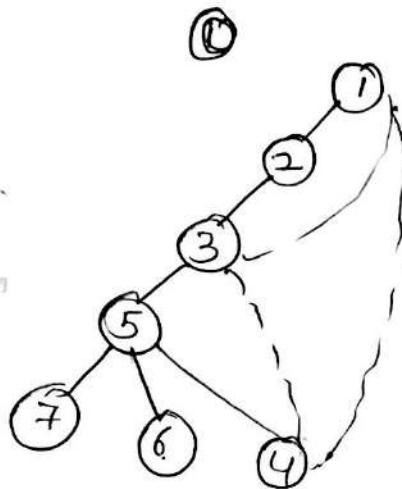
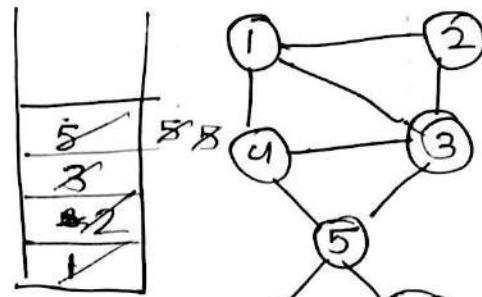
$\underline{\underline{O(n^2)}}$

31*

Depth First Search (DFS)

like Preorder Traversal

DFS: 1



DFS: 1, 2, 3, 5, 7, 6

Analytical time: $O(n)$

DFS Spanning Tree
↳ also DFS

since we are ~~doing~~
visiting all elements

1. 1, 3, 5, 4, 7, 6, 2

2. 1, 2, 3, 4, 5, 6, 7

3. 1, 4, 5, 7, 3, 2, 6

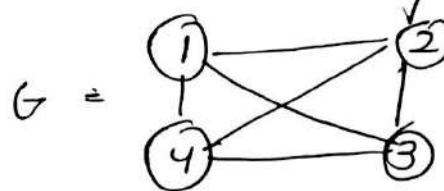
Program:

```

void DFS (int u)
{
    if (visited[u] == 0)
        print ("", d "", u);
    visited[u] = 1;
    for (v = 1; v <= n; v++)
        if ((A[u][v] == 1) && (visited[v] == 0))
            DFS(v);
}

```

Spanning Tree



sub graph of a graph

$$G = (V, E)$$

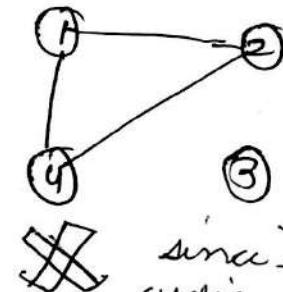
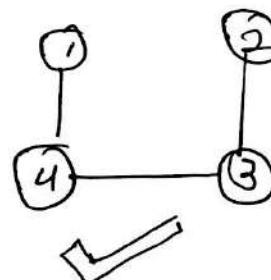
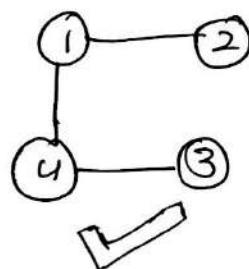
$$\begin{aligned} n &= |V| \\ e &= |E| \end{aligned}$$

$$S \subset G$$

sub graph

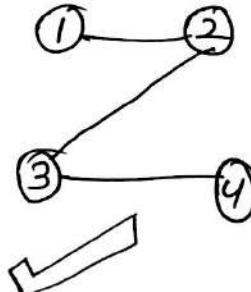
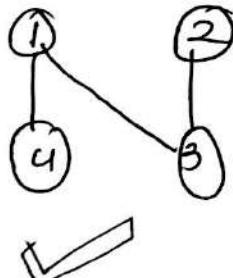
$$S = (V', E') \Rightarrow |V'| = V \Rightarrow |E'| = |V| - 1$$

Spanning Tree of G



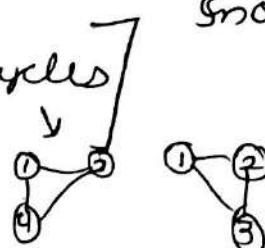
~~since it is cyclic.~~

~~& ③ is not connected~~

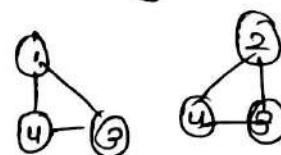


$$\begin{aligned} |V| &= 4 \\ |E| &= 6 \end{aligned}$$

No. of Spanning Trees = $C_{|V|-1}^{|E|} - \text{cycles}$



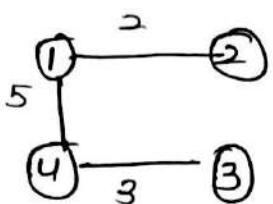
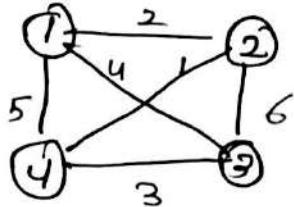
Given ex:
 $= 6C_3 - 4 = 20 - 4 = 16$



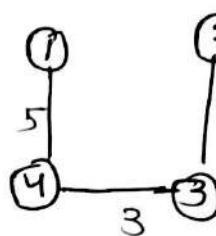
- -

If we gives weights to the edges

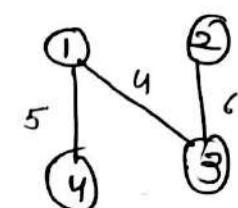
(32)



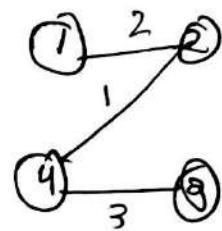
10



14



15



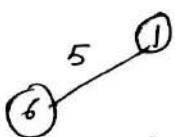
6

~~these~~ To find minimum ^{cost} Spanning Tree

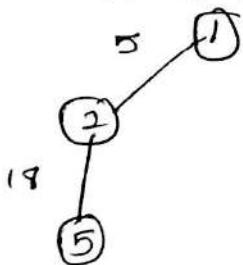
I Pram's Algorithm

~~First~~

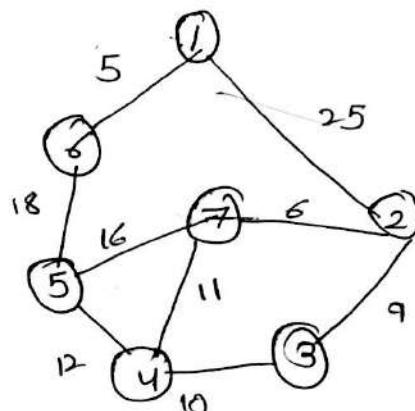
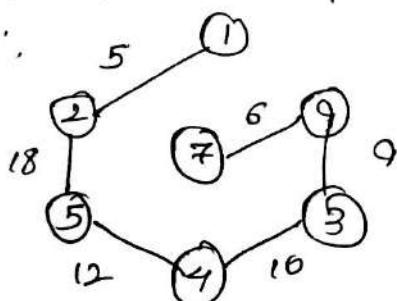
- Select minimum cost edge



- Then find minimum cost edge connected to these vertices
18 is min.



- Repeat 2 till we get $V-1$ edges.



Analysis.

$$(|V|-1) * |E| \\ (n-1 \text{ edges}) \\ (\text{are selected})$$

\rightarrow max we are checking
all edges for finding
min.

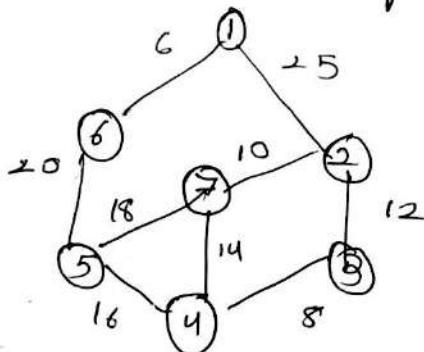
$$\underline{\underline{O(n^2)}}$$

or Red Black Tree

If we use Heap Method to find min.
Cost edge -

$$(|V|-1) \log(E) \\ \underline{\underline{O(n \log n)}}$$

Prism's Programs



0	1	2	3	4	5	6	7
0	-	-	-	-	-	-	-
1	-	-	25	-	-	-	-
2	-	25	-	12	-	-	5
3	-	-	12	-	8	-	-
4	-	-	-	8	-	16	-
5	-	-	-	-	16	-	14
6	-	5	-	-	-	20	18
7	-	-	10	-	19	18	-

(- is max. possible no.)

neare

0	1	2	3	4	5

t

1	1	1	1	1	1
6					

Find
min
from
upper
as
lower

→ ↴

Finding min in rows \rightarrow matrix.

near

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

+ put zero as near

1						
6						

0 1 2 3 4 5

Now compare for ②, ③, ④, ⑤, ⑥ that they are near to which node i.e. who is near (here ① & ⑥)

\therefore we will store it in near.

$$\therefore 2 \Rightarrow \text{cost}[2][1] = \infty$$

$$\text{cost}[2][6] = \infty$$

\therefore Near to ①

$$3 \Rightarrow \text{cost}[3][1] = \infty$$

$$\text{cost}[3][6] = \infty$$

Near to 6 (∞ take any)

$$4 \Rightarrow \text{cost}[4][1] = \infty$$

$$\text{cost}[4][6] = \infty$$

Near to 6

$$5 \Rightarrow \text{cost}[5][1] = \infty$$

$$\text{cost}[5][6] = \infty$$

Near to 6

$$7 \Rightarrow \text{cost}[7][1] = \infty$$

$$\text{cost}[7][6] = \infty$$

\therefore near to 6

near

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

Now see min among. near array & $\text{cost}[5][6]$, $\text{cost}[6][6]$, $\text{cost}[7][6]$

$$\text{cost}[7][6]$$

Here min is $\text{cost}[5][6]$

$\therefore 5 \& 6$ are stored in 6 nodes.

324

1	5					
6						
0	1	2	3	4	5	

Now update ⑤ & ⑥ in near to zero

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

Now update near for 1, 5, 6

$$\begin{array}{l} \Rightarrow \text{cost}[2][1] = 25 \quad \Rightarrow \text{cost}[3][1] = \infty \\ \text{cost}[2][5] = \infty \\ \text{cost}[2][6] = \infty \\ \therefore ② \text{ is near to } ① \end{array} \quad \begin{array}{l} \text{cost}[3][5] = \infty \\ \text{cost}[3][6] = \infty \\ \therefore \text{near to } ⑥ \end{array}$$

$$\begin{array}{l} \Rightarrow \text{cost}[4][1] = \infty \quad \Rightarrow \text{cost}[7][1] = \infty \\ \text{cost}[4][5] = 16 \\ \text{cost}[4][6] = \infty \\ \text{near to } ④ \quad ④ \quad ⑤ \end{array} \quad \begin{array}{l} \text{cost}[7][5] = 18 \\ \text{cost}[7][6] = \infty \\ \therefore \text{near to } ⑤ \end{array}$$

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

WHAT WE ARE DOING?

∴ we have found nearest node to each node which is taken in t .

Now we find min from all these nearest pair.

$$\begin{array}{l} \therefore [2][1] = 25 \quad [3][1] = \infty \quad [4][5] = 16 \quad [7][5] = 18 \\ \therefore \text{min is } [4][5] \end{array}$$

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

Now repeating same step

near	-	0		0	0	0	0
0	1	2	3	4	5	6	

∴ we do same steps till $t[5][7]$ is filled.

∴ finally

near $\boxed{-10|0|0|0|0|0|0}$

$+ \ 0$	1	5	4	3	2	2
$\times \ 2$	6	6	5	4	3	7

Program:

```
#define I 32767
int cost[8][8] = { {I, I, I, I, I, I, I, I},
                    {I, I, 25, I, I, I, 5, I},
                    {I, 25, I, 12, I, I, I, 10},
                    {I, I, I, I, I, I, I, I} }
```

```
int near[8] = { I, I, I, I, I, I, I, I }
int t[2][6];
void main()
{
```

```
    int i, j, k, u, v, n = 7, min = I,
        for (i = 1; i <= n; i++)
            for (j = i; j <= n; j++)
                if (cost[i][j] < min)
```

finding
min.
cost
edge
(by upper Δ)

```
                { min = cost[i][j];
                  u = i, v = j } }
```

```
t[0][0] = u; t[1][0] = v;
near[u] = near[v] = 0;
```

826

```
for (i=1; i<n; i++)  
{ if (near[i] == 0)  
    { if (cost[i][k] < cost[i][v])  
        near[i] = u  
    else  
        near[i] = v;  
 } }
```

// Now Repeating steps

```
for (i=1; i<n-1; i++)
```

finding
min
edge
 $k = 5$

```
{ min = I  
for (j=1; j<=n; j++)  
{ if (near[j] != 0 && cost[j][near[j]] < min)  
    { min = cost[j][near[j]]  
     k = j; } } }
```

updating
near
array.

```
t[0][i] = k, t[1][i] = near[k];  
near[k] = 0;  
for (j=1; j<=n; j++)  
{ if (near[j] != 0 && cost[j][k] < cost[j][near[j]])  
    near[j] = k; }
```

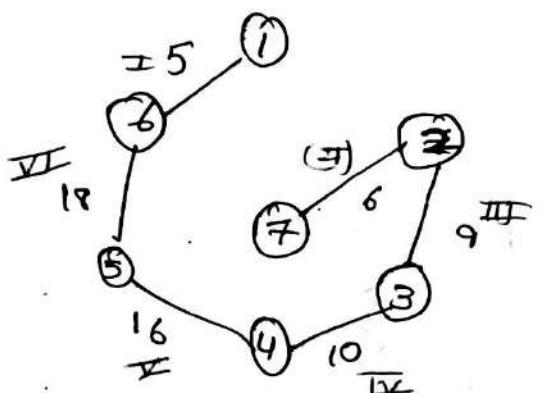
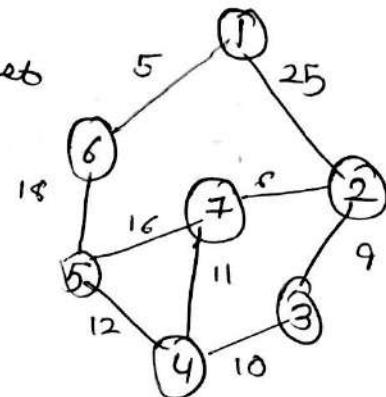
print t;

MAIN IS DECLARING DATA STRUCTURES

Kruskal's Minimum Cost ST

Procedure:

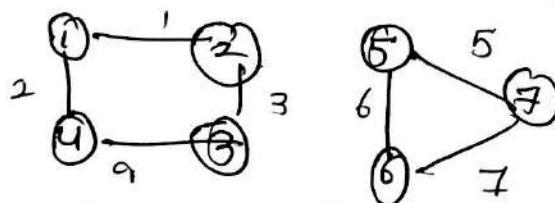
Always select one min. cost edge from graph & make sure selection of this edge is not forming cycle. If it is making cycle go to next min. edge.



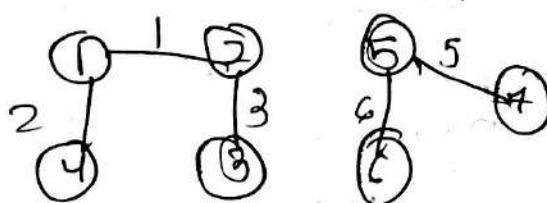
Time Complexity
 $(M-1)(E)$
 $n^2 = n \times n$
 $= O(n^2)$

Finding min by Min heap will be $\log n$
 Thus $O(n \log n)$

For non connected graphs



\therefore Kruskal's MST will result into



\therefore It finds ST of different components of non connected tree.

Disjoint Sets

$U = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$
 universal
 sub

Nodes:	1	2	3	4	5	6	7	8	9	10
S	-1	-1	-1	-1	4	5	6	7	8	10
	0	1	2	3	4	5	6	7	8	9

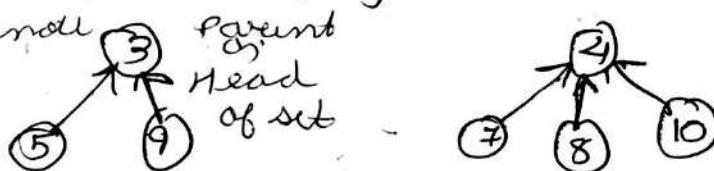
$$A = \{3, 5, 9\} \quad B = \{4, 7, 8, 10\} \quad A \cap B = \emptyset$$

Disjoint set
 $A \cap B = \emptyset$

-1 denotes it is itself a subset.

$$A = \{3, 5, 9\} \quad B = \{4, 7, 8, 10\}$$

underlined as



∴ we denote in data structure like

S	-1	-1	-1	-3	-4	3	-1	4	4	3	9
	0	1	2	3	4	5	6	7	8	9	10

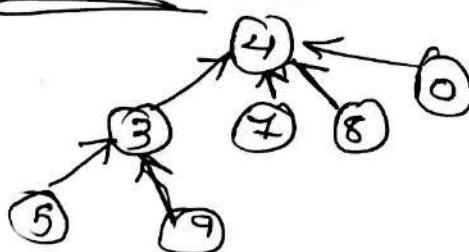
denotes
parent of
set having 3 elements

$$\text{Now } A \cup B = \{3, 4, 5, 7, 8, 9, 10\}$$

∴ Now we have to make a single parent

∴ A & B has no. of elements

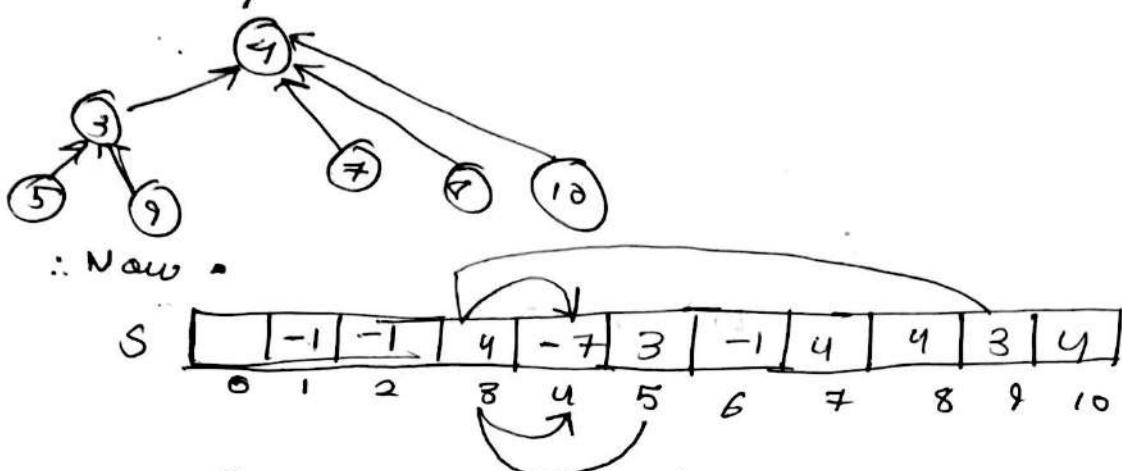
∴ 4 will be parent



Operations:

- Union
- find.

Union,



\therefore Parent of 5 & 9 is 4

void union (int u, int v)

```
{
    if (s[u] < s[v])
        {
```

```
        s[u] = s[u] + s[v]
        s[v] = u
    }
```

else

```
{
    if (s[v] < s[u])
        {
```

```
        s[v] = s[u] + s[v]
        s[u] = v;
    }
```

}

Q How to know whether by connecting it will form cycle or Not

If they are of same parent then don't connect Node. If they are of diff. parent then connect.

(330)

find

to find the parent of node

ex 4

$$s[4] = 7$$

$$s[3] = 4$$

$$s[4] = \text{---} 7$$

(--- denotes it is parent)

int find (int u)

{

int x = 0;

(while $s[x] > 0$)

{ x = s[x];

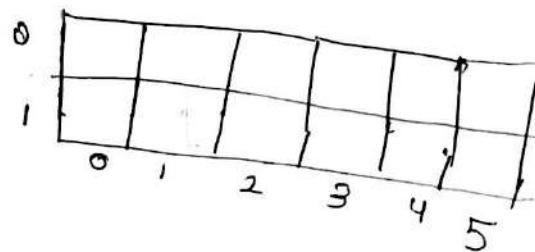
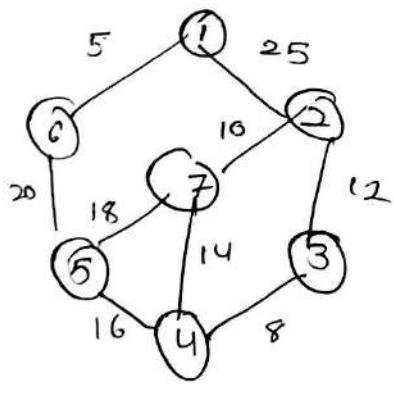
}

return x;

}

(367)

Kruskal's Programs



edges

	1	1	2	2	3	4	4	5	5
1	2	6	3	7	4	5	7	6	7
2	25	5	12	10	8	16	14	20	18

included

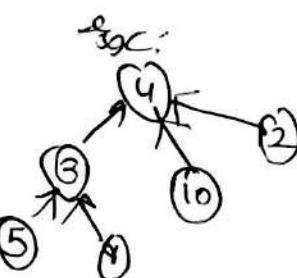
0	1	0	0	0	1	0	1
0	1	2	3	4	5	6	7

```

#include <stdio.h>
#define SS35
int edge[9][3] = {{1, 2, 8}, {2, 3, 16}, {3, 7, 14}}
}
int set[8] = {-1, -1, -1, -1, -1, -1, -1, -1}
int included[8] = {0, 0, 0, 0, 0, 0, 0, 0}
void join (int u, int v)
{
    if (set[u] < set[v]) union
    {
        set[v] += set[u]
        set[v] = u;
    }
    else
    {
        set[u] += set[v];
        set[u] = v;
    }
}

int find (int u)
{
    int x = u, v = 0
    while (set[x] > 0)
    {
        x = set[x];
    }
    while (u != x)
    {
        v = set[u];
        set[u] = x;
        u = v;
    }
    return x;
}

```



∴ 5 is parent of 4 since
parent set me -
not 3.

```

int t[2][7]; // Storing result.
int main (int)
{
    int u=0, v=0, i, j, k=0, min = 65535, n=9;
    while (i<6) // For selecting (t-1) edges.
    {
        min = 65535;
        for (j=0; j<n; j++)
        {
            if (included[j] == 0 && edge[j][i]<min)
            {
                u = edge[i][j];
                v = edge[j][i];
                min = edge[i][j];
                k = j;
            }
        }
        if (find(u) != find(v))
        {
            t[0][i] = u; t[1][i] = v;
            join(find(u), find(v));
            included[k] = 1;
        }
        else
        {
            included[k] = 1;
        }
    }
    printf("Spanning Tree (%d)\n", n);
    for (i=0; i<6; i++)
    {
        printf("%d %d ", t[0][i], t[1][i]);
    }
    return 0;
}

```

Kirchhoff's Theorem for Calculating no. of spanning trees of graph.

Case I: If G is complete graph.

$$\therefore \text{No of spanning trees} = n^{(n-2)}$$

(Cayley's Formula)

where n is no of vertices.

Case II If G is not complete

Polynomial

time complexity

Matrix - Tree / Kirchhoff's Algo.

Algo:

Step 1: Create Adjacency Matrix for the given graph.

Step 2 Replace all the diagonal elements with the degree of nodes. Ex: for pos (1, 1) it will be replaced by degree of node 1

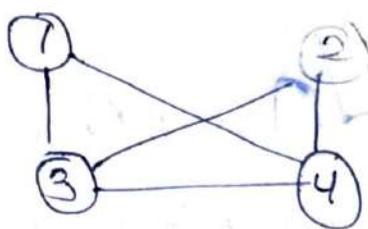
Step 3: Replace all diagonal 1's with -1

Step 4: Calculate cofactors for any element

Step 5: Cofactors will be the ans.

P.T.O

(2)



	Degree Matrix	Adjacency Matrix	Laplacian Matrix
1	$\begin{bmatrix} 1 & 0 & 3 & 4 \\ 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 3 \end{bmatrix}$	$\begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$	$\begin{bmatrix} 1 & 2 & -1 & 4 \\ 0 & 2 & -1 & -1 \\ -1 & -1 & 3 & -1 \\ -1 & -1 & -1 & 3 \end{bmatrix}$

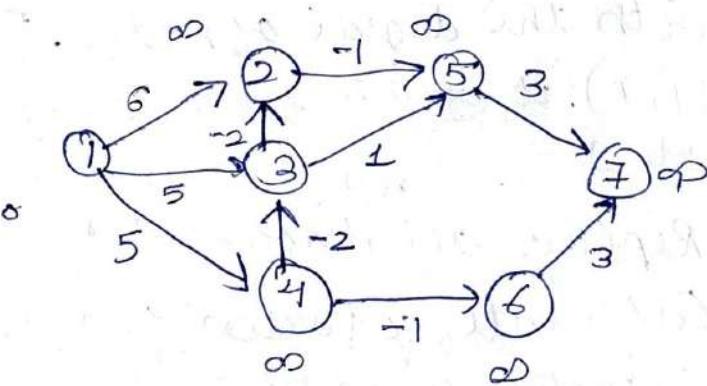
Now find the cofactors any element.

$$C_{ij} = (-1)^{i+j} |M_{ij}|$$

$$C_{11} = (-1)^2 * (2 * (3 * 3 - (1)) - (-1)(-3 - 1) + (1)(1)) \\ = 8$$

Bellman Ford Algo

Single source shortest Paths - DP



Now in this Algo we have to relax each edge $(V-1)$ times since max edge in spanning tree of size V is $(V-1)$.

Relaxation : if $d_{uv} + c(u,v) < d_{uv}$
 $d[u]^d = d[u] + c(u,v)$

considers all edges

↳ $(1,2), (1,3), (1,4), (2,5), (3,2), (3,5), (4,3),$
 $(4,6), (5,7), (6,7)$

Now traverse through all the edges
 $|V|-1$ times

Time complexity:

$O(|E| |V| - 1)$ $O(|V||E|)$

Average: $O(n^2)$

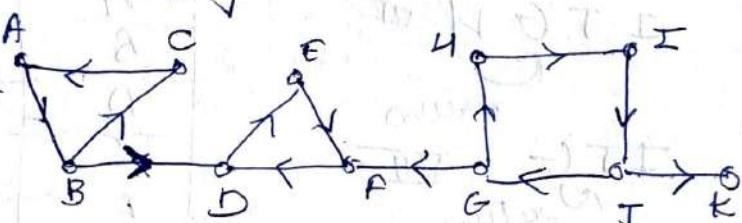
Worst case: For complete graph

$O\left(\frac{n(n-1)n}{2}\right) = O(n^3)$

[IMP: Tushar Roy - Coding Made Simple - YouTube]

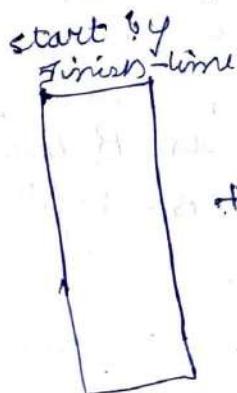
Kosaraju's algorithm

Strongly Connected Component.



For finding strongly connected we need to traverse graph twice & ~~marked~~

∴ we need



Now we will start with B & do DFS.

MEANING
Paths between
pair of
vertices -

IMP
Strongly
Connected
means
every vertex
can be visited
from every
other vertex

completely.
Connected
is every vertex
is connected
with other
vertices
by some edge

∴ First
the

B

B C

B C A Now returns I

B C returns II

B Now explores D

B → D

the B → D → E

B → D → E → F III

Now returns

B → D → E → F IV

returns

B → D returns II

B returns V

V	B
IV	D
III	E
II	F
I	C
=	A

Visited



Now take another vertex which is not visited

I

I J

I J G

I J G H II

G
returns

I J G VII

G
returns

I J K VIII

K
returns

I J IX

J
returns

I
returns

I	
II	J
III	K
IV	G
V	H
VI	B
VII	D
VIII	E
IX	F
X	C
=	A

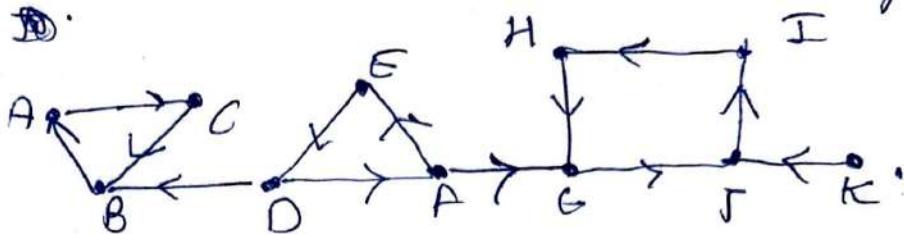
Visited
(all)

Stack

One concept is B will be on top of D & C since B & D are closely connected.

Now. 2nd Iterations

Now pop from stack and this does
 & first reverse all the edges.



(all edges reversed)

Now POP out from Stack.

I pop.

~~I H G J~~

I

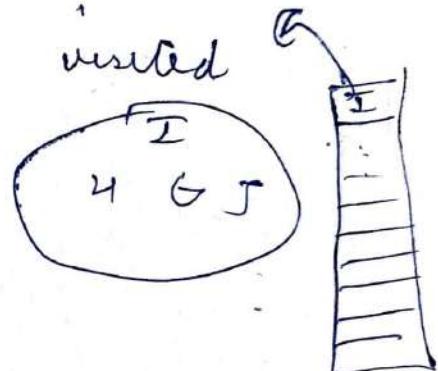
I H

I H G

~~I H G J~~

print

Now
cannot visit
any more



Now we will pop J, & do nothing since its already visited.

Pop K

DPS on K

print, ~~K~~ already visits : return

Now pop G, H

New node B

so

print B C A

Now pop D

print D F E

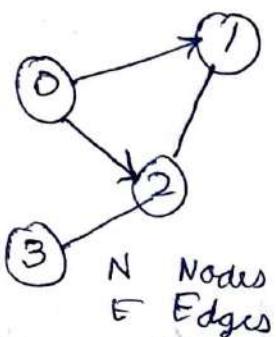
Now pop E, F, A, C

& stack empty.

visited



Graph Representation



Directed Edge

Undirected \leftrightarrow Bidirectional edge.

	0	1	2	3
0	T	T		
1		T		
2	T		T	
3	T			T

① Adj Matrix

Rows & Col: N^2

⇒ Neighbors
 $O(N)$

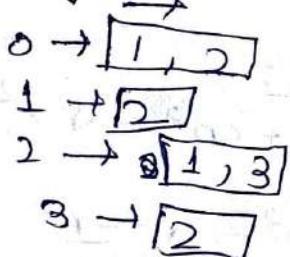
$$V = \{0, 1, 2, 3\}$$

$$E = [(0,1), (0,2), (1,2), (2,1), (2,3), (3,2)]$$

Edge list

② Edges & vertices

③ Adj list



Quicksort

$O(N \log N)$

Memory

efficient
 $O(E)$

④ Implicit Graph

0	0	1	1
1	1	0	1
2	0	0	1
3	0	0	0

↑ 2D array
+ ← 1 → +
↓

Given all vertices
that are connected
to each other.

Adjacency list Implementation

```

class Graph {
    int v;
    list<int> *l;
public:
    Graph(int v) {
        this->v = v;
        l = new list<int>[v];
    }
}
    
```



Array of list of size V

0	→ [1, 2]	L_1
1	→ [2, 0]	L_2
2	→ [3, 0, 1]	L_3
3	→ [2]	L_4

(2)

```
void addEdge(int x, int y) {
```

```
    l[x].push_back(y),  
    l[y].push_back(x);
```

```
}
```

Adjacency List General Implementation

we need to store in form of:

$A \rightarrow (B, 20) (D, 50) (C, 10)$

$$\text{list} < \text{pair} < \text{string, int} >, CD^2 = r^2 = \frac{r^2}{4}$$

Key value

$$CD = \frac{\sqrt{r^2}}{2}$$

∴ we will create Hash map.

class Graph

$$\frac{1}{2} \times r \times \frac{\sqrt{3}r}{2}$$

// Adj list

unordered_map<string,

list<pair<string, int>> l;

public:

```
void addEdge(string x, string y, bool bidir, int wt)
```

```
{
```

```
    l[x].push_back(make_pair(y, wt));
```

```
    if (bidir) {
```

```
        l[y].push_back(make_pair(x, wt));
```

(3)

```

void printAdjlist(11)
    "Iterate over all the key in map"
    for (auto p : l) {
        string dest = p.first;
        int dist = p.second;
        cout << dest <<
        string city = p.first;
        list<pair<string, int> nbrs = p.second;
        for (auto nbr : nbrs) {
            string dest = nbr.first;
            int dist = nbr.second;
            cout << dest << " " << dist << ", ";
        }
        cout << endl;
    }

```

BFS traversal. (level order)

```

#include <iostream>
using namespace std;

```

```

template <typename T>
class Graph {

```

```
    map<T, list<T>> l;
```

public:

```
    void addEdge (T x, T y) {

```

```
        l[x].push_back(y);

```

```
        l[y].push_back(x);
    }
```

④

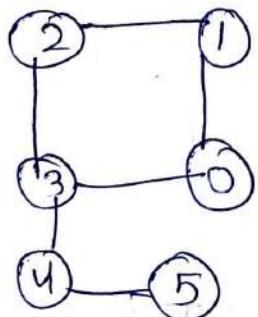
```

void bfs(T src) {
    map<T, int> visited;
    queue<T> q;
    q.push(src);
    visited[src] = 1;
    cout << src;
    while (!q.empty()) {
        T node = q.front();
        q.pop();
        for (auto nbr : l(node)) {
            if (visited[nbr] == 0) {
                q.push(nbr);
                cout << nbr << " ";
                visited[nbr] = true;
            }
        }
    }
}

```

Single Source Shortest Paths

- UNWEIGHTED GRAPH
- BFS for Single Source



source

↓

dist

0 - 1 - 2 - 3 - 4 - 5
0 - 3 - 4 - 5 ✓

```

void shortest ( T src )
{
    m ap < T, int > dist;
    queue< T > q;
    q.push(src);

    for ( auto node-pair : l )
        T node = node-pair . first;
        dist [ node ] = INT-MAX;
    }

    q.push ( src );
    dist [ src ] = 0;

    while ( ! q.empty () )
        T node = q.front ();
        q.pop();

        for ( int nbr : l [ node ] )
            if ( dist [ nbr ] <= INT-MAX ) {
                q.push ( nbr );
                dist [ nbr ] = dist [ node ] + 1;
            }
}

// Printing
for ( auto node-pair : l )
    T node = node-pair . first;
    int d = dist [ node ];
    cout << " Node " << node << " Dist from src " << d;
}

```

⑥

Snake & Ladder Problem

DFS Gives shortest Path

Minimum No. of Die Throws Required to reach the destination starting from source.

Find the shortest paths as well.

5 Snakes

12, 34
4, 17

2 ladders

2, 15 del(2 to 15)
18 29

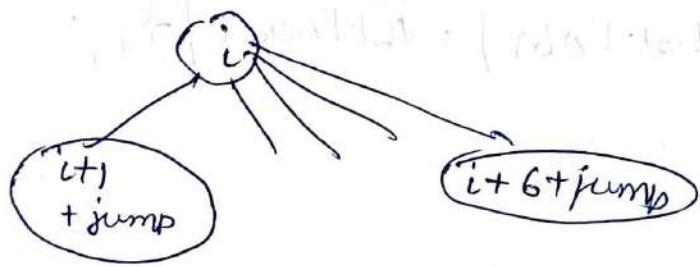
It is a Graph

V \rightarrow 0, ..., 36
E \rightarrow

Value Verb
 $i \rightarrow i + 1$ + jump \uparrow →
 $\rightarrow i + 2$
 $\rightarrow i + 3$
 $\rightarrow i + 4$
 $\rightarrow i + 5$
 $\rightarrow i + 6$

↑
(snake)

Now this Graph will be unweighted graph
since we need to calc. Die Throws.



Here Graph will be created & we will
modify our graph class accordingly.

int board[50] = {0};

board[2] = 13;

board[3] = 2;

board[9] = 18;

board[18] = 11

board[17] = -13

board[20] = -14

board[24] = -8

board[25] = 10

board[32] = -2

board[34] = -22

// Add Edges to the Graph

Graph g;

for (int i = 0; i < 36; i++) {

 for (int dice = 1; dice <= 6; dice++) {

 int j = i + dice;

 j += board[j];

 if (j <= 36)

 g.addEdge(i, j);

 }

 g.addEdge(36, 36);

 shortest(0, 36);

 return 0;

(8)

Shortest is same as ~~for~~ prev. code 2 changes

(ii)

void shortest(~~T~~ src, ~~T~~ dest)

1

// No count of dist

return dist[dest];

4

To get the shortest Path we do

map<T, T> parent;

parent[src] = src;

while(!q.empty()) {

for (int n6r : q) {

if (_____)

q.push(n6r);

dist[n6r] = dist[node] + 1;

parent[n6r] = node;

}

}

temp = dest;

while (temp != src) {

cout << temp << " - ";

temp = parent[temp];

}

cout << src << endl;

return dist[dest];

J

DFS

In class Graph

class Graph

public:

```
void dfs_helpers (int src, map<T, bool> &visited)
```

```
{ cout << src << " "
```

```
visited[src] = true;
```

```
for (T nbr : l[src]) {
```

```
if (!visited[nbr]) {
```

```
dfs_helpers (nbr, visited);
```

```
}
```

Y ~~defn helpers~~

```
void dfs (T src) {
```

```
map<T, bool> visited;
```

```
for (auto p : l) {
```

```
T node = p.first;
```

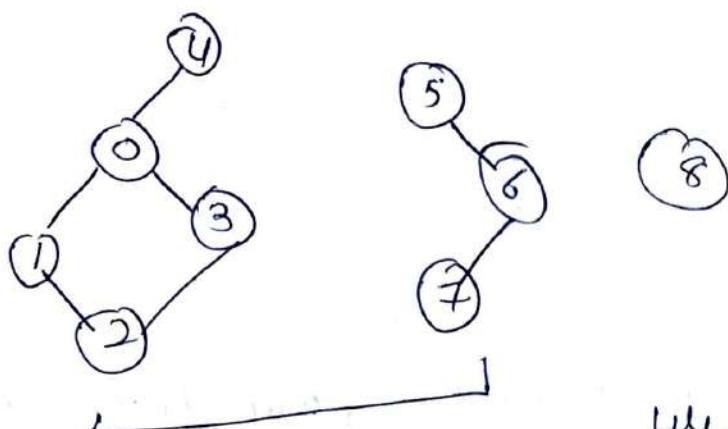
```
visited[node] = false;
```

```
dfs_helpers (src, visited);
```

Y

(8)

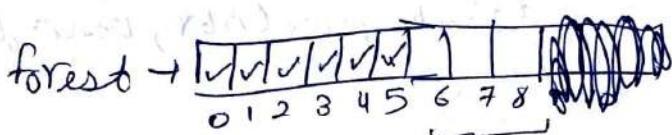
Undirected Connected Graph



$$5 + 3 = 8 \text{ Nodes}$$

how many
connected
are present.

We can use
BFS or DFS for
getting no. of connected
graphs.



DFS 0 : 0 1 2 3 4

DFS 5 : 5 6 7

DFS 8 : 8

// Change in dfs function.

void dfs() {

map<T, bool> visited;

for (auto p:l) {

T node = p.first;

visited[node] = false;

}

// Iterate over all the vertices & init a dfs call.

for (auto p:l) {

T node = p.first;

int cnt=0 // no. of connected graphs.

if (!visited[node])

```

    cout << "Component " << cnt << " --> ";
    dfs_helper(node, visited);
    cnt++;
    cout << endl;
}

```

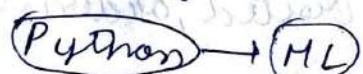
DFS Traversal

DAG Topological Sort using DFS:

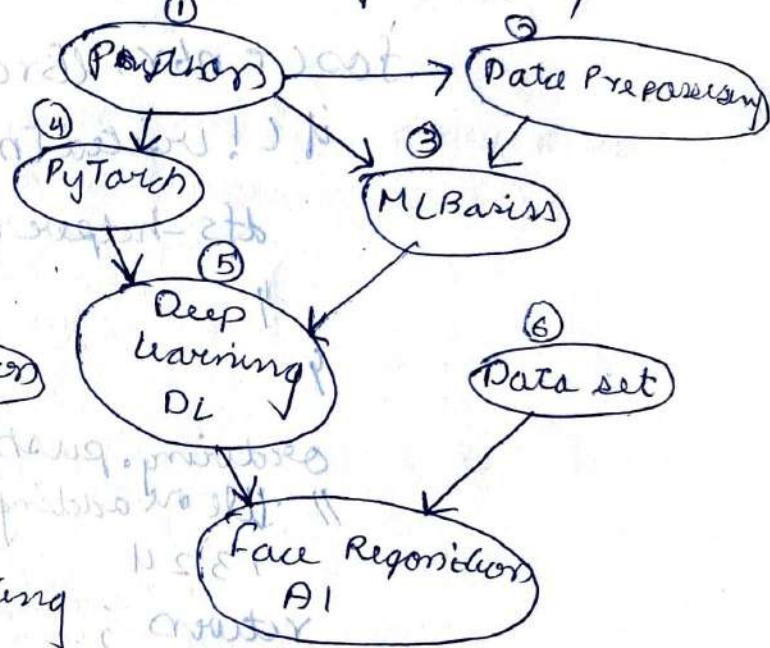
Topological Sorting

$$2 \rightarrow 4$$

4 has 2 as its dependency



Dependency Graph

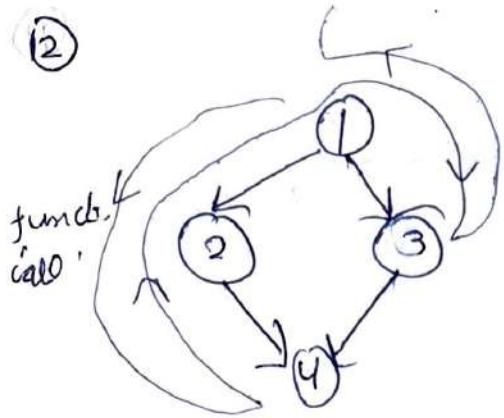


Output: A linear ordering



① ② ③ ④ ⑤ ⑥ ⑦

Topological Sortings is only for Directed Acyclic Graph



ordering

~~call neighbour's names~~

Implementation :

```
void dfs_helpers(T src, map<T,goal> &visited  
list<T> &ordering) {
```

```

visited[src] = true;
for (T nbr : ll[src]) {
    if (!visited[nbr]) {
        dfs-helper(nbr, visited, ordering);
    }
}

```

ordering, push-front (SYC)
!! ~~the~~ not adding front since it convert will

13) 24 returns ; prescribed processes (A. required)

void fsc() {

map < T, 600 > ; i, i-

list $\langle T \rangle$ ordering.

for cause $p : \ell$) {

$T \text{ node} = p_0 \text{ first};$

visited [node] = false;

(3)

```

for(auto p:l) {
    T_node = p.first;
    if (!visited[node]) {
        dfs-helps (node, visited, ordering);
    }
}

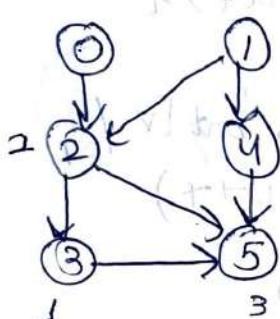
```

if finally to print the list

```

for (auto node : ordering) cout << node << endl;
}
};
```

Topological Sort using BFS



we will calculate indegree of every node and start exploring the one with 0 indegree

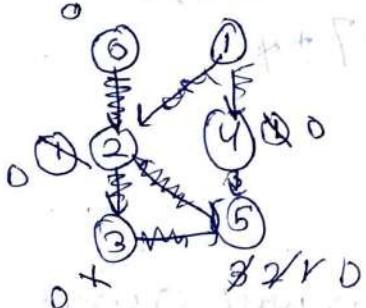
Now will add nodes with indegree 0 in list

0	1	
---	---	--

Now explore 0 : neighbour of 0 : nodes connected with 0 will subtract 1 edge from indegrees

X	1	
---	---	--

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



(14)

class Graph {

```
list < int > *l;
int v;
```

public:

Graph(int v) :

this->v = v,

l = new list < int > [v],

|

void addEdge(int x, int y) {

l[x].push-back(y);

C++ private implementation

void topologicalSort() {

Initializing indegree array

```
int *indegree = new int[v];
for (int i = 0; i < v; i++)
    indegree[i] = 0;
```

```
// update indegree by traversing edge x → y
indegree[y] += 1;
```

```
for (int i = 0; i < v; i++) {
```

```
    for (auto y : l[i])
```

```
        indegree[y] += 1;
```

```
if fs
```

```
queue < int > q;
```

```
1. step find nodes with 0 indegrees
```

initializing

q with pushing

mdegree 0 to

vertices

```
for (int i = 0; i < v; i++)
```

```
{ if (indegree[i] == 0) {
```

```
q.push(i);
```

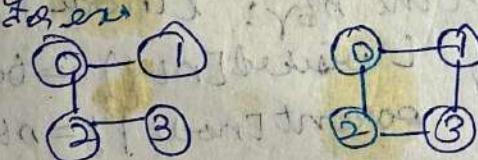
```

    // start removing elements from queue
    while (!q.empty()) {
        int node = q.front();
        q.pop();
        cout << node << endl;
        if (previous
            node) // iterate over neighbors of current node
            and
            reduce their indegree by 1
        for (auto nbr: l[node]) {
            indegree[nbr]--;
            if (indegree[nbr] == 0) {
                q.push(nbr);
            }
        }
    }
}

```

Check of Undirected Graph is Tree or Not.

A graph will be tree when not cyclic.



tree
Not a tree

Ques changes into If Graph is cyclic
or not.

If traversal we found that
a visited node is visited again (except
the case that visited is parent of that
node) will result in cycle.

(16)

Code:

Graph is Array of linked list

class Graph()

{
=}

public:

bool isBcc()

{ bool *visited = new bool[V];

int *parent = new int[V];

// initializing parent & visited node

for(int i = 0; i < V; i++) {

visited[i] = false;

parent[i] = -1; // initialize every node

is parent of itself.

int src = 0;

q.push(src);

visited[src] = true;

// pop out one node & visit its nbrs

while(!q.empty()) {

int node = q.front();

q.pop();

for(int nbr : l[node]) {

if (!visited[nbr] == true and

parent[node] != nbr) {

visited[nbr] = true;

parent[nbr] = node;

q.push(nbr);

}

}

}

int main()

{ Graph g(V)

g.addEdge(0, 1)

" (3,2)

" (1,2)

" (0,3)

return 0;

}

visited[nbr] = true;

parent[nbr] = node;

q.push(nbr);

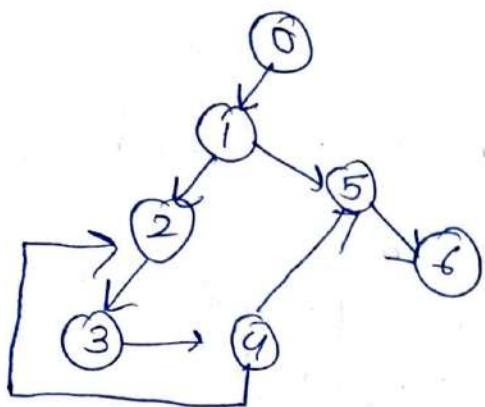
}

return true;

};

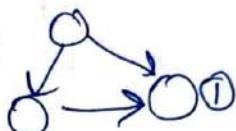
17

Cycle Detection in Directed Graph



we will do
with help of DFS.

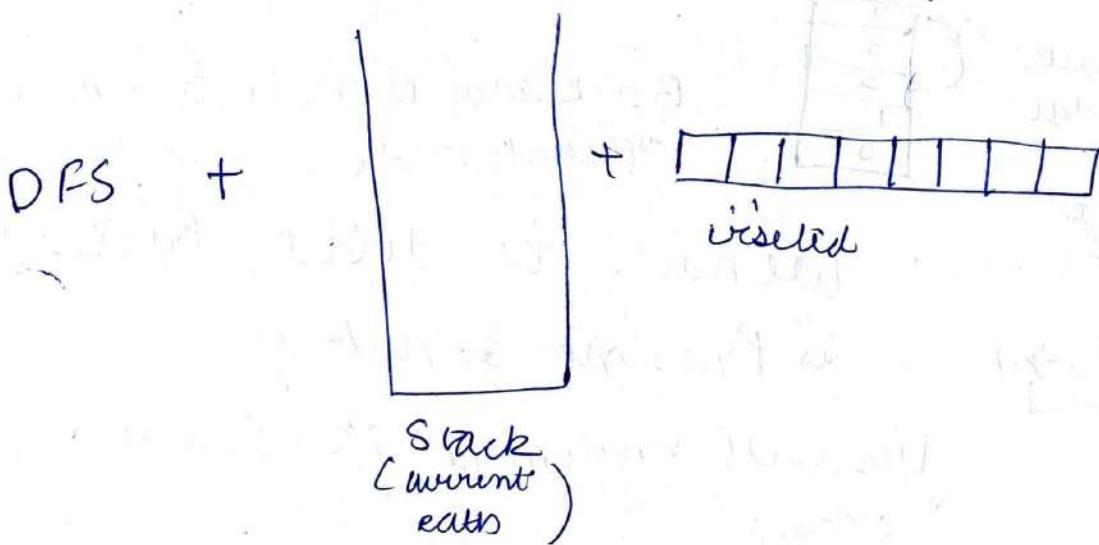
Here the algorithm of
directed graph may not
work because
ex.



~~Two edges are incoming to~~
here 0 will be visited twice
~~thus it's not cyclic~~

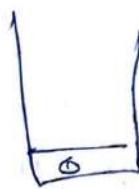
Here the CONCEPT is when a node
is visited to the node i.e. in ~~stack~~
between one path, of traversal to the
current node creates a cycle

Now to get nodes which are in between the
paths ~~nodes~~. we will use stack.

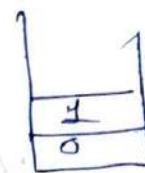


18

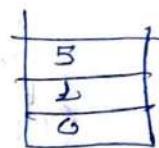
Visiting 0



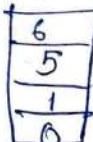
Visiting 1



Visiting 3



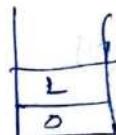
Visiting 6



Returning to 5



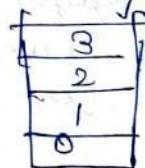
Returning to 1



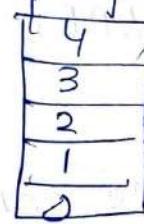
Visiting 2



Visiting 3



Visiting 4



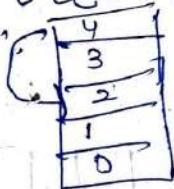
Now at Node 4 we can visit to 5 & 2

Now both are visited ~~so~~ But5 does not lie in current path ~~so~~

but 2 lies in the current path

So $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4$

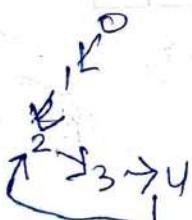
Back edge



Back edge is node to a ancestor of that node.

∴ We have to detect Back edge is present or not?

We will implement stack as an "array"



bool cycle - helpers (int node, bool *visited, bool *stack)

// visit a node

visited [node] = true;

stack [index] = true;

for (int nbr : l [node])

{ // two cases

if (stack [nbr] == true)

{

return true;

}

else if (visited [nbr] == false)

{

bool cycle - mila = cycle - helpers (nbr, visited, stack);

THOUGHT

here use node
to recursively
call kar aur

Pucha ki cycle
mila if yes then
return true

else see other
neighbours

if (cycle - mila == true)

{ return true; }

}

{

// leave a node

stack [node] = false

return false;

}

\$

bool contains - cycle ()

// check for cycle in Directed Graph

bool *visited = new bool [V];

bool *stack = new bool [V];

~~return cycle - helpers~~

[For (int i = 0; i < V; i++) {

visited [i] = stack [i] = false;

return cycle - helpers (0, visited, stack);

Initialising

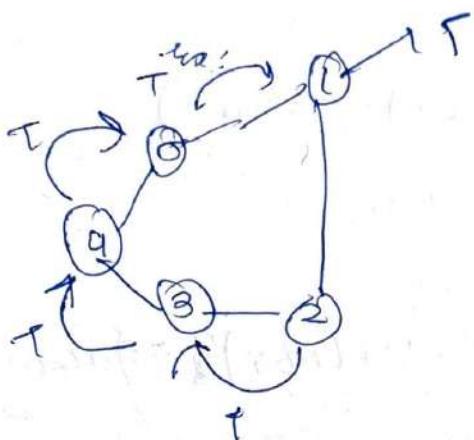
Y

(28)

Cycle Detection in Undirected Graphs

DFS

If there exists a node that we are visiting in the traversal & that node is already visited



bool cycle_helper (int node, bool &visited, int parent) {

 visited[node] = true;

 for (auto nbr : l[node]) {

 // two cases

 if (!visited[nbr]) {

 // go and recursively visit the nbr

 bool cycle_mild = cycle_helper(nbr, visited, node);

 if (cycle_mild == true)

 return true;

}

 a nbr is & visited but nbr should not be equal to parent.

 else if (nbr != parent) {

 return true;

}

 } // returns false;

(21)

```

bool contains_cycle() {
    // Check if cycle is Directed Graph
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++) {
        visited[i] = false;
    }
    return cycle_helper(0, visited, -1);
}

```

FLOOD FILL ALGORITHM

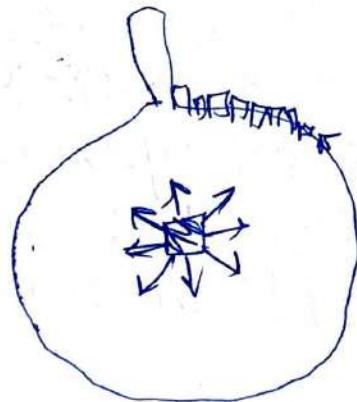
(Applications of DFS)

→ (colouring)

used to color various component in graph

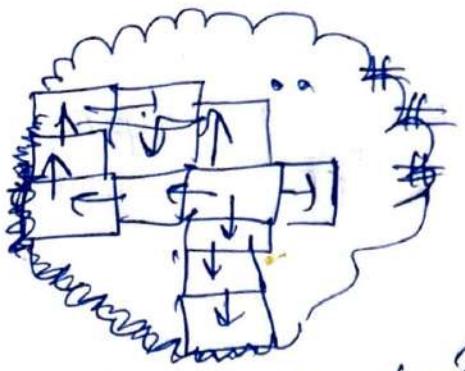
generally performed on implicit graph

Starting from a particular cell we can call 'DFS' on the neighbouring cell to colour them. Neighbors can be '4' (up, down, \leftarrow , \rightarrow) or '8' if we include diagonal case.



∴ we won't be call the recursion to go further & further until it reaches pixel black

(22)



ch is •

bie characters to
be replaced.

all possible directions

$(0, -1)$
$(0, 1)$
$(1, 0)$
$(1, 1)$

int dx[] = {1, -1, 0, 1, 0};

int dy[] = {0, 1, 0, 1, 1};

void floodfill (char mat[50], int i, int j, char ch, char color) {

// Base case - Matrix Boundary

if (i <= 0 || i >= 5 || j >= R || j >= C) {
 return ;
}

// Figure Boundary Condition

if (mat[i][j] != ch)
 return ;

// Recurssion Call

mat[i][j] = color;

for (int k = 0; k < 5; k++) {

floodfill (mat, i + dx[k], j + dy[k],

ch, color);

}

Here you can see we don't need visited array since once we colors a pixel it bie characters change
 $\therefore \text{mat}[i][j] \neq ch$ & will return

```
int main ()  
{    cin >> R >> C;  
    char input[15][50];  
    for (int i=0; i<R; i++) {  
        for (int j=0; j<C; j++) {  
            cin >> input[i][j];  
        }  
    }  
  
    printMatC(input);  
    floodfill (input, 18, 13, '.', 'x');  
    print (input);  
    return 0;  
}
```

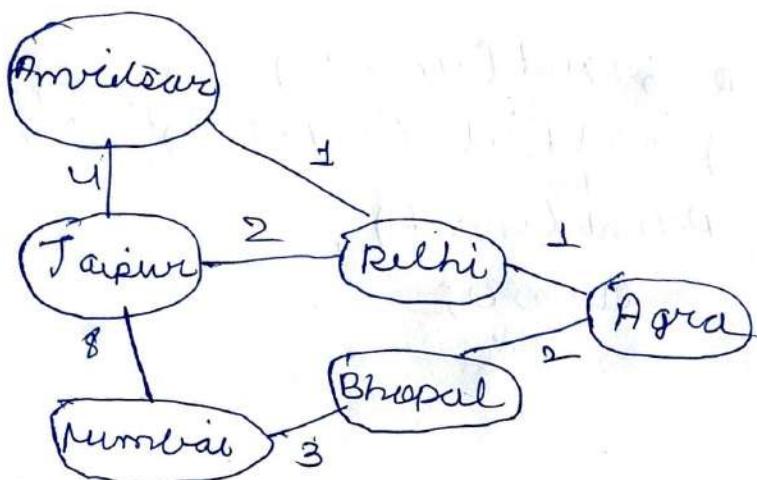
Dijkstra's Algorithm (+as weighted graph)

[SSSP]

single source to all other nodes.

NOTE: No -ve weight

Edges can be uni or Bi directional



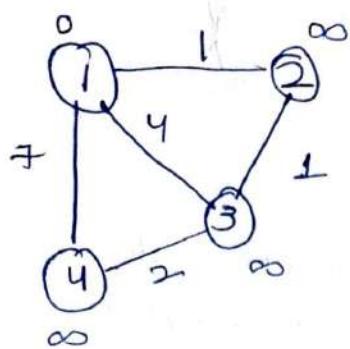
Adjacency list.

key	list of pair (string, int)
Amritsar	(Delhi, 1) (Jaipur, 4)
Jaipur	(Delhi, 2) (Mumbai, 8)
Mumbai	(Bhopal, 3) (Jaipur, 8)
Bhopal	(Agra, 2) (Mumbai, 3)
Delhi	(Amritsar, 1), (Jaipur, 2) (Agra, 1)
Agra	(Delhi, 1) (Bhopal, 2)

∴ Generalising how to make Adjacency list.

hashmap < T, list < pair < T, int > >
 map < K, unordred_map >

(25)



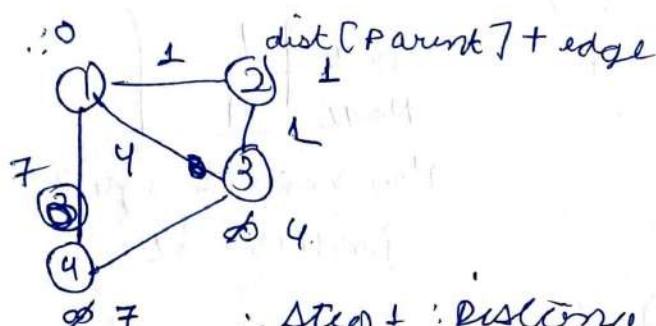
we want to find the with min -
imum distance from 1?

use
array
 $O(N)$

Priority Queue (set)

find min: $O(1)$
remove id: $\log N$

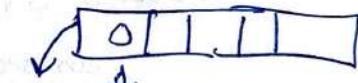
✓ Better way



\therefore Step 1: distance is being updated
Step 2: Pick node having minimum
distance

Set (stores element in sorted manner)
~~Defn~~ according to the first row
 set in pair.

Initially, it's having



Remove, it and

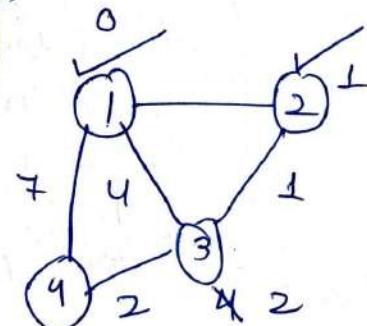
do step 2

\therefore Node 1 has been finalized

Now neighbors will go in set

dist.	1	4	7	
Node	2	3	4	
remove node				

i. remove



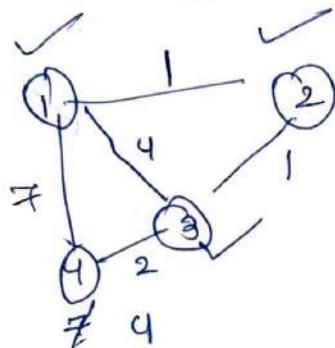
\therefore Node 3's dist. is being updated

Now in set no update func? \therefore we have to first
remove & then add the updated node.

26 ie Remove(3) & then add (3, 2)

list	<table border="1"> <tr> <td>2</td><td>7</td></tr> <tr> <td>3</td><td>4</td></tr> </table>	2	7	3	4	list
2	7					
3	4					
node						

Now remove ③ & finalize it
 ∵ we will see ~~dist~~ distance of nodes connected
 with ③ which are not finalized



list	<table border="1"> <tr> <td>4</td><td></td></tr> <tr> <td>4</td><td></td></tr> </table>	4		4	
4					
4					
node					

Now remove & just
 finalize it

∴ 1 → 0
 2 → 1
 3 → 2
 4 → 4

{ Total Source to all other
 notes.

Program :

class Graph {

unordered_map<T, list<pair<T, int>> m;

public:

void addEdge (T u, T v, int dist, bool bidir

Adjacency List {

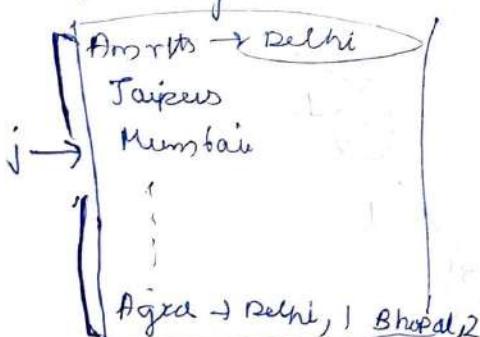
m[u].push_back(make_pair(v, dist));
 if (bidir) {

m[v].push_back(make_pair(u, dist));

void printAdj () {

for (auto i : m) {

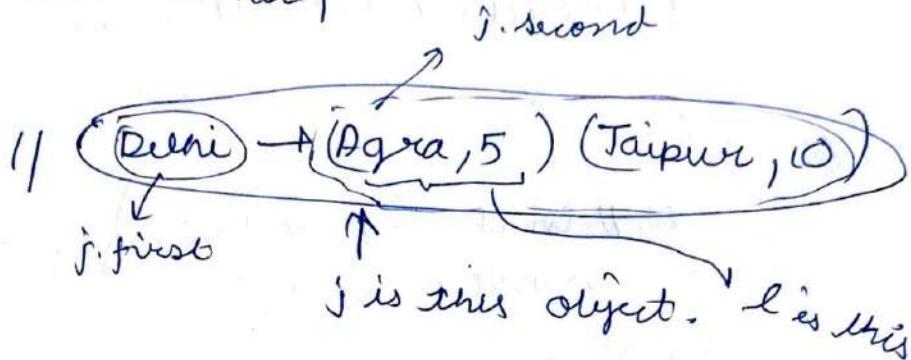
cout << j.first << " → ",



```

for (auto l : i.second) {
    cout << "C" << l.first << "," << l.second << ")";
}
cout << endl;
}

```



```
void dijkstra SSSP(T src)
```

```
{
    unordered_map<T, int> dist;
```

```
// set all distance to  $\infty$ 
```

```
for (auto j : m) {
```

```
    dist[j.first] = INT_MAX;
```

```
}
```

// MAKE a set to find out node with the minimum distance.

```
set<pair<int, T>> s;
```

```
dist[src] = 0;
```

```
s.insert(make_pair(0, src));
```

```
while (!s.empty()) {
```

// Find the pair at ~~min~~ from

```
auto p = *s.begin();
```

```
T node = p.second;
```

```
int nodeDist = p.first;
```

```
s.erase(s.begin());
```

// Iterate over neighbours of current node.

(28)

// Generate our neighbours / children of current node

for auto childPair : m[node] {

if (nodeDist + childPair.second) < dist

{

[childPair]
* first

// In the set update of a particular
is not possible

// we have to remove the old pair
and insert the new pair to signify
update of update.

T dest = childPair.first;

auto f = s.find(make_pair(dist[dest],
dest));

if (f != s.end())

{ s.erase(f),

}

// Insert the new pair

dist[dest] = nodeDist + childPair.second;

s.insert(make_pair(dist[dest], dest));

}

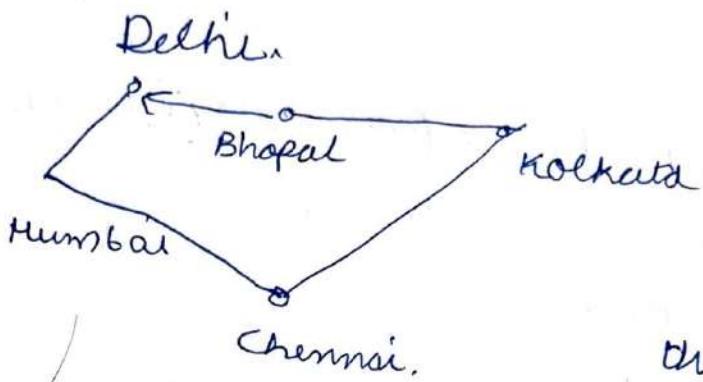
// For getting the path we need
to make Parent map

Parent[node] ✓

Print the path

Travelling Salesman Problem

(IMP) Questions



Starts from Delhi,
& returns to original source.

We have to
minimize
the total distance
travelled by him.

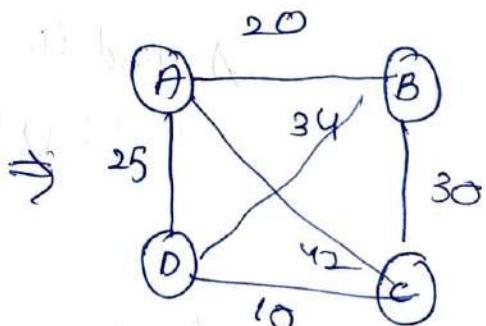
Such a cycle is called Hamiltonian cycle
ie covering all nodes of Graph and
coming to some node.

Set of edges such that every node is visited
and comes back to starting node.

For the above problem we need
to find min. weight Hamiltonian
cycle.

~~dist~~
Input

	A	B	C	D
A	0	20	42	25
B	20	0	30	34
C	42	30	0	10
D	25	34	10	0



Dist[i][j]

30

Hamiltonian cycles are

start from A : A - B - C - D - A } cyclic permutations
 if we start from B : B - C - D - A - B } of A B C D A

\therefore Brute force for all the Hamilton cycle will see permutations of all nodes

$\therefore n!$ i.e. $A \ B \ C \ D$
 $B \ C \ A \ D$
 all are permutations

$\therefore O(n!)$

Now ~~to~~ calc. weight of all these $n!$ ~~Hamilton~~ permutations to find weight of each Hamilton cycle

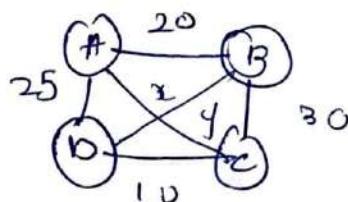
Since $A \ B \ C \ D$ & $B \ C \ D \ A$, $C \ D \ A \ B$,
 $D \ A \ B \ C$ all are cyclic permutations of each other & have equal weight
 so we find the starting node as A and then permute $(n-1)$ nodes

$\therefore O((n-1)!)$ Pretty time consuming

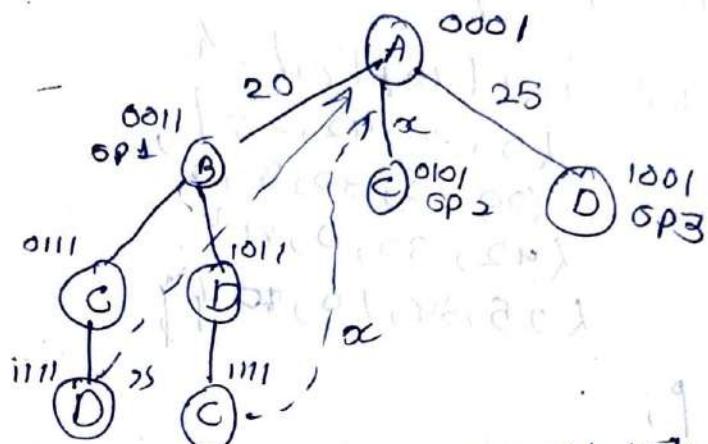
(31)

Dynamic Programming

(Top-Down DP)



Store set of
visited
so far in
current branch



so in to call D should return cost
from D to A.

if visited - all

use BITMASKING

mask = 0000 (no visited)
mask = 1111 (all visited)
= $2^4 - 1$
= $(1 \ll 4) - 1$

if visited - all

return cost[city][0];

cost + Revision

A → (B + C + D + A)

subproblem

$\{ \begin{matrix} 20 + OP_1 \\ 25 + OP_2 \\ 30 + OP_3 \end{matrix} \}$ min is any

Code:

```
#include <iostream>
using namespace std;
```

```
#define INT_MAX 999999
```

```
int n = 4;
```

```
int dist[10][10] = {  
    {0, 20, 42, 25},  
    {20, 0, 30, 34},  
    {42, 30, 0, 14},  
    {25, 34, 14, 0}}
```

;

// If all cities have been visited
int VISITED_ALL = (1 << n) - 1;

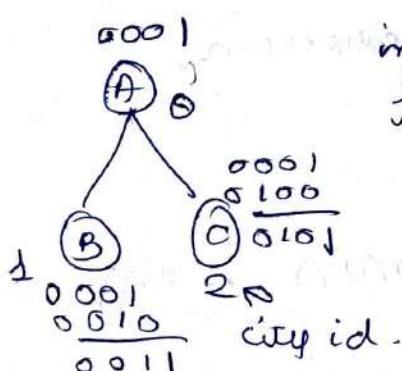
, int top, int mask, int pos

{
 if (mask == VISITED_ALL)

return dist[pos][0];

(E)

// Try to go to i-th city



city
int ans = INT_MAX;

for (int city = 0; city < n; city++) {

if ((mask & (1 << city)) == 0)

{

int newAns = dist[pos][city] + tsp(
 mask | (1 << city), city)

}

ans = min (ans, newAns); // id of city.

}

return ans; // for dp its return
dp[mask](pos) = ans;

}

THEORY

mask:

0000

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

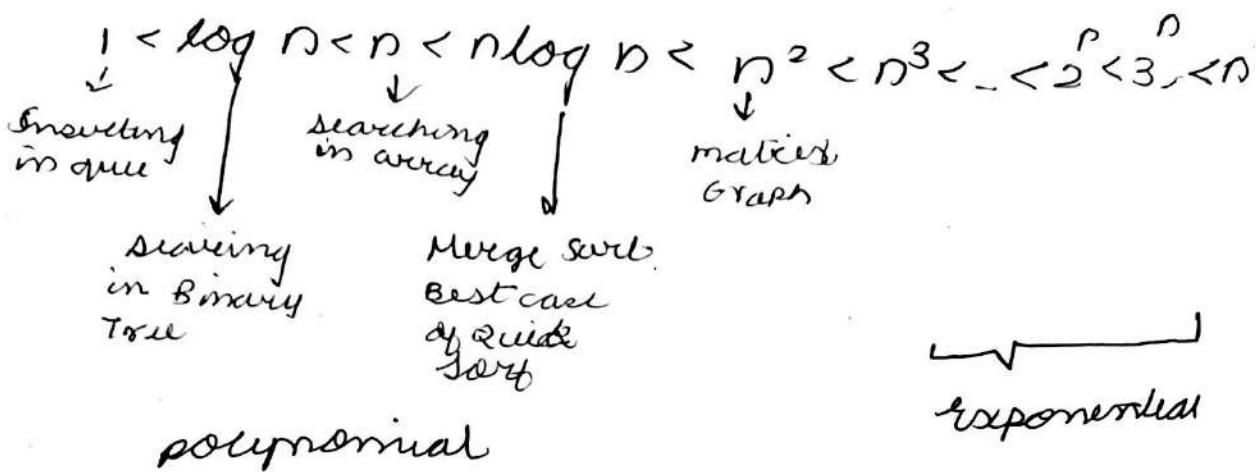
|

|

|

|

Asymptotic Notations



$$f(n) = n \quad O(n)$$

$$f(n) = n^2 \quad O(n^2)$$

$$f(n) = \sum_{i=1}^n i \Rightarrow 1+2+3+\dots+n = \frac{n(n+1)}{2} = O(n^2)$$

$f(n) = \sum_{i=0}^n i \times 2^i$. Now we cannot get polynomial simply.

$$2^{(2^n)} \quad O(n^{2^n})$$

$$O(n^{2^n})$$

Lower Bound

Upper Bound

Exact Bound

$\Omega(\text{big omega})$ is generally used
since we need upper bound of
algorithms

(534)

(When we know exact order also then
also we prefer to say Θ than $\Omega(\theta)$)
since

$$-2(n) = f(n)$$

$$\Theta(n) = \Omega(n)$$

$f(n) = \Theta(n)$ i.e. $-2, \Theta, \Omega$ are same.