



DATA STRUCTURE USING C



- Way of organising data
- Data should be efficiently used by the program

VARIOUS DATA STRUCTURES

- ARRAY
- MATRIX
- LINKED LIST

Physical Data Structure
(How data is arranged)

- STACK
- QUEUE
- TREE
- GRAPH
- HASHING

Linear
Non Linear
Tabular

Logical Data Structure
(How data is utilized)

- RECURSION
- SORTING TECHNIQUE

Logical Data structures are implemented using physical Data Structures



BASICS

1. Arrays
2. Structure
3. Pointer
4. Reference
5. Parameters Passing
6. Classes
7. Constructor
8. Templates

ARRAYS

Collection of similar datatypes

STRUCTURES

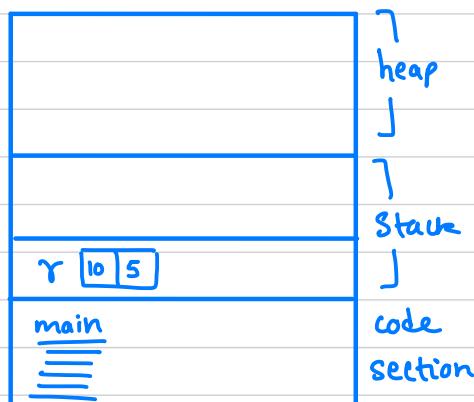
Collection of different data types

Struct rectangle

```
int length; — 2
int breadth; — 2
}; 4 bytes
```

but till here, it does not occupy any space in memory

Main memory



int main()

```
Struct rectangle r = {10,5} → here it occupies memory
r.length = 10;
r.breadth = 5;
```

}

Struct card

```
int face;
int shape;
int colour;
};
```

int main()

```
{ struct card deck[52] = {---}
printf("%d", deck[0].face);
printf("%d", deck[0].shape);}
```

}

POINTERS

→ address variables

1. Why pointers
2. Declaration
3. Initialization
4. Dereferencing
5. Dynamic allocation

```
int a=10;  
int *p; → declaration  
p=&a; → initialization  
printf("%d", a);  
printf("%d", *p); → dereferencing
```

USES OF POINTER

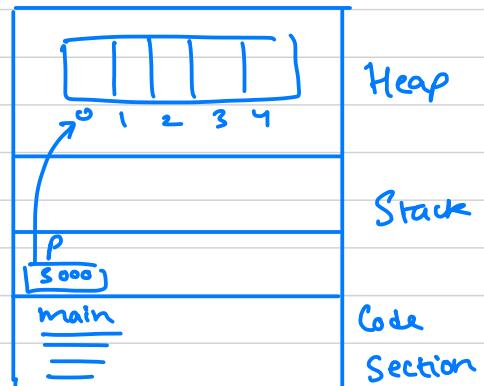
1. Accessing heap
2. Accessing resources like keyboard or mouse.
3. Parameter passing.

X ————— X

HOW TO USE POINTER FOR ALLOCATING HEAP

#include < stdlib.h > → header file for malloc()

```
int main()  
{  
    int *p;  
    p = (int *) malloc(5 * sizeof(int));  
    ↴ for returning  
    ↴ as malloc function  
    ↴ returns void pointer  
    ↴ (type casting)  
    ↴ Creating space for 5  
    ↴ integer values in heap.
```



X ————— X

REFERENCE IN C++

```
int main()  
{  
    int a=10;  
    int dr=a;  
    cout << a;  
    cout << r;  
}
```



POINTER TO STRUCTURE

```

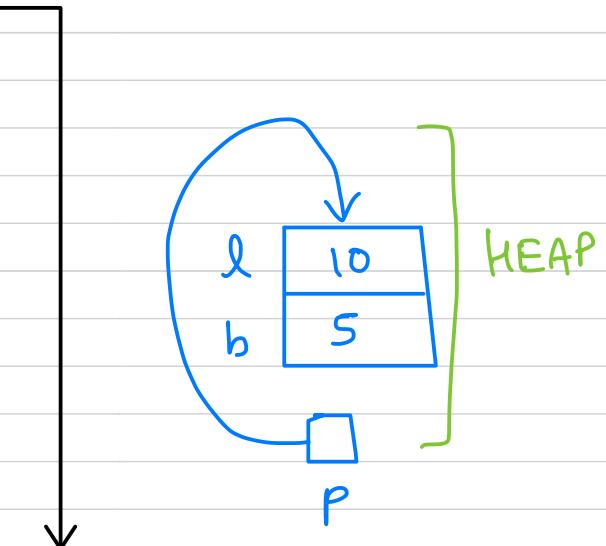
Struct rectangle
{
    int length;
    int breadth;
};

int main()
{
    struct rectangle r = {10, 5};
    struct rectangle *p = &r;

    ✓ r.length = 15;
    ✓ (*p).length = 20;
    ✓ p->length = 20;
}

```

doesn't occupy
4 bytes of
memory
(2 bytes)



```

int main()
{
    struct rectangle * p ;
    p = (struct rectangle *) malloc ( sizeof ( struct rectangle ) )

    p->length = 10;
    p->breadth = 5;
}

```

FUNCTIONS

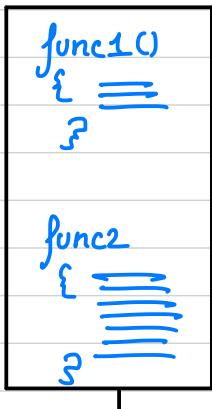
What are functions → Performs a specific task

Parameters of passing functions

- Pass by value
 - Pass by address
 - Pass by reference
-] Only in C] In C++



MONOLITHIC
PROGRAMMING



MODULAR
PROGRAMMING

FUNCTION EXAMPLE

prototype

```

int add(int a, int b)
{
    int c;
    c = a + b;
    return c;
}

```

Formal Parameter

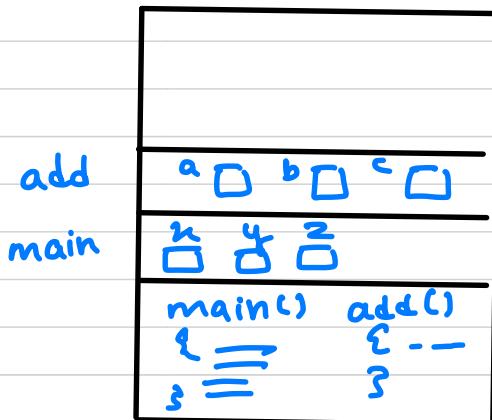
```

int main()
{
    int x, y, z;
    x = 10;
    y = 5;
    z = add(x, y);
    printf("%d", z);
}

```

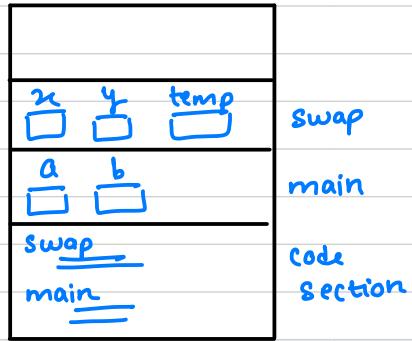
Actual Parameter

15



CALL BY VALUE

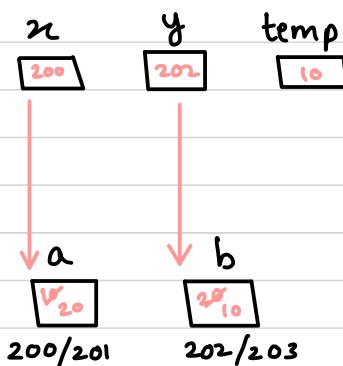
```
Void swap( int x ,int y )
{
    int temp ;
    x = y ;
    y = temp ;
}
```



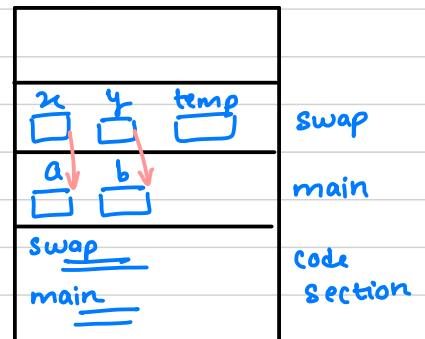
```
Int main()
{
    Int a, b;
    a = 10;
    b = 20;
    swap(a,b);
    printf ("%d %d",a,b);
}
10 20
```

CALL BY ADDRESS

```
Void swap( int *x ,int *y )
{
    int temp ;
    *x = *y ;
    *y = temp ;
}
```



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

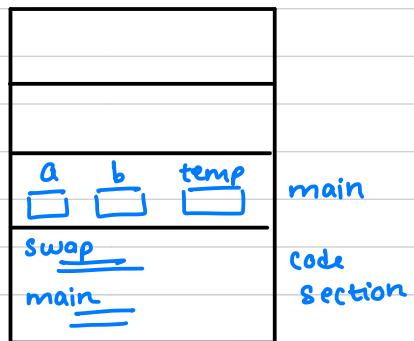


CALL BY REFERENCE (Only in C++)

```
Void swap( int &x , int y )  
{  
    int temp ;  
    x = y ;  
    y = temp ;  
}
```

a/x
20
200/201

b/y
10
202/203



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

temp
10

Only 2 bytes of memory is used as in reference, only another name is given to the pre-existing variable

ARRAY AS PARAMETER

```
void fun( int A[], int n )  
{  
    int i ;  
    for ( i = 0 ; i < n ; i++ )  
        printf ( " %d " , A[ i ] ) ;  
}
```

→ or int *A as ARRAYS can only be passed as call by address.

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

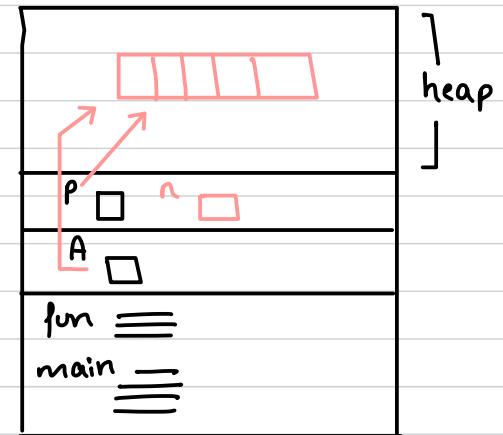
↑ or int * fun(int n)

```

int [] fun(int n)
{
    int * p;
    p = (int *) malloc(n * sizeof(int));
    return p;
}

```

it returns an array



```

int main()
{
    int * A;
    A = fun(5);
}

```

STRUCTURE AS PARAMETER

CALL BY VALUE

```

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

```

```

struct rectangle
{
    int length;
    int breadth;
};

```

```

int main()
{
    struct rectangle r = {10, 5};
    printf ("%d", area(r));
}

```

CALL BY REFERENCE

```
int area(struct rectangle *r1)
```

CALL BY ADDRESS

```
void changel ( struct rectangle *p, int l )
```

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

```
int main()
```

```
struct rectangle r = {10, 5};
changel (&r, 20);
```

PASSING ARRAYS AS CALL BY VALUE USING STRUCTURES

```
Void fun(struct test t1)
{
    t1.A[0] = 10;
}
```

```
int main()
{
    struct test t={ {2,4,6,8,10}, 5 };
    fun(t);
}
```

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

2	4	6	8	10
5				

changes made in the array in the function will not be reflected as it is call by value

STRUCTURES AND FUNCTIONS

```
Struct rectangle
{
```

```
    int length;
    int breadth;
}
```

```
Void initialize (Struct rectangle *r, int l, int b)
{
```

```
    r->length = l;
    r->breadth = b;
}
```

```
int main()
{
```

```
    Struct rectangle r;
```

```
    initialize (&r, 10, 5);
    area (r);
    change (&r, 20);
}
```

```
int area (Struct rectangle r)
{
    return r.length * r.breadth;
}
```

l	10
b	5

```
void change (Struct rectangle *r, int l)
{
    r->length = l;
}
```

because values need to be changed, so call by address.

CLASS AND CONSTRUCTOR

```
class rectangle
{
```

private:

```
    int length;
    int breadth;
```

public:

```
rectangle (int l, int b)
{
```

```
    length = l;
    breadth = b;
```

}

```
int area()
{
```

```
    return length * breadth;
```

}

```
void changelength (int l)
{
```

```
    length = l;
```

}

};

```
class rectangle
{
```

private:

```
    int length;
    int breadth;
```

public:

```
rectangle ()
{
```

```
    length = breadth = l;
```

```
}
```

```
rectangle (int l, int b);
```

```
int area();
```

```
int perimeter();
```

constructors
(overloaded)

facilitators
(They do
operations on data members)

Accessor

```
int getlength
{
```

```
    return length;
```

```
}
```

```
void setlength (int l)
```

```
{
```

```
    length = l;
```

```
}
```

Mutator

Destructor → ~ Rectangle();

```
{
```

```
rectangle :: rectangle (int l, int b)
```

```
{
```

```
    length = l;
```

```
    breadth = b;
```

```
}
```

```
int rectangle :: area()
```

```
{
```

```
    return length * breadth;
```

```
}
```

```
int rectangle :: perimeter()
```

```
{
```

```
    return 2 * (length + breadth);
```

```
}
```

```
rectangle :: ~rectangle()
```

```
{
```

```
int main()
```

```
{
```

```
    rectangle r(10, 5);
```

```
    cout << r.area;
```

```
    cout << r.perimeter;
```

```
    r.setlength(20);
```

```
    cout << r.getlength();
```

TEMPLATE CLASS

template < class T >

class arithmetic
{

private :

T int a;
T int b;

public :

T T
arithmetic (int a, int b)
T int add ();
T int sub ();

};

for using various datatypes.
Using the same class for
different datatypes.

template < class T > → because effect of previous template

arithmetic < T > :: arithmetic (int a, int b) has ended

{

this · a = a;

this · b = b;

}

template < class T >

T int arithmetic :: add ()
{

T int c;

c = a + b;

return c;

}

template < class T >

int arithmetic :: sub ()
{

T int c;

c = a - b;

return c;

}

int main ()

{

arithmetic < int > ar (10, 5);

cout << ar.add ();

arithmetic < float > ar1 (1.5, 1.2);

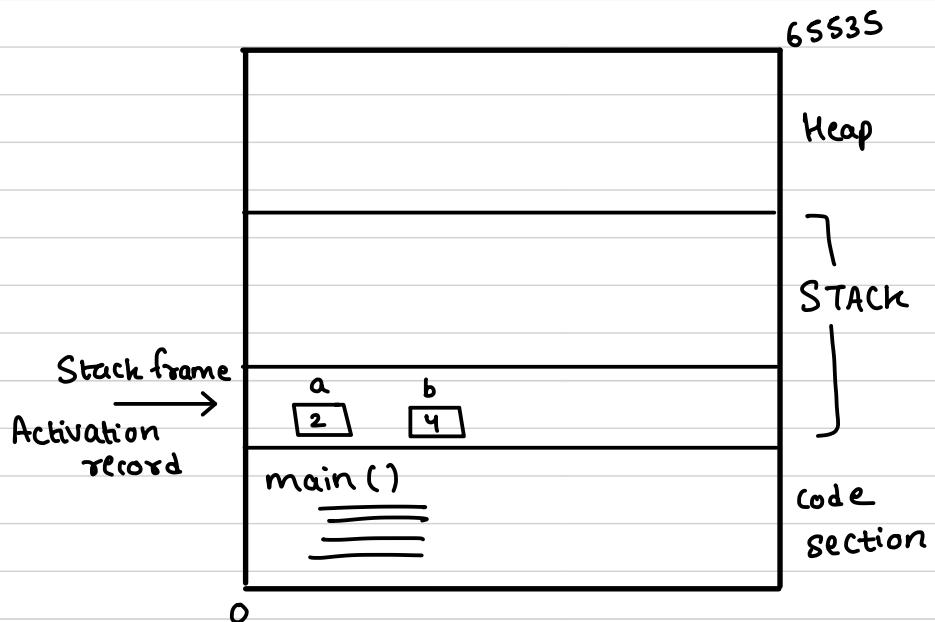
cout << ar1.add ();

}

INTRODUCTION

STATIC VS DYNAMIC MEMORY ALLOCATION

```
void main()
{
    int a;
    float b;
}
```



```
void fun2(int i)
{
```

```
    int a;
    ===
```

```
}
```

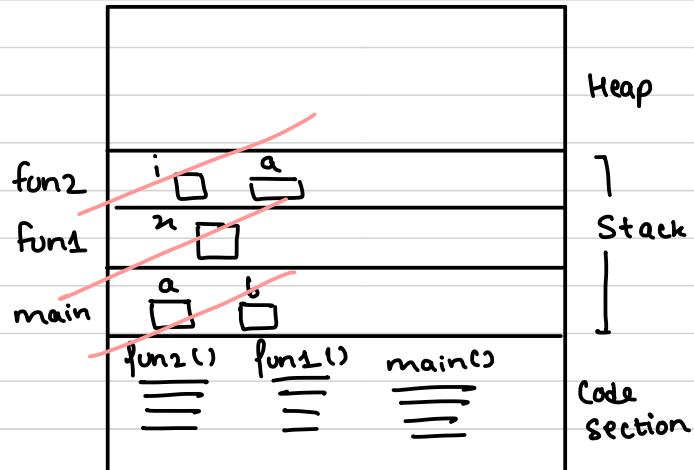
```
void fun1 ()
{
    int x;
    fun2(x);
}
```

```
}
```

```
void main()
{
```

```
    int a;
    float b;
    fun1();
}
```

HOW DOES STACK WORK?



1. Stack is always organised memory.

HOW DOES HEAP MEMORY WORK?

1. Heap may be organised memory or unorganised memory.
2. Heap must be treated as a resource i.e when it is needed, we must use it and when not needed, free it so that it can be used by other applications.

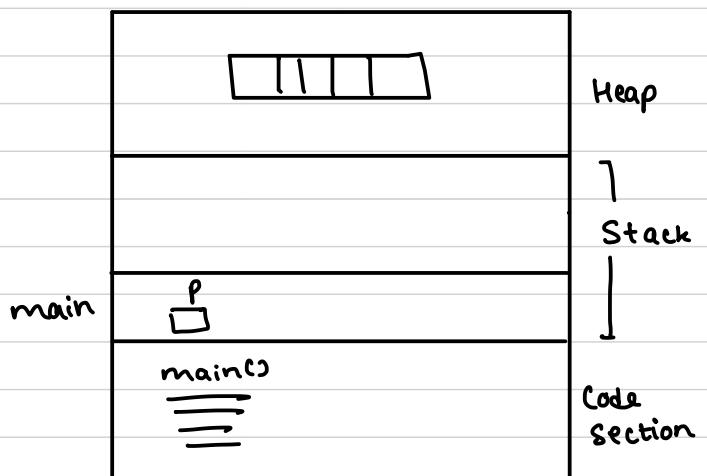
For eg - printer is a resource.

```
void main()
{
    int *p;

    for (int i = 0; i < 5; i++)
        p[i] = i;

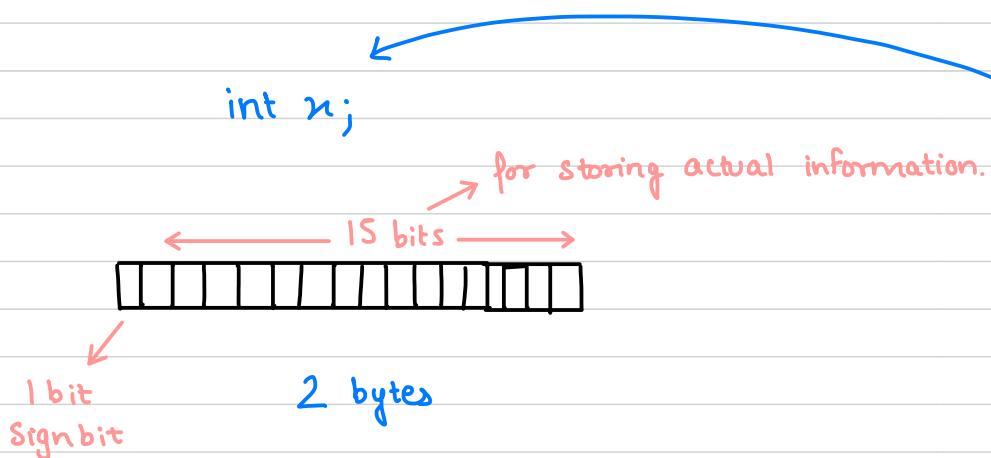
    for (int i = 0; i < 5; i++)
        cout << p[i] << " ";
}

DEALLOCATION
delete []p;
p = NULL;
```



DATATYPE?

- ↳ 1. Representation of data.
2. Operation on data.



ABSTRACT DATATYPE

↳ hiding internal details

(part of object oriented programming)

What does log mean?

$\log_2 n \rightarrow$ This gets divided by this until it reaches 1.

Time and Space complexity

```
void swap(n,y)
```

```
{
```

```
    int t;  
    t=n; —————|  
    n=y; —————|  
    y=t; —————|  
            3
```

```
}
```

$O(1)$

```
int sum(int A[], int n)
```

```
{
```

```
    int s,i;  
    s=0; —————|  
    for(i=0; i<n; i++) ————— n+1  
    { —————|  
        s = s + A[i]; ————— n  
    }  
}
```

```
}
```

$\frac{1}{2n+3}$

$O(n)$

because i will be initialized (+1) and i will be incremented for n no of times and 1 time it will fail too (+n)

// because one time it will fail too.

```
Void Add (int n)
{
```

```
    int i, j;
```

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

```
{
```

```
        for (j=0; j<n; j++)
    {
```

```
        c[i][j] = A[i][j] + B[i][j];
    }
```

```
}
```

```
}
```

$n+1$

$n^*(n+1)$

n^*n

because the nested loop
will run again and
again for $(n+1)$ no of
times.

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

$$O(n^2)$$



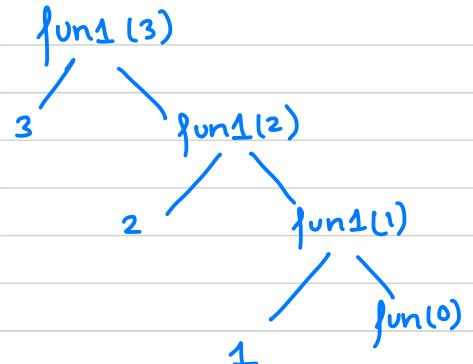
RECURSION

→ When a function calls itself

EXAMPLE #1

```
void fun1(int n)
{
    if (n > 0)
    {
        printf("%d", n);
        fun1(n-1);
    }
}
```

TRACING TREE



```
void main()
{
    int n = 3;
    fun1(n);
}
```

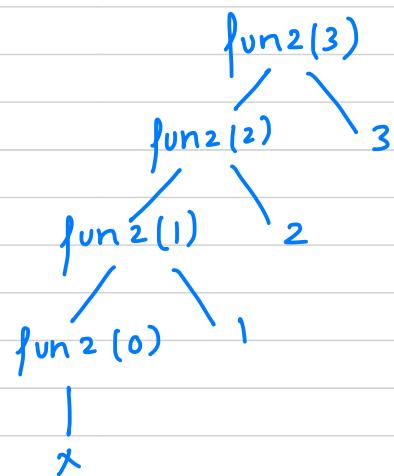
OUTPUT

3 2 1

EXAMPLE #2

```
void fun2(int n)
{
    if (n > 0)
    {
        fun2(n-1);
        printf("%d", n);
    }
}
```

TRACING TREE



```
void main()
{
    int n = 3;
    fun2(n);
}
```

OUTPUT

1 2 3

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

```
    if (n > 0)
```

```
{
```

ASCENDING 1. calling

2. fun(n-1)

DESCENDING 3. returning

```
}
```

```
}
```

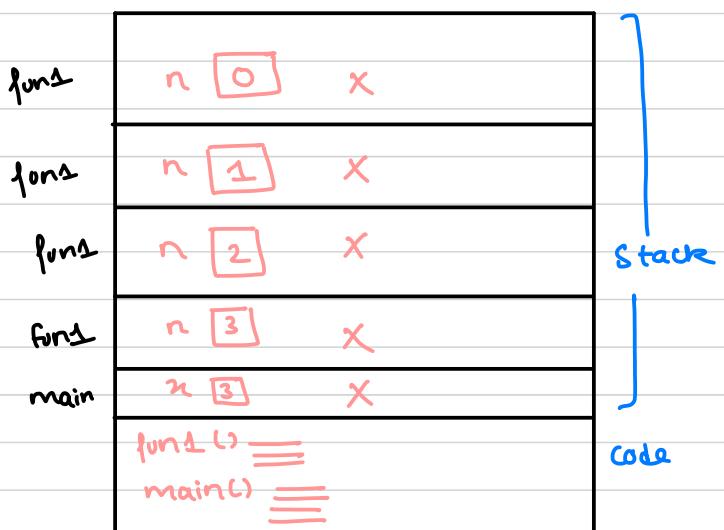
DIFFERENCE BETWEEN LOOP AND RECURSION

The main difference between loop and recursion is that recursion allows two phases i.e ascending and descending while loop allows only ascending.

HOW STACK IS UTILISED IN RECURSIVE FUNCTIONS?

EXAMPLE #1

```
void fun1(int n)
{
    if (n > 0)
    {
        printf("%d", n);
        fun1(n-1);
    }
}
```



```
void main()
{
```

```
    int n = 3;
    fun1(n);
}
```

Here, value of n was 3, so there were 4 calls
(Tracing tree).

So for value of n, there will be $n+1$ calls

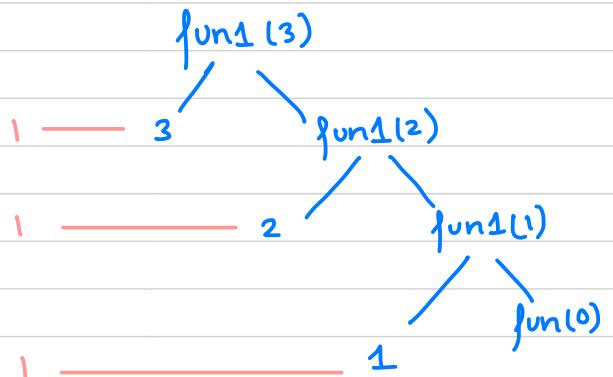
TIME COMPLEXITY (using Tree)

EXAMPLE #1

```
void fun1(int n)
{
    if (n > 0)
    {
        → printf("%d", n);
        fun1(n-1);
    }
}
```

```
void main()
{
    int n = 3;
    fun1(n);
}
```

TRACING TREE



Thus, for n calls $\rightarrow n$ units of time.

$O(n)$

TIME COMPLEXITY (using recurrence relation)

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

Assume it as 1 for constant

$$T(n) = T(n-1) + 1$$

$$T(n) = T(n-1) + 1$$

$$T(n-1) = T(n-2) + 1$$

$$T(n) = \underbrace{T(n-2)}_{T(n-3)+1} + 1 + 1$$

$T(n)$ — This function takes
n time.

$T(n)$ — void fun1(int n)
1 — if ($n > 0$)
1 — printf("%d", n);
 $T(n-1)$ — fun1(n-1);
As it is similar to
 $T(n)$

$$T(n) = T(n-3) + 1 + 2$$

:

$$T(n) = T(n-k) + k$$

$$T(n) = T(n-n) + n$$

$$T(n) = T(0) + n$$

$$T(n) = 1 + n$$

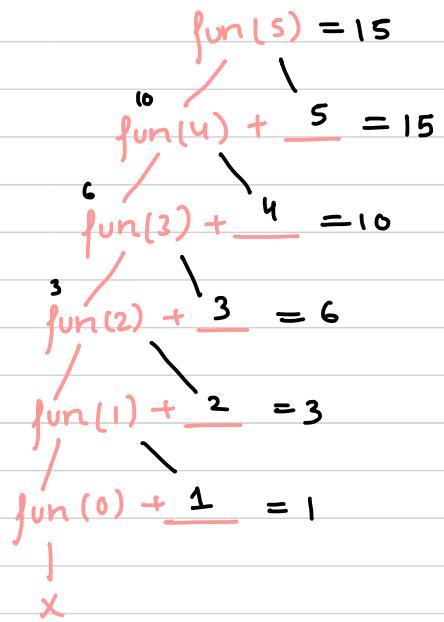
$O(n)$

STATIC VARIABLES IN RECURSION

TRACING TREE

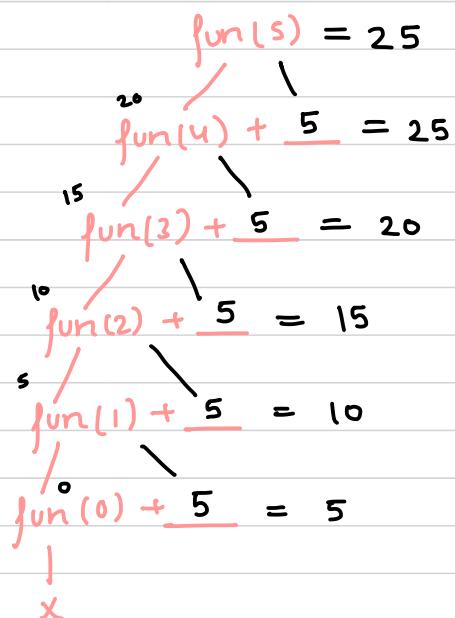
```
int fun (int n)
{
    if (n > 0)
    {
        return fun (n-1) + n;
    }
    return 0;
}
```

```
main()
{
    int a = 5;
    printf ("%d", fun (a));
}
```



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

```
main()
{
    int a = 5;
    printf ("%d", fun (a));
}
```



TYPES OF RECURSION

1. Tail Recursion
2. Head Recursion
3. Tree Recursion
4. Indirect Recursion
5. Nested Recursion

1. TAIL RECURSION

When the function is calling itself and that call is the last call in the function.

```
fun(n)
{
    if (n > 0)
        {
            _____
            _____
            _____
            fun(n-1);
        }
}
```

1. Every operation is performed at calling time.

2. Tail recursions can easily converted into loops.

TAIL RECURSION AND LOOPS

Some compilers convert your program to loop if you have used tail recursion as they are more efficient

```
void fun(int n)
{
    while(n > 0)
    {
        printf("%d", n);
        n--;
    }
}
fun(3);
```

space $O(1)$

```
void fun(int n)
{
    if (n > 0)
    {
        printf("%d", n);
        fun(n-1);
    }
}
fun(3);
```

$O(n)$

2. HEAD RECURSION

It means that the function does not need to perform any operation at the time of calling. It has to do all operations at returning time.

```
void fun (int n)
{
    int i = 1;
    while (i <= n)
    {
        printf ("%d", i);
        i++;
    }
}
fun(3);
```

```
void fun (int n)
{
    if (n > 0)
    {
        fun (n-1);
        printf ("%d", n);
    }
}
fun(3);
```

Head recursions cannot be so easily converted into loops.

3. TREE RECURSION

LINEAR RECURSION

```
fun(n)
{
    if (n > 0)
    {
        _____
        fun(n-1);
        _____
    }
}
```

When the function calls itself only once.

TREE RECURSION

```
fun(n)
{
    if (n > 0)
    {
        _____
        fun(n-1);
        _____
        fun(n-2);
        _____
    }
}
```

When the function calls itself more than one time.

EXAMPLE OF TREE RECURSION

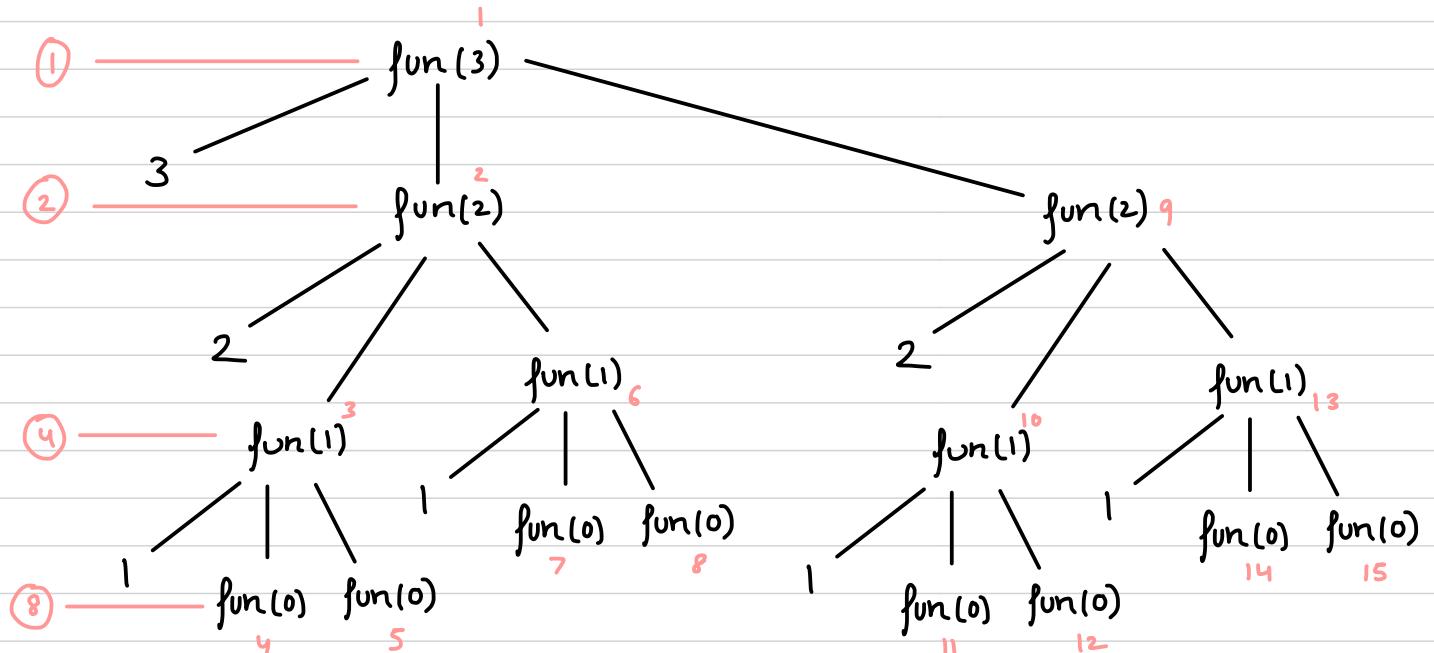
```

void fun (int n)
{
    if (n > 0)
    {
        printf ("%d", n);
        fun (n-1);
        fun (n-1);
    }
}

```

fun (3);

● ACTIVATION RECORDS



$$1 + 2 + 4 + 8 = 15$$

$$2^0 + 2^1 + 2^2 + 2^3 = 2^{3+1} - 1 \quad 2^{n+1} - 1$$

GP Series

$$\text{Time} = O(2^n)$$

$$\text{Space} = O(n)$$

[No of levels of activation records $n+1$]

3 - parameter

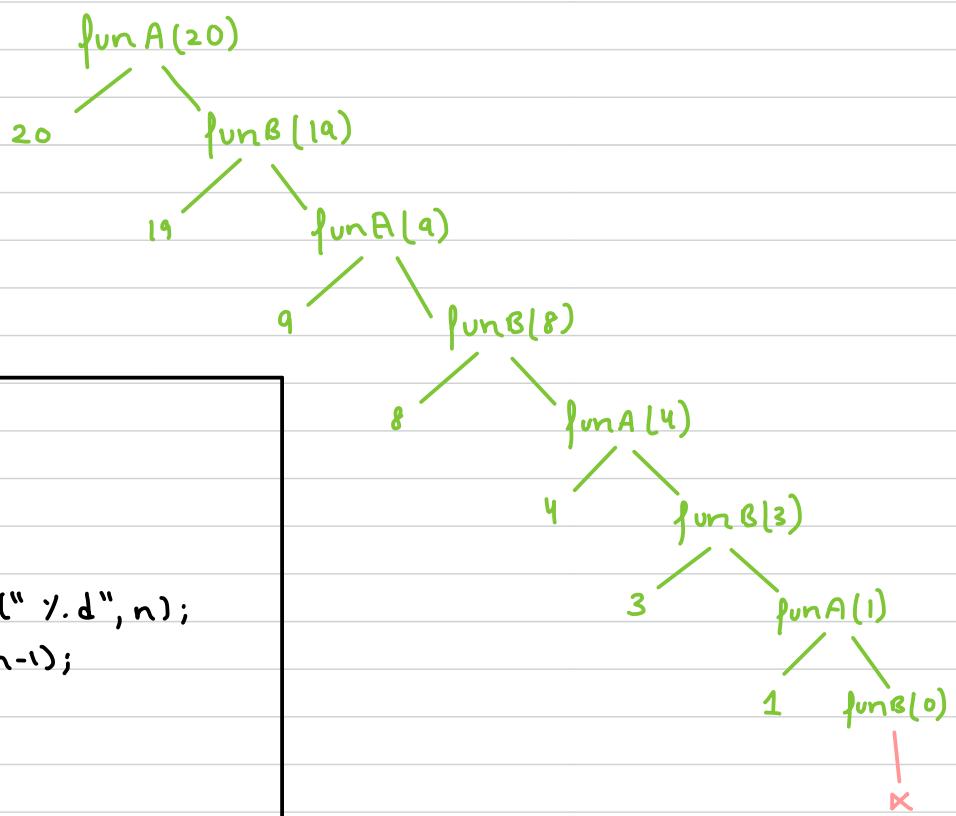
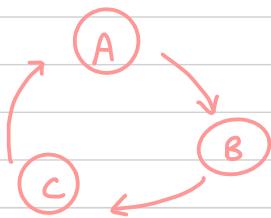
4 - levels

OUTPUT

3 2 1 1 2 1 1

4. INDIRECT RECURSION

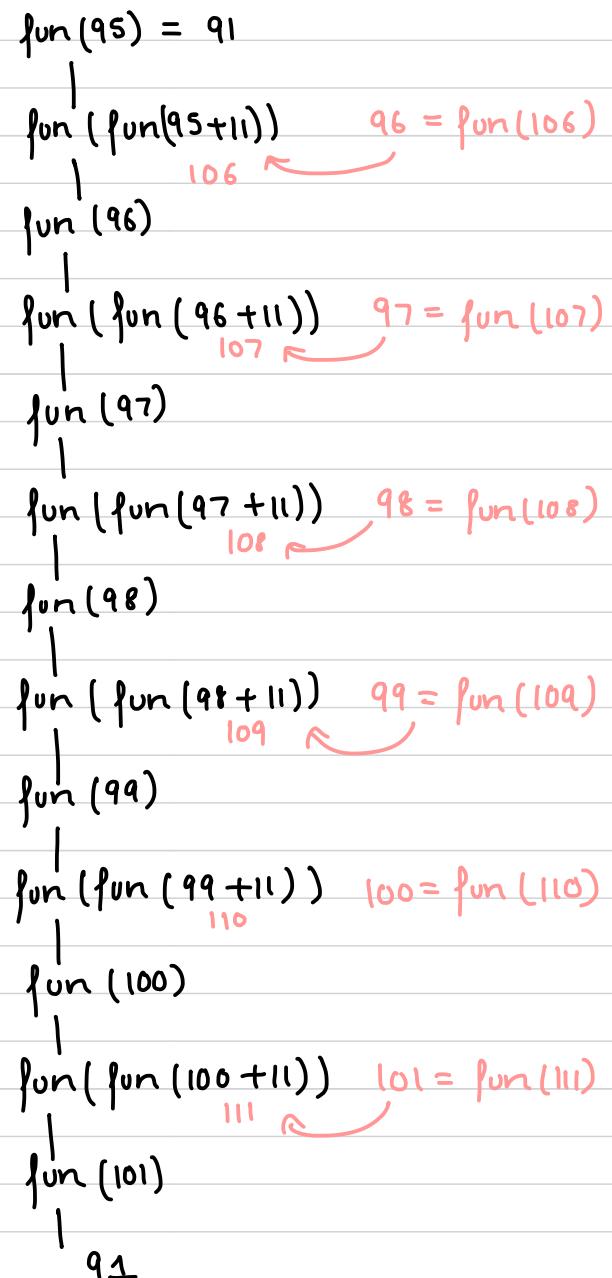
In indirect recursion, there are more than one function and they are calling one another in a circular manner.



5. NESTED RECURSION

In a nested recursion, the recursive function will pass parameter as a recursive call.

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



SUM OF FIRST N NATURAL NUMBERS

$$1+2+3+4+\dots+n$$

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

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

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

```

int sum(int n)
{
    if (n == 0)
        return 0;
    else
        return sum(n-1)+n;
}

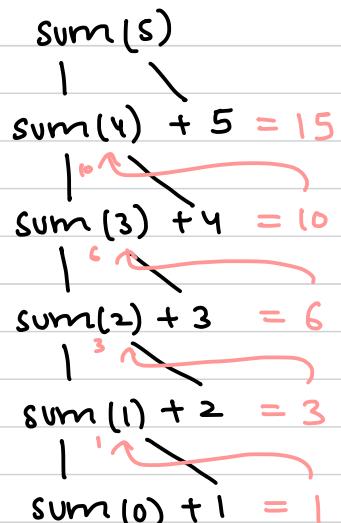
```

Time = O(n)
Space = O(n)

```

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

```



```

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

```

$\frac{1}{2n+3}$
 $O(n)$

FACTORIAL USING RECURSION

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

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

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

```

int fact(int n)
{
    if(n==0)
        return 1;
    else
        return fact(n-1)*n;
}

```

POWER USING RECURSION

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

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

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

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

```

int pow(int m, int n)
{
    if(n==0)
        return 1;
    return pow(m,n-1);
}

```

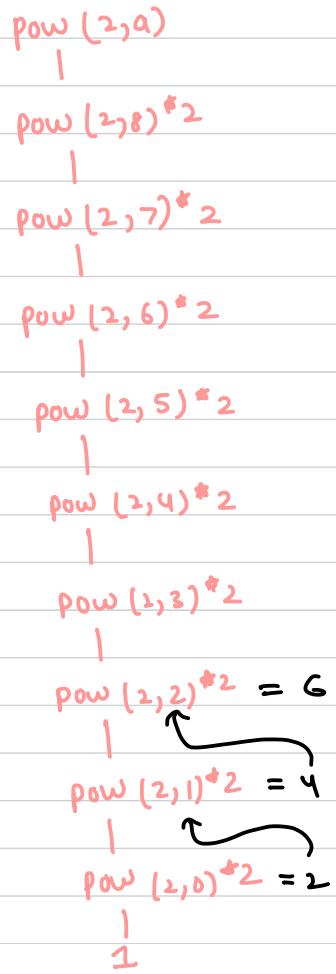
Here 10 calls are being performed

But this way is taking longer

$$2^8 = (2^2)^4$$

$$= (2 * 2)^4$$

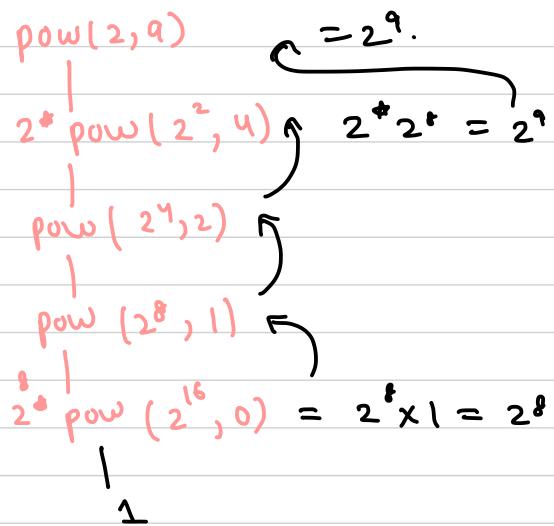
$$2^9 = 2 * (2^2)^4$$



REWRITING POWER FUNCTION

```
int pow(m,n)
{
    if (n == 0)
        return 1;
    else
        if (n % 2 == 0)
            return pow(m*m, n/2);
        else
            return m * pow(m*m, (n-1)/2);
}
```

FASTER METHOD



TAYLOR'S SERIES

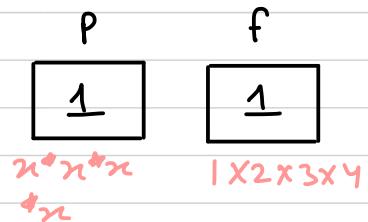
$$e^x = 1 + \frac{x}{1} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \dots \dots \frac{x^n}{n!}$$

$$\begin{aligned}
 e(x, 4) &= 1 + \frac{x}{1} + \frac{x^2}{2} + \frac{x^3}{3!} + \frac{x^4}{4!} \\
 e(x, 3) &= 1 + \frac{x}{1} + \frac{x^2}{2} + \frac{x^3}{3!} \\
 e(x, 2) &= 1 + \frac{2x}{1} + \frac{2x^2}{2} \\
 e(x, 1) &= 1 + \frac{x}{1} \\
 e(x, 0) &= 1
 \end{aligned}$$

||
 1

```

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



TAYLOR'S SERIES USING HORNER'S RULE

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

$\frac{x \times x}{2 \times 1}$ $\frac{2x \times x}{3 \times 2 \times 1}$ No of multiplications
 0 0 1+1 2+2 ↓
 2 4 6 8 10

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

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

$O(n^2)$ Quadratic

$$\begin{aligned}
 & 1 + \frac{x}{1} + \frac{x^2}{1 \times 2} + \frac{x^3}{1 \times 2 \times 3} + \frac{x^4}{1 \times 2 \times 3 \times 4} \\
 & 1 + \frac{x}{1} \left[\frac{x}{2} + \frac{x^2}{2 \times 3} + \frac{x^3}{2 \times 3 \times 4} \right] \\
 & 1 + \frac{x}{1} \left[\frac{x}{2} \left[\frac{x}{3} + \frac{x^2}{3 \times 4} \right] \right] \\
 & 1 + \frac{x}{1} \left[\frac{x}{2} \left[\frac{x}{3} \left[1 + \frac{x}{4} \right] \right] \right]
 \end{aligned}$$

↑ ↑ ↑ ↑
 $O(n)$ Linear

```
int e(int x, int n)
{
```

```
  int s=1;
```

```
  for(; n>0; n--)
    s = 1 + x/n * s;
```

```
  return s;
```

```
}
```

USING FOR LOOP

```
int e(int x, int n)
{
```

```
  static int s=1;
```

```
  if(n==0)
```

```
    return s;
```

```
  else
```

```
    s = 1 + x/n * s;
```

```
  return e(x, n-1);
```

```
}
```

USING RECURSION

FIBONACCI SERIES

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

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

PROGRAM USING ITERATION

```

int fib (int n)
{
    int t0 = 0, t1 = 1, s, i; — 1
    if (n <= 1)
        return n; — 1
    for (i = 2; i <= n; i++) — n
    {
        s = t0 + t1; — n-1
        t0 = t1; — n-1
        t1 = s; — n-1
    }
    return s; — 1
}

```

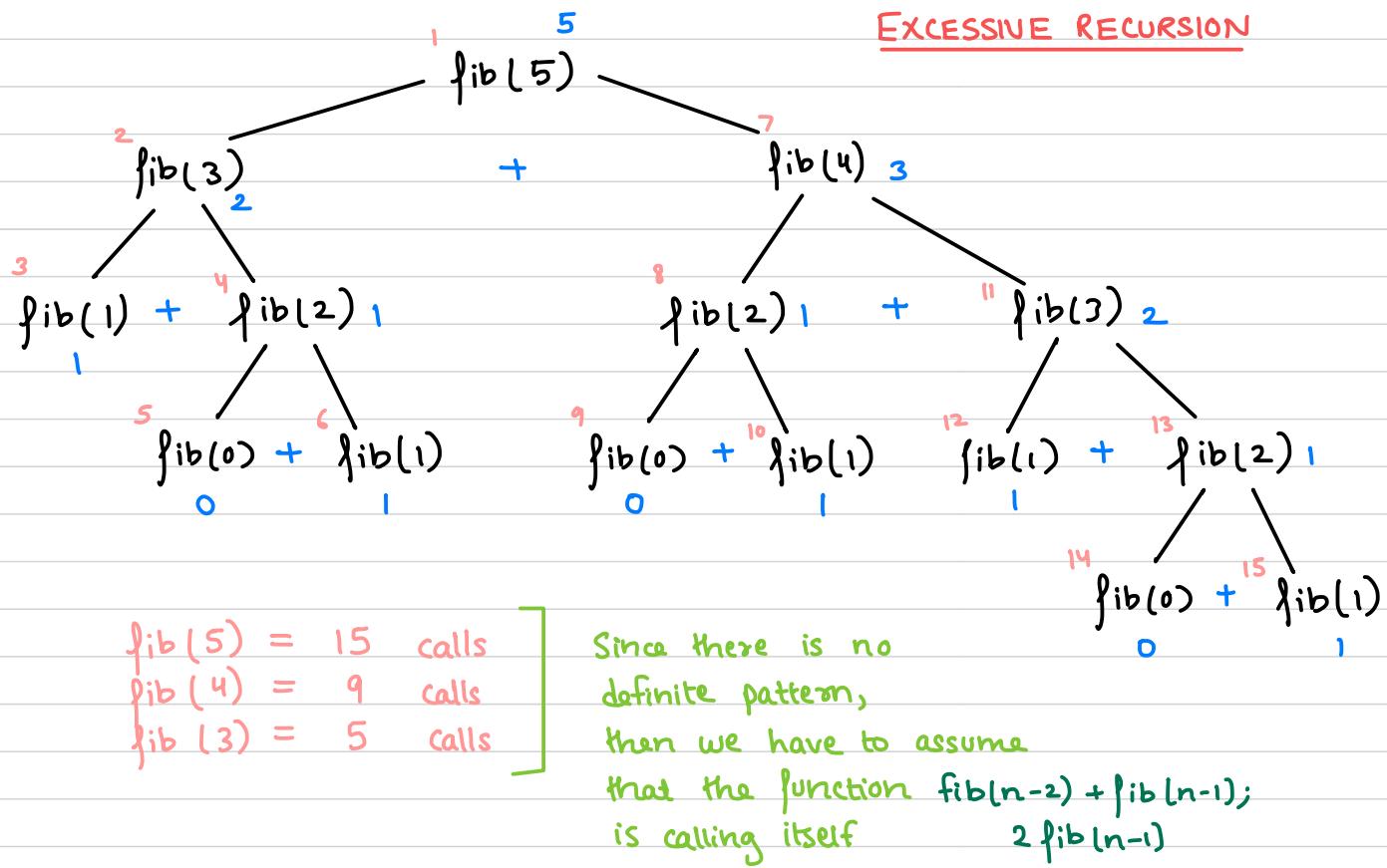
$O(n)$

PROGRAM USING RECURSION

```

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

```

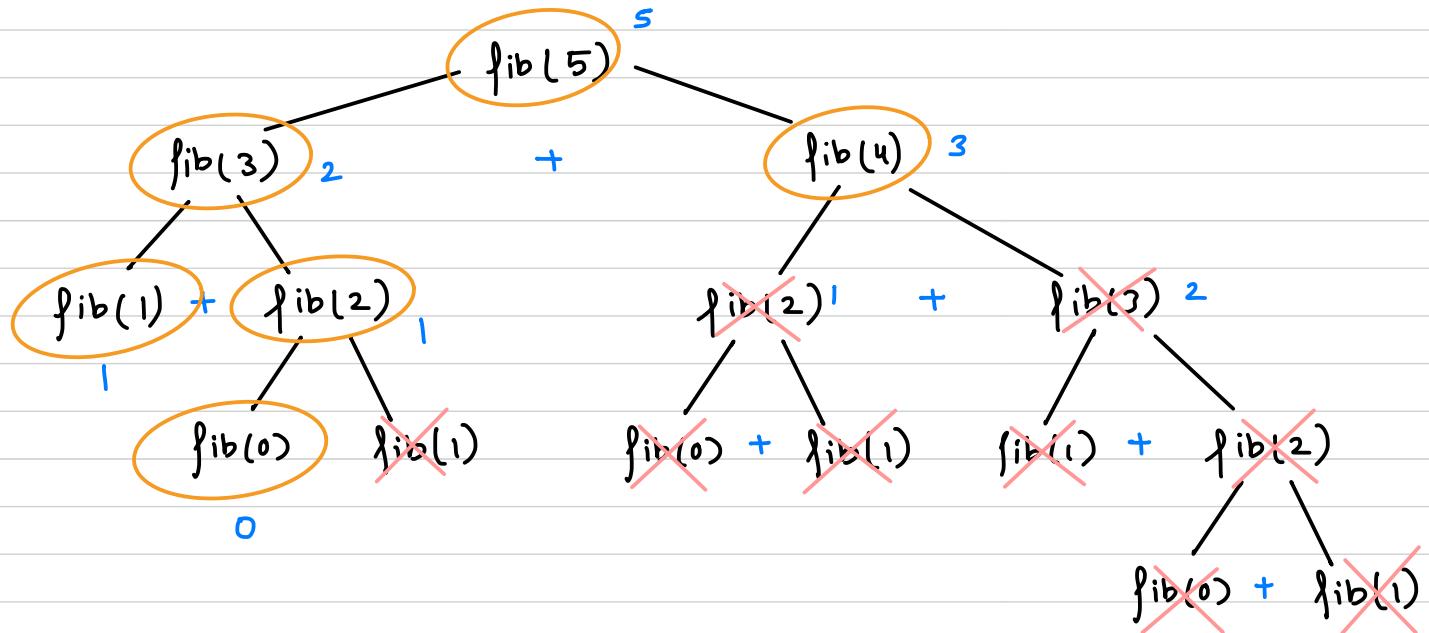


This tree lies in the category of excessive recursion because it calls itself multiple times for the same parameters.

↓
order of (2^n)

To reduce the order of this function, we will write another program using static array and initialize all its values with -1.

F	-1						
	0	1	2	3	4	5	6



So, for 5 as n , 6 calls are made
 \therefore for n , $n+1$ calls are made

$O(n)$

This approach of storing result in an array is called **MEMOIZATION**.

→ Storing the result of function calls, so they can be utilized again for avoiding excessive calls

int F[10];

int fib(int n)

{

if ($n \leq 1$)

{

$F[n] = n;$

return $n;$

}

else

{

if ($F[n-2] == -1$)

$F[n-2] = \text{fib}(n-2);$

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

$F[n-1] = \text{fib}(n-1);$

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

}

COMBINATION FORMULA

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

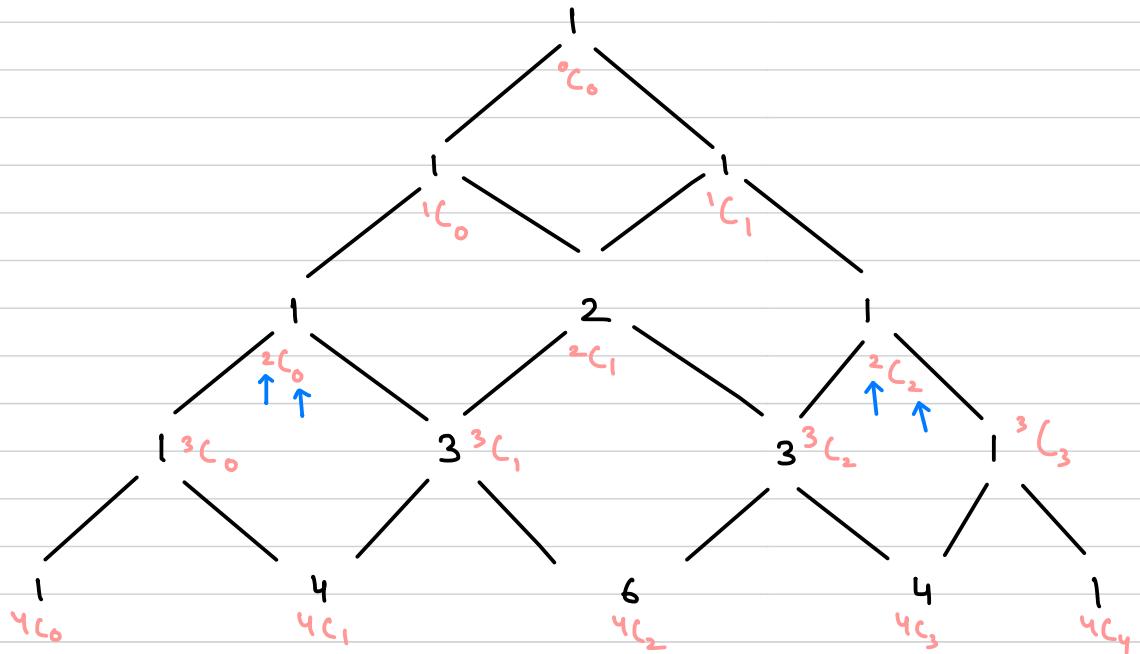
```
int C (int n, int r)
{
```

```
    int t1, t2, t3;
    t1 = fact(n); ----- n
    t2 = fact(r); ----- n
    t3 = fact(n-r); ----- n
```

```
    return t1 / (t2 * t3); ----- 1 / 3n
}
```

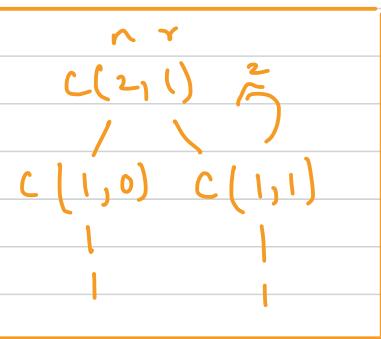
$O(n)$

PASCAL'S TRIANGLE

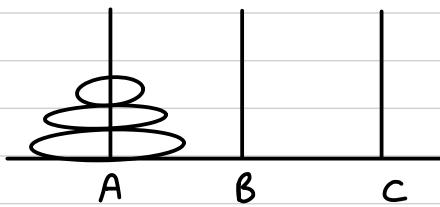


```
int C (int n, int r)
{
```

```
    if (r == 0 || n == r) ●
        return 1;
    else
        return C (n-1, r-1) + C (n-1, r);
}
```



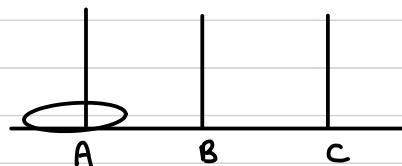
TOWER OF HANOI



1. Move one disk at a time.
2. NO bigger disk can be there above
or smaller one.

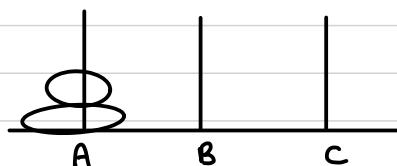
TOH(1, A, B, C)

Move disk A to C using B.



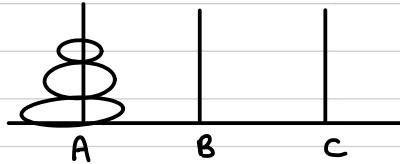
TOH(2, A, B, C)

1. TOH(1, A, C, B)
2. Move disk A to C using B.
3. TOH(1, B, A, C)



TOH(3, A, B, C)

1. TOH(2, A, C, B)
2. Move disk from A to C using B
3. TOH(2, B, A, C).



FOR n number of disk.

TOH($\frac{n}{2}$, A, B, C)

1. TOH($\frac{n-1}{2}$, A, C, B)
2. Move disk from A to C using B
3. TOH($\frac{n-1}{2}$, B, A, C).

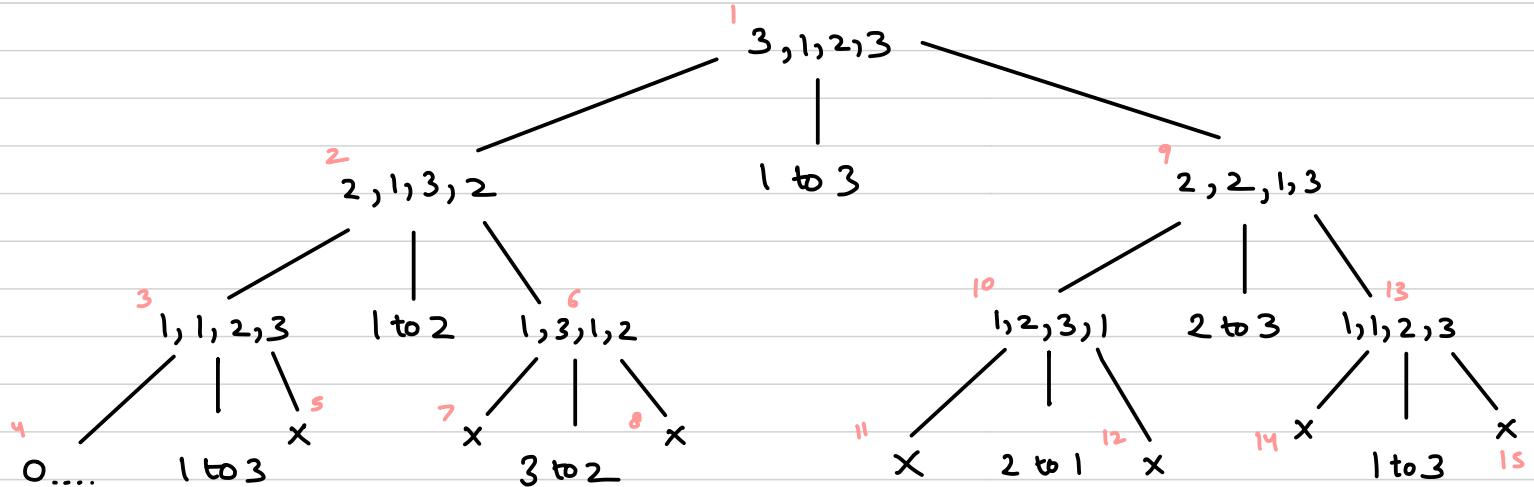
```

void TOH(int n, int A, int B, int C)
{
    if (n > 0)
    {
        TOH(n-1, A, C, B);
        printf("from %d to %d", A, C);
        TOH(n-1, B, A, C);
    }
}

```

TOH(3, 1, 2, 3)

$A \rightarrow 1$
 $B \rightarrow 2$
 $C \rightarrow 3$



OUTPUT

(1 to 3), (1 to 2), (3 to 2), (1, 3), (2, 1), (2, 3), (1, 3)

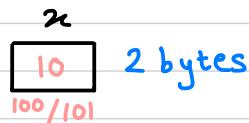
$$\begin{array}{lll}
 n = 3 & 15 & \text{(calls} \\
 & & 1+2+2^2+2^3 = 2^4-1 \\
 n = 2 & 7 & 1+2+2^2 = 2^3-1 \\
 & & \boxed{2^{n+1}-1} \\
 & & O(2^n)
 \end{array}$$



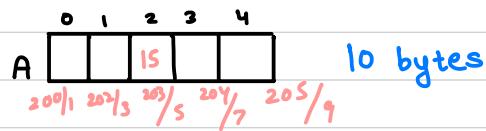
ARRAYS

Scalar →

int $x = 10;$

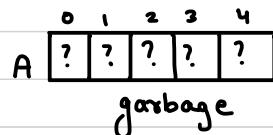


vector →
int $A[5];$
 $A[2] = 15;$

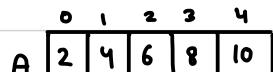


DECLARATION OF ARRAYS

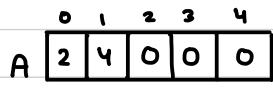
① int $A[5];$



② int $A[5] = \{2, 4, 6, 8, 10\};$

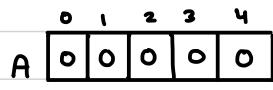


③ int $A[5] = \{2, 4\};$

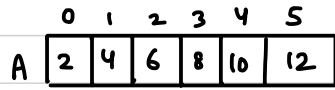


→ Rest of the elements get automatically initialized by 0.

④ int $A[5] = \{0\};$

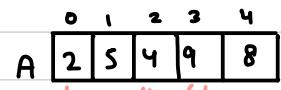


⑤ int $A[] = \{2, 4, 6, 8, 10, 12\};$



→ Depending upon the number of elements, size of the array is automatically allocated.

int $A[5] = \{2, 5, 4, 9, 8\};$



200/1 202/3 203/5 204/7 205/9

→ Array addresses are contagious.

printf(" %d", A[2]);

printf(" %d", *A);

printf(" %d", *A+2);

4

for(i=0; i<5; i++)

printf(" %d", &A[i]);

↓

↓

for printing address.

STATIC VS DYNAMIC ARRAY

Size of the array
is static

→ Size of the array is dynamic

→ Once an array is created, its size cannot be modified.

In c

→ size of the array is decided at compile time

In C++

→ The size of the array can be decided at run time also.

ACCESSING HEAP

```
int n;  
cin >> n;  
int A[n];
```

```
void main()
{
```

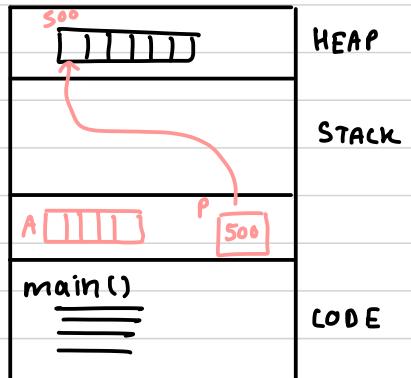
int A[5];
int * p;

C++ p = new int[5];

$$\rho[0] = 5$$

C p = (int *)malloc(5 * sizeof(int));
; **<stdlib.h>**

MEMORY LEAK
shortage of memory.

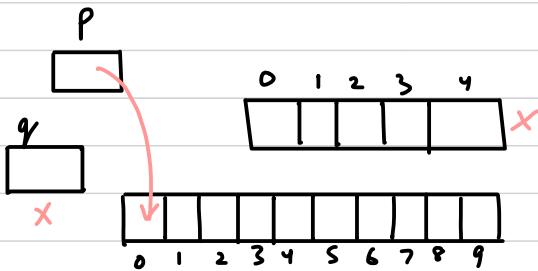


ONE WAY OF INCREASING SIZE OF ARRAY

```
int *p = new int [5];  
int *q = new int [10];
```

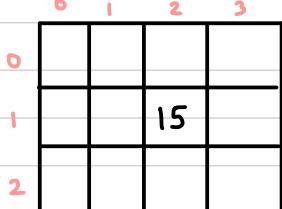
$\text{for } (i=0; i < 5; i++)$
 $\quad q[i] = p[i]$

```
delete []p;  
p = q;  
q = NULL;
```



2D - ARRAY

① $\text{int } A[3][4] = \{ \{ 1, 2, 3, 4 \}, \{ 2, 4, 6, 8 \}, \{ 3, 5, 7, 9 \} \};$



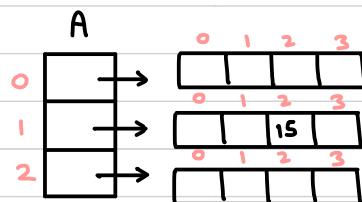
↓
Array will be stored
inside stack.

$$A[1][2] = 15$$

→ Array of pointers

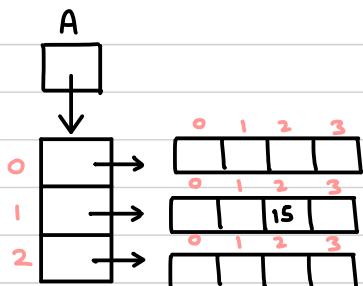
② $\text{int } * A[3];$

$A[0] = \text{new int}[4];$ # Memory is created inside
 $A[1] = \text{new int}[4];$ created inside
 $A[2] = \text{new int}[4];$ heap.



$A[1][2] = 15;$ # We can access array
of pointers in the same
way.

③



$\text{int } ** A;$
 $A = \text{new int } * [3];$
 $A[0] = \text{new int } [4];$
 $A[1] = \text{new int } [4];$
 $A[2] = \text{new int } [4];$

All the memory is
allocated in heap.

for
column

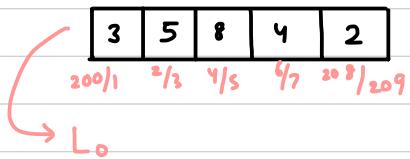
$\text{for } (i=0; i<3; i++)$
 {
 for
row
 []
 {
 for
row
 []
 {
 A[i][j] = ___;
 }
 }
 }

● Elements accessed in order

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

HOW COMPILER GENERATES FORMULA FOR ADDRESS OF AN ARRAY

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



$$\text{Add}(A[3]) = 200 + 3^* 2 = 206$$

$$\text{Add}(A[3]) = L_0 + 3^* 2$$

$$\text{Add}(A[i]) = L_0 + i^* w \quad \xrightarrow{\text{FORMULA}}$$

Base
Address Index
Size of datatype.

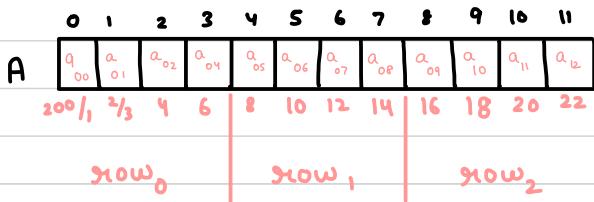
No of operations = 2

$$\text{Add}(A[i]) = L_0 + (i-1)^* w \quad \xrightarrow{\text{Formula, when indices are starting from one onwards}}$$

No of operations = 3
More time consuming.

That's why C and C++ do not start array index with 1. Only for one extra operation, time taken by the program will increase and this will make the program slower.

ROW MAJOR



int A[3][4];
m n

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

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

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

4 operations

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

6 operations

COLUMN MAJOR

	0	1	2	3	4	5	6	7	8	9	10	11
A	a ₀₀	a ₁₀	a ₂₀	a ₃₀	a ₄₀	a ₅₀	a ₆₀	a ₇₀	a ₈₀	a ₉₀	a ₁₀₀	a ₁₁₀
		col 0		col 1		col 2		col 3				

$$ADD(A[1][2]) = 200 + [2^* 3 + 1]^* 2 = 214$$

$$ADD(A[1][3]) = 200 + [3^* 3 + 1]^* 2 = 220$$

$$ADD(A[i][j]) = L_0 + (j^* m + i)^* \omega$$

FORMULAS FOR nD ARRAYS

$$A[d_1][d_2][d_3][d_4], \quad 4D \text{ ARRAY}$$

Row Major

$$ADD(A[i_1][i_2][i_3][i_4]) = L_0 + [i_1^* d_2^* d_3^* d_4 + i_2^* d_3^* d_4 + i_3^* d_4 + i_4]^* \omega$$

Column Major

$$ADD(A[i_1][i_2][i_3][i_4]) = L_0 + [i_4^* d_3^* d_2^* d_1 + i_3^* d_2^* d_1 + i_2^* d_1 + i_1]^* \omega$$

ROW MAJOR FOR nD

$$L_0 + \sum_{p=1}^n \left[i_p^* \prod_{q=p+1}^n d_q \right]^* \omega$$

COLUMN MAJOR for nD

$$L_0 + \sum_{p=n}^1 \left[i_p^* \prod_{q=p-1}^1 d_q \right]^* \omega$$

(self tried)
(please check)

$A[d_1][d_2][d_3][d_4]$;

4D ARRAY

HORNER'S RULE

Row Major

$$\text{Add}(A[i_1][i_2][i_3][i_4]) = L_0 + \left[\underbrace{i_1 * d_2 * d_3 * d_4}_3 + \underbrace{i_2 * d_3 * d_4}_2 + \underbrace{i_3 * d_4}_1 + i_4 \right]^* \omega$$

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

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

$$nD \rightarrow n-1 + n-2 + n-3 \dots + 1 = \frac{n(n-1)}{2}$$

$O(n^2)$

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

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

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

$O(n)$

FORMULA FOR 3D ARRAYS

int $A[l][m][n]$;

Row MAJOR

$$\text{Add}(A[i][j][k]) = L_0 + [i * m * n + j * n + k]^* \omega$$

COLUMN MAJOR

$$\text{Add}(A[i][j][k]) = L_0 + [k * m * l + j * l + i]^* \omega$$



ARRAY ADT

↳ Abstract Datatype
↓

1. Representation of data
2. Operations on data

DATA

1. Array Space
2. Size
3. Length (No of elements)

OPERATIONS

1. Display()
2. Add(x) / Append(x)
3. Insert(index, x)
4. Delete(index)
5. Search(x)
6. Get(index)
7. Set(index, x)
8. Max(), Min()
9. Reverse()
10. Shift() / Rotate()

① int A[10];
 ② int *A;
 $A = \text{new int}[size];$

1. DISPLAY

pseudo code

```
for(i=0; i<length; i++)
{
    print(A[i])
}
```

Array size = 10

Length = 6

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

2. Add(x) / Append(x)

$$\begin{array}{l}
 A[\text{Length}] = x; \quad | \\
 \text{Length}++; \quad |
 \end{array}$$

$$\frac{f(n) = 2}{f(n) = 2^n}$$

Array size = 10

Length = 7

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

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

3. Insert(4, 15)

`for (i = length; i > index; i--)`

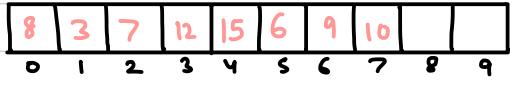
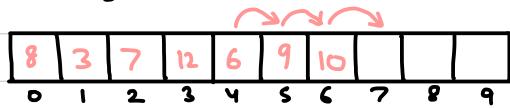
`A[i] = A[i-1]; — o-n`

Inserting at \downarrow index 8 Inserting at \downarrow 0

`A[index] = x;` ————— |
Length++;

Array Size = 10

Length = 8



$O(1)$ $O(n)$
min max

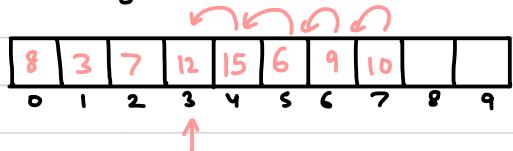
4. Delete(3)

`x = A[index];` ————— |

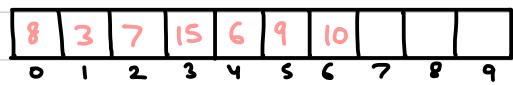
`for (i = index; i <= Length-1; i++)`
`A[i] = A[i+1];` ————— o-n

Array Size = 10

Length = 7



`Length--;` ————— |
min=2 max=n+2



Best $O(1)$ Worst $O(n)$

Index should be in range of length

we cannot leave empty space between two elements in an array.

LINEAR SEARCH → Searching each element and incrementing

Array Size = 10
Length = 10



Key = 5 ← Successful

Key = 12 ← Unsuccessful

```
for (i=0; i < length; i++)
    if (key == A[i])
        return i;
    return -1;
```

Average

Best — $O(1)$
Worst — $O(n)$
Average — $O(n)$

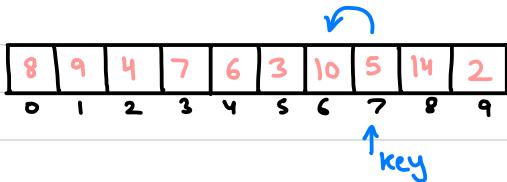
$$\frac{1+2+3+\dots+n}{n} = \frac{(n+1)n}{2}$$

$$= \frac{n+1}{2}$$

↑ found at location 1
↑ found at 2

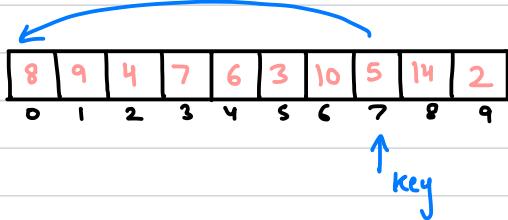
FASTER WAY OF LINEAR SEARCH

1. Transposition: Moving the searched element one step back



```
for (i=0; i<length; i++)  
{  
    if (key == A[i])  
    {  
        swap(A[i], A[i-1]);  
        return i-1;  
    }  
}
```

2. Move to front / head: The searched element is brought to first position in array.



```
for (i=0; i<length; i++)  
{  
    if (key == A[i])  
    {  
        swap(A[i], A[0]);  
        return 0;  
    }  
}
```

BINARY SEARCH

Size = 15

length = 15

A

4	8	10	15	18	21	24	27	29	33	34	37	41	43
1	2	3	4	5	6	7	8	9	10	11	12	13	14

Sorted Array

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;
    }
    return -1;      # key not found
}

```

ITERATIVE VERSION

Binary search is faster than linear search

Algorithm RBinSearch(l, h, key)

```

{
    if (l <= h)
    {
        mid = [(l+h)/2];
        if (key == A[mid])
            return mid;
        else if (key < A[mid])
            return RBinsearch(l, mid-1, key);
        else
            return RBinsearch(mid+1, h, key);
    }
    return -1;      # key not found
}

```

RECURSIVE VERSION

Tail Recursion

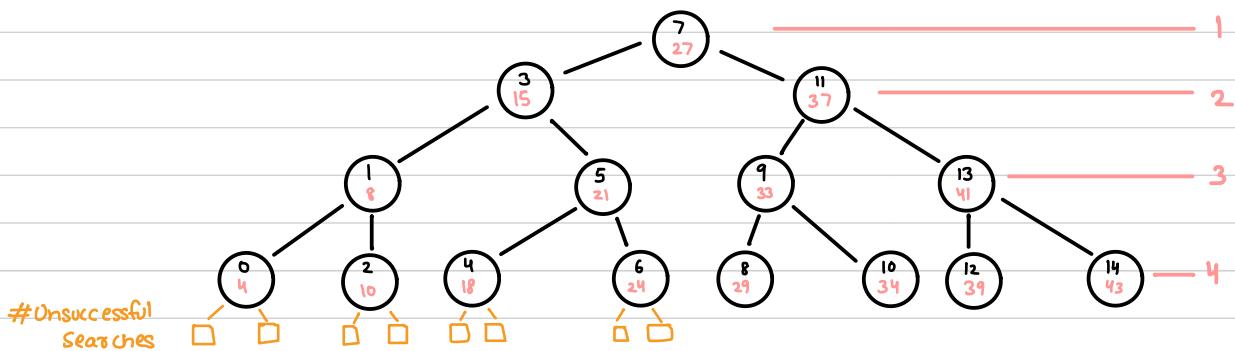
ANALYSIS OF BINARY SEARCH

Size = 15

length = 15

A

4	8	10	15	18	21	24	27	29	33	34	37	41	43
1	2	3	4	5	6	7	8	9	10	11	12	13	14



Best min - $O(1)$

Worst max - $O(\log n)$

Unsuccessful max - $O(\log n)$

Why $\log n$?

\log

let size of array be 16

$$\frac{16}{2} \\ \frac{8}{2} \\ \frac{4}{2} \\ \frac{2}{2}$$

$$2^4 = 16 \\ 4 = \log_2 16$$

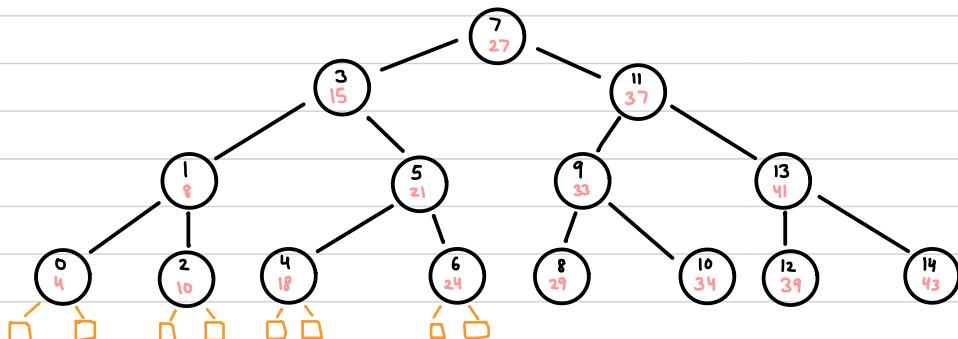
Inverse of power is log.

$\log n$

16 is divided by 2 multiple times

AVERAGE CASE ANALYSIS OF BINARY SEARCH

= Total time taken in all possible cases
Number of cases



$$1 + 1 \times 2 + 2 \times 4 + 3 \times 8$$

No of \leftarrow No of elements

Comparisons

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

$\log \rightarrow$ Level of tree (Here 4)

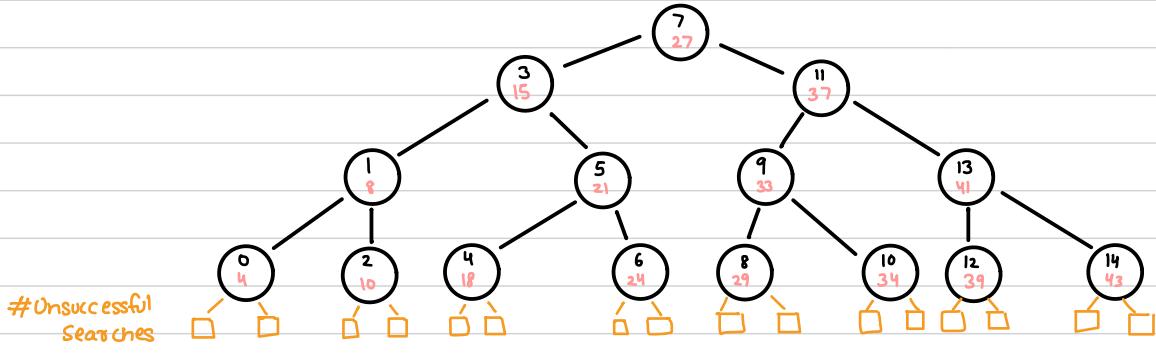
$$\sum_{i=1}^{\log n} i \times 2^i = \frac{\log n \times 2^{\log n}}{n}$$

i is replaced with $\log n$

Same as formula

$$= \frac{\log n \times n^{\log 2}}{n}$$

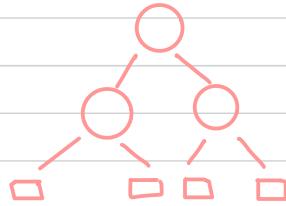
$$= \log n$$



$$I = \text{sum of all internal nodes}$$

$$E = \text{sum of all external nodes}$$

$$E = I + 2n$$



$$n = 3$$

$$I = 2$$

$$E = 2 \times 4 = 8$$

$$E = I + 2n$$

$$i = \text{num of internal nodes}$$

$$e = \text{num of external nodes}$$

$$n = \text{num of nodes}$$

$$e = i + 1$$

Average successful time for n elements

$$AS(n) = 1 + \frac{I}{n} \Rightarrow 1 + \frac{E - 2n}{n} \Rightarrow 1 + \frac{n \log n}{n} - 2 \\ \Rightarrow 1 + \frac{E}{n} - 2 \Rightarrow \log n$$

Average unsuccessful time

$$AV(n) = \frac{E}{n+1} = \frac{n \log n}{n+1} = \log n \quad E = n \log n \\ E = I + 2n \quad I = E - 2n$$

6. Get(index):

```
if (index >= 0 && index < length)          O(1)
    return A[index];
```

7. Set(index, x)

```
if (index >= 0 && index < length)          O(1)
    A[index] = x;
```

8. Max()

```

max = A[0];           —————— 1
for (i=1; i < length; i++) ————— n
{
    if (A[i] > max) ————— n-1
        max = A[i];
}
return max; —————— 1
                                —————— 2n+1 O(n)

```

9. Min()

```

min = A[0];
for (i=1; i < length; i++)
{
    if (A[i] < min )
        min = A[i];
}
return min;

```

10. Sum()

```

Total = 0;           —————— 1
for (i=0; i < length; i++) ————— n+1
    Total += A[i] ————— n
return total —————— 1
                                —————— 2n+3
                                O(n)

```

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

11. Avg()

```

Total = 0;
for (i=0; i < length; i++)
    Total += A[i]
return total/n;

```

RECURSIVE (ALL)

```

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

```

sum(A, length-1) — call

REVERSE AND SHIFT AN ARRAY

1. Reverse
2. Left Shift
3. Left Rotate
4. Right Shift
5. Right Rotate

1. Reversing an array

Reversing using auxiliary array B.

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

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

$A \rightarrow B \quad n$

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

$B \rightarrow A \quad \frac{n}{2n}$
 $O(n)$

for ($i = length - 1; j = 0; i >= 0; i--, j++$)
 $B[j] = A[i];$] Reverse copying array

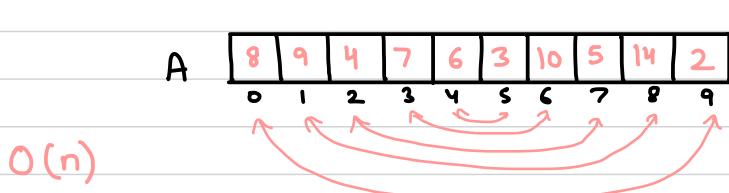
for ($i = 0; i < length; i++$)
 $A[i] = B[i];$

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

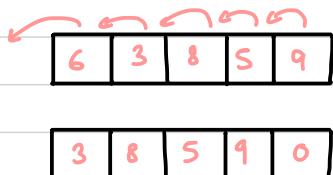
 temp = $A[i];$
 $A[i] = A[j];$
 $A[j] = temp;$

}

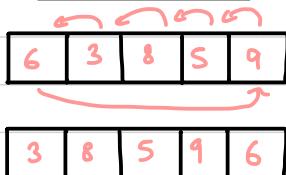
ANOTHER METHOD OF REVERSING ARRAY



2. Left Shift



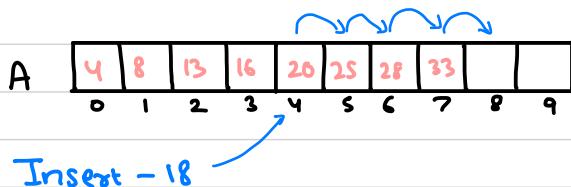
3. Left Rotate



CHECK IF ARRAY IS SORTED

1. Inserting in an sorted array
2. Checking if array is sorted
3. Arranging -ve elements on left side and +ve on right side.

1. Inserting in an sorted array



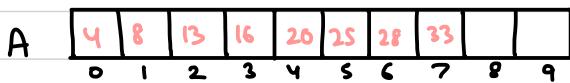
```

x = 18;
i = length - 1;
while (A[i] > x)
{
    A[i+1] = A[i];
    i--;
}
A[i+1] = x;

```

Starting from last element, loop will run until the element is greater than the element to be inserted

2. Checking if array is sorted



```

Algorithm isSorted (A, n)
{
    for (i=0, i<n-1, i++)
        if (A[i] > A[i+1])
            return false;
    }
    return true;
}

```

We will check false condition first by checking if any element is greater than its next element.

$O(n)$ - Max Worst
 $O(1)$ - Min Best

3. Arranging -ve elements on left side and +ve on right side.

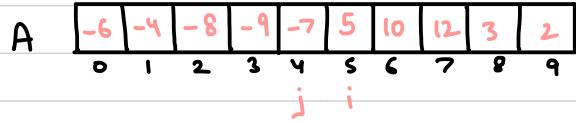
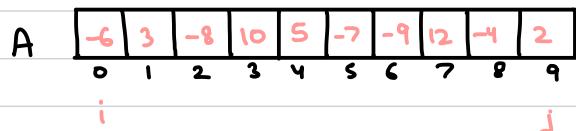
```
i=0;
j = length-1;
```

```
while ( i < j )
{
```

```
    while ( A[i] < 0 )
        i++;
    while ( A[j] ≥ 0 )
        j--;

```

```
}
```



$O(n)$

$n+2$ comparisons are made

```
if ( i < j )
    swap( A[i], A[j] )
```

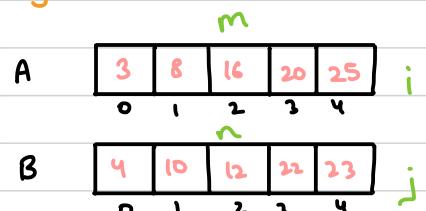
MERGING ARRAYS

```
i=0;
j=0;
k=0;
```

```
while ( i < m && j < n ) → # When either of i or j have reached end of array.
{
```

```
    if ( A[i] < B[j] )
    {
        C[k] = A[i];
        k++;
        i++;
    }
```

Can also be written as
 $C[k++] = A[i++];$



```
else
{
```

```
    C[k] = B[j];
    k++;
    j++;
}
```

$C[k++] = B[j++]$

$\Theta = (m+n)$

↳ Time is known



```
for ( ; i < m; i++ )
    C[k++] = A[i];
for ( ; j < n; j++ )
    C[k++] = B[j];
```

When some elements in either of array are remaining to be copied.

```
}
```

SET OPERATIONS

1. UNION

A	3 5 10 4 6	m
	o 1 2 3 4	

B	12 4 7 2 5	n
	o 1 2 3 4	

C	3 5 10 4 6 12 7 2	
	o 1 2 3 4 5 6 7 8 9	

A	3 4 5 6 10	m
	o 1 2 3 4	

B	2 4 5 7 12	n
	o 1 2 3 4	

C	2 3 4 5 6 7 10 12	
	o 1 2 3 4 5 6 7 8 9	

- Copy all elements of A in C
- Then copy elements of B which are not there in C.

$$\begin{aligned} & m + m \cdot n \\ & n + n \cdot n \\ & n + n^2 \\ & O(n^2) \end{aligned}$$

- Use merge procedure
- Copy the smaller element to C
- If same element, copy once
- Use i, j, k method

$$\Theta(m+n)$$

$$\Theta(n+n)$$

$$\Theta(n)$$

2. INTERSECTION

A	3 5 10 4 6	m
	o 1 2 3 4	

B	12 4 7 2 5	n
	o 1 2 3 4	

C	5 4	
	o 1 2 3 4 5 6 7 8 9	

A	3 4 5 6 10	m
	o 1 2 3 4	

B	2 4 5 7 12	n
	o 1 2 3 4	

C	4 5	
	o 1 2 3 4 5 6 7 8 9	

Copy the common elements of A and B to C

- One by one, take each element of A
- If it is present in B, copy it to C otherwise move on to next element

$$\begin{aligned} & n+m \\ & n \cdot n \\ & O(n^2) \end{aligned}$$

- Use merge method
- If element is smaller, don't copy.
- If element is same, copy
- Not merging but similar to it.

$$\Theta(m+n)$$

$$\Theta(n)$$

3. DIFFERENCE

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			
	<table border="1"> <tr> <td>0</td><td>1</td><td>2</td><td>3</td><td>4</td> </tr> </table>	0	1	2	3	4	
0	1	2	3	4			

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			
	<table border="1"> <tr> <td>0</td><td>1</td><td>2</td><td>3</td><td>4</td> </tr> </table>	0	1	2	3	4	
0	1	2	3	4			

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

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	i
3	4	5	6	10				
	<table border="1"> <tr> <td>0</td><td>1</td><td>2</td><td>3</td><td>4</td> </tr> </table>	0	1	2	3	4		
0	1	2	3	4				

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	j
2	4	5	7	12				
	<table border="1"> <tr> <td>0</td><td>1</td><td>2</td><td>3</td><td>4</td> </tr> </table>	0	1	2	3	4		
0	1	2	3	4				

A-B

- we want elements of A which are not there in B.
- One by one, take each element of A
 - If it is not present in B, copy it to C, otherwise move to next element

$\Theta(m+n)$

$\Theta(n^2)$

$\Theta(n^2)$

- Use merge procedure.
- Compare A[i] and B[i], if A[i] is small, copy it to C otherwise increment j.
- If same, don't copy.

$\Theta(m+n)$

$\Theta(n)$

FIND MISSING ELEMENT

1. Single missing element in an sorted array.
2. Multiple missing element in an sorted array.
3. Missing element in unsorted array.

1. Single missing element in an sorted array.

METHOD 1

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

ITERATIVE METHOD

for (i=0; i<11; i++)

 sum += A[i];

 s = n * (n+1) / 2; # Formula for sum of

 n natural numbers

 s - sum

 78 - 71 = 7

 Missing num

METHOD 2

A

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

↓ ↓ ↓
 6-0 7-1 8-2
 " " "
 6 6 6

$$\begin{aligned} l &= 6 && \text{To be known} \\ h &= 17 \\ n &= 11 \end{aligned}$$

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

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

```
    if (A[i] - i != diff)  
{
```

```
        printf(" missing element : %d ", i+diff);  
        break;
```

```
}
```

```
}
```

$O(n)$

2. Multiple missing element in an sorted array.

METHOD 1

A

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
 6 6 6 6 8 9 9

$$\text{diff} = 6 - 0$$

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

```
    if (A[i] - i != diff)  
{
```

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

```
{
```

```
            printf("%d ", i + diff);  
            diff++;
```

```
}
```

```
}
```

```
}
```

negligible (few elements)

$O(n)$

METHOD 2

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

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

Hash Table / Bit Set

A table is created of size equal to the largest element of array A. The array is initialized with 0. One by one, each element present in array A, the index of hash array is initialized by 1 corresponding to it.

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

for ($i=0; i < n; i++$)
 if (H[i] == 0)
 printf("y.d", i);

n
2n
O(n)

FINDING DUPLICATE IN A SORTED ARRAY

lastDuplicate = 0;

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

for($i=0; i < n; i++$)
{
 if (A[i] == A[i+1] && A[i] != lastDuplicate) # So that if there are
 {
 printf("y.d\n", A[i]);
 lastDuplicate = A[i];
 }
}

COUNTING NO OF TIMES OF DUPLICATE ELEMENT

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

```
    if (A[i] == A[i+1])
    {
```

```
        j = i+1;
```

```
        while (A[i] == A[j])
            j++;
```

```
        printf "%d %d is appearing %d times, A[%d], %d";
        i = j-1;
```

$O(n)$

```
}
```

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

FINDING DUPLICATES IN SORTED ARRAY USING HASHING

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

0	0	0	/	0	0	/	0	/	2	0	/	1	0	/	1	0	0	/	3	0	0	0	0	/	1	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20						

```
for (i=0; i<n; i++)
    H[A[i]]++;
```

$n+n = 2n$

$O(n)$

```
for (i=0; i<=max; i++)
    if (H[i]>1)
        printf ("%d %d", i, H[i]);
```

FINDING DUPLICATES IN AN UNSORTED ARRAY

METHOD 1

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

```

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

    if (A[i] != -1)
    {
        for (j = i+1; j < n; j++)
        {
            if (A[i] == A[j])
            {
                count++;
                A[j] = -1;
            }
        }

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

```

$O(n^2)$

METHOD 2

USING HASH TABLE

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

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

$$\begin{array}{r}
 n \\
 + \\
 n \\
 \hline
 O(n)
 \end{array}$$

FIND PAIR WITH SUM K ($a+b=k$)

SORTED

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

$i = 0, j = n-1$
 while ($i < j$)
 {

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

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

 i++;

 j--;

}

 else if ($A[i] + A[j] < k$)

 i++;

 else

 O(n)

 j--;

}

Let $a+b=10$ (to be searched)

$i+j$

if $i+j > 10$

decrement j

$i+j < 10$

increment i

$i+j = 10$

increment i

decrement j

FIND PAIR WITH SUM K ($a+b=k$)

UNSORTED

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

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

{

 for ($j=i+1; j < n; j++$)

{

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

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

}

}

O(n^2)

FIND PAIR WITH SUM K (a+b=k)

USING HASHING

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

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

```
        if (H[k - A[i]] != 0)
```

```
            printf("y.d + y.d = y.d", A[i], k - A[i], k);
```

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

FIND MAX AND MIN IN SINGLE SCAN

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

```
min = A[0];
```

```
max = A[0];
```

```
n=10
```

```
O(n)
```

Best =  10, 9, 8, 7, 2, 1

comp = n-1

Worst =  1, 2, 3, 5, 8, 9

comp = 2(n-1)

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

```
        if (A[i] < min)
```

```
            min = A[i];
```

```
        else if (A[i] > max)
```

```
            max = A[i];
```

}



STRINGS

1. Character Sets / ASCII codes
2. Character Array
3. String
4. Creating a string

→ For English Language

1. Character Sets / ASCII codes

American Standard Code for Information Interchange

A - 65	a - 97	0 - 48
B - 66	b - 98	1 - 49
:	:	:
Z - 90	z - 122	9 - 57

UNICODES

2 byte

16 bits Represented in form of
4x4 bits hexadecimal

↓ enter - 10
Space - 13
esc - 27

0 - 127	Total 128	1 byte
	$2^7 = 128$	
	7 bits	

2. Character Array

```
char temp;
temp = 'A';
```

A

65 (Stored as 65)

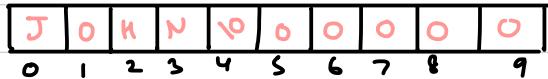
```
printf("Y.C", temp);
```

Displays 'A' and not 65

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

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

char name[10] = { 'J', 'o', 'h', 'N', '\0' };



String delimiter
End of string char
NULL char

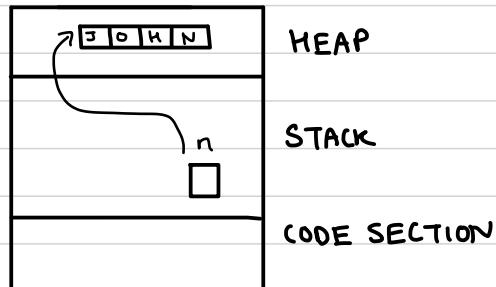
CREATING A STRING

(1) char name[10] = { 'J', 'o', 'h', 'N', '\0' };

(2) char name[] = { 'J', 'o', 'h', 'N', '\0' }; Size of array = 5

(3) char name[] = "John"; compiler takes null character on its own

(4) char *n = "John";



DISPLAYING A STRING

char name[10] = "David";

printf("y.s", name);

scanf("y.s", name); → cannot read after space

gets(name) → can read until you hit enter

FINDING LENGTH OF STRING

```
int main()
{
```

char *s = "welcome"; // No size is required

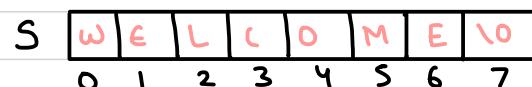
int i;

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

// This will remain empty

}

} printf("Length is %d", i);



i=7

CHANGING CASE OF A STRING

```

int main()
{
    char A[] = " WELCOME";
    int i;
    for(i=0; A[i] != '\0'; i++)
        A[i] = A[i] + 32;
    printf (" Y.S", A);
}

```

TOGGLE CASES

S	w	E	L	C	O	M	E	\0
0	1	2	3	4	5	6	7	

```

int main()
{
    char A[] = " wElLCome";
    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;
}

```

COUNTING NO OF VOWELS AND CONSONANTS

```

for(i=0; A[i] != '\0'; i++)
{
    if ( A[i] == 'a' || A[i] == 'e'.... )
        vcount++;
    else if ((A[i] >= 65 && A[i] <= 90) || (A[i] >= 97 && A[i] <= 122))
        ccount++;
}

```

COUNTING NO OF WORDS

```
int main()
{
    char A[] = "How are you";
    int i, word = 1;
    for (i=0; A[i] != '\0'; i++)
        if (A[i] == ' ' || A[i-1] == ' ')
            word++;
}
```

white space ↗ when there are more than one space

VALIDATING A STRING

```
int valid (char * name)
{
    int i;
    for (i=0; name[i] != '\0'; i++)
        if (! (name[i] >= 65 && name[i] <= 90) &&
            !(name[i] >= 97 && name[i] <= 122) &&
            !(name[i] >= 48 && name[i] <= 57))
            return 0;
    else
        return 1;
}
```

```
int main()
{
    char * name = "Anil321";
    if (validate(name))
        printf ("Valid String");
    else
        printf ("Invalid String");
}
```

This type of string is not modifiable

FINDING DUPLICATES IN A STRING

```

int main()
{
    char A[] = "finding";
    int H[26], i;

    for (i=0; A[i]!='\0'; i++)
        H[A[i]-97] += 1;

    for (i=0; i<26; i++)
        if (H[i]>1)
            printf("%c", i+97);
}

```

FINDING DUPLICATES IN A STRING USING BITWISE OPERATIONS

1. Left Shift <<
 2. Bits ORing (Merging)
 3. Bits ANDing (Masking)
- most significant bit

Manipulating bits of a byte

(Represents how 8 is stored)

7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0
128	64	32	16	8	4	2	1

1 BYTE = 8 BITS

char H=8;

least significant bit

(1) LEFT SHIFT

$$H = H \ll 2$$

7	6	5	4	3	2	1	0
0	0	1	0	0	0	0	0
128	64	32	16	8	4	2	1

(3) BITS ANDing

$$\begin{array}{r}
 a = 10 \rightarrow 1010 \\
 b = 6 \rightarrow 0110 \\
 \hline
 0010 \rightarrow 2
 \end{array}$$

(2) BITS ORing

$$\begin{array}{r}
 a = 10 \rightarrow 1010 \\
 b = 6 \rightarrow 0110 \\
 \hline
 1110 \rightarrow 14
 \end{array}$$

MASKING

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

128 64 32 16 8 4 2 1

Q We want to know whether 4th bit in H is on or off.

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

128 64 32 16 8 4 2 1

Sol

$$a = 1$$

$$a = a \ll 4$$

$$a \& H$$

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

128 64 32 16 8 4 2 1

(perform ANDing)

MERGING

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

128 64 32 16 8 4 2 1

Q we want to set 3rd bit in H as on.

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

128 64 32 16 8 4 2 1

Sol

$$a = 1$$

$$a = a \ll 2$$

$$H = a \mid H$$

FINDING DUPLICATES

ASCII Codes

A	102	105	110	100	105	110	103
	f	i	n	d	i	n	g
	0	1	2	3	4	5	6
	7						

int main()

{

```
char A[] = "finding";
long int H=0, x=0;
for(i=0; A[i]!='\0'; i++)
{
```

x=1;

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

if(x & H>0)

printf("%c is duplicate", A[i]);

else

H = x | H;

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

32 bits created for 26 alphabets

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

X

}

X also consists of 32 bits

★

CHECK FOR ANAGRAM



When two strings are made up of same alphabets

A	100	101	99	105	109	97	108	
	d	e	c	i	m	a	l	\0
	0	1	2	3	4	5	6	7

B	100	101	99	105	109	97	108	
	m	e	d	i	c	a	l	\0
	0	1	2	3	4	5	6	7

H

1	1	1	1			1			1	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

H

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	20	21	22	23	24	25

```
int main
```

```
{
```

```
char A[] = "decimal";
char B[] = "medical";
int i, H[26] = {0};
```

```
for (i=0; A[i]!='\0'; i++)
    H[A[i]-97] += 1; // Making it 1
```

```
for (i=0; B[i]!='\0'; i++)
    if (H[B[i]-97] < 0)
        {
```

```
        printf(" Not Anagram");
        break;
    }
```

```
if (B[i] == '\0')
    printf(" Anagram");
```

First check if the strings are of same length.

★

PERMUTATION OF A STRING

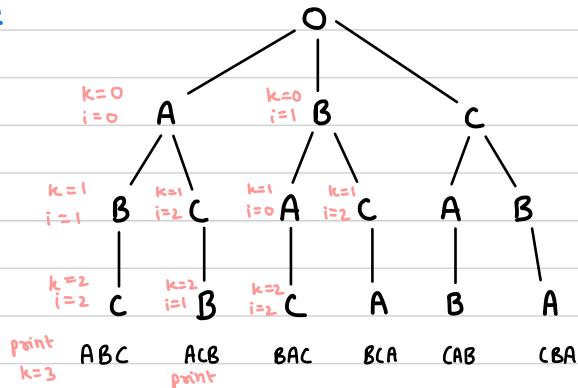
A	B	C	\o
0	1	2	3

1st Method

3!

n!

State Space Tree



Brute Force: Finding all possible permutations

Back Tracking

- Implemented using recursion
- To achieve brute force.

void perm(char s[], int k)

```

{
    static int A[10] = {0};
    static char Res[10];
    int i;
    if (s[k] == '\o')
    {
        Res[k] = '\o';
        printf("%s", Res);
    }
    else
    {
        for (i=0; s[i]!='\o'; i++)
            if (A[i]==0)
            {
                Res[k] = s[i];
                A[i] = 1;
                perm(s, k+1);
                A[i] = 0;
            }
    }
}
  
```

S	A	B	C	\o
0	1	2	3	

main()

{

char s[] = "ABC";
perm(s, 0);

A	0	0	0	0
0	1	2	3	

Res	A	B	C	\o
0	1	2	3	

2nd Method

void perm(char s[], int l, int h)

```

{
    int i;
    if (l==h)
        printf("%s", s);
    else
        for (i=l; i<=h; i++)
            swap(s[l], s[i]);
            perm(s, l+1, h);
            swap(s[l], s[i]);
}
  
```

}

}



MATRICES

SPECIAL MATRICES

↳ Only square matrix
 $n \times n$

1. Diagonal Matrix
2. Lower Triangular
3. Upper Triangular
4. Symmetric Matrix
5. Tridiagonal Matrix
6. Band Matrix
7. Toeplitz Matrix
8. Sparse Matrix

1. DIAGONAL MATRIX

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

A	3	7	4	9	6
	0	1	2	3	4

```
int A[5];
void set ( int A[], int i, int j, int x )
{
```

ROWS COLUMNS ELEMENT TO BE INSERTED

```
    if (i==j)
        A [i-1] = x;
```

}

```
void get ( int A[], int i, int j )
{
```

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

}

C++ Class For Diagonal Matrix

```
class Diagonal()
```

```
{
```

```
private :
```

```
    int n;  
    int *A;
```

```
public :
```

```
    Diagonal(int n)  
{
```

```
        This → n = n;
```

```
        A = new int(n);  
    }
```

```
    void set(int i, int j, int k);
```

```
    void 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[i-1];
```

```
    else
```

```
        return 0;
```

```
}
```

```
void Diagonal :: Display()
```

```
{
```

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

```
    {
```

```
        if (i == j)
```

```
            cout << A[i-1];
```

```
        else
```

```
            cout << " 0 ";
```

```
}
```

```
    cout << endl;
```

```
}
```

LOWER TRIANGULAR MATRIX

$$M = \begin{bmatrix} a_{11} & 0 & 0 & 0 & 0 \\ a_{21} & a_{22} & 0 & 0 & 0 \\ a_{31} & a_{32} & a_{33} & 0 & 0 \\ a_{41} & a_{42} & a_{43} & a_{44} & 0 \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{bmatrix}$$

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

$$M[i, j] = \text{Non Zero} \quad \text{if } i \geq j$$

$$\begin{aligned} \text{Non Zero} &= 1+2+3+4+5 \\ &= 1+2+3+4+\dots+n \\ &= \frac{n(n+1)}{2} \end{aligned}$$

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

ROW MAJOR

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

row1 row2 row3 row4 row5

$$\text{Index}(A[4][3]) = [1+2+3]+2 = 8$$

$$\text{Index}(A[5][4]) = [1+2+3+4]+3 = 13$$

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

COLUMN MAJOR

a_{11}	a_{21}	a_{31}	a_{41}	a_{51}	a_{22}	a_{32}	a_{42}	a_{52}	a_{23}	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	

col1 col2 col3 col4 col5

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

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

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

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

$$\begin{aligned} &= [n(j-1) - [1+2+3+\dots+(j-2)]] + (i-j) \\ &= [n(j-1) - \frac{(j-2)(j-1)}{2}] + (i-j) \end{aligned}$$

UPPER TRIANGULAR MATRIX

$$M = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ 0 & a_{22} & a_{23} & a_{24} & a_{25} \\ 0 & 0 & a_{33} & a_{34} & a_{35} \\ 0 & 0 & 0 & a_{44} & a_{45} \\ 0 & 0 & 0 & 0 & a_{55} \end{bmatrix}$$

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

$$M[i, j] = \text{Non Zero} \quad \text{if } i \leq j$$

$$\begin{aligned} \text{Non Zero} &= 5 + 4 + 3 + 2 + 1 \\ &= \frac{n(n+1)}{2} \end{aligned}$$

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

ROW MAJOR

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						

row1 row2 row3 row4 row5

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

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

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

COLUMN MAJOR

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						

$$\text{INDEX}(A[4][5]) = [1+2+3+4]+3 = 13$$

$$\text{INDEX}(A[i][j]) = [1+2+3+\dots+j-1] + i-1 = \left[\frac{j(j-1)}{2} \right] + i-1$$

SYMMETRIC MATRIX

$$M = \begin{bmatrix} 2 & 2 & 2 & 2 & 2 \\ 2 & 3 & 3 & 3 & 3 \\ 2 & 3 & 4 & 4 & 4 \\ 2 & 3 & 4 & 5 & 5 \\ 2 & 3 & 4 & 5 & 6 \end{bmatrix}$$

if $M[i, j] = M[j, i]$

Either we can store lower triangular matrix, or we can store upper triangular matrix

TRI DIAGONAL MATRIX

$$M = \begin{bmatrix} a_{11} & a_{12} & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 0 \\ 0 & a_{32} & a_{33} & a_{34} & 0 \\ 0 & 0 & a_{43} & a_{44} & a_{45} \\ 0 & 0 & 0 & a_{54} & a_{55} \end{bmatrix}$$

a_{21}	a_{32}	a_{43}	a_{54}	a_{11}	a_{22}	a_{33}	a_{44}	a_{55}	a_{12}	a_{23}	a_{34}	a_{45}
0	1	2	3	4	5	6	7	8	9	10	11	12

- Main diagonal $i - j = 0$
- Lower diagonal $i - j = 1$
- Upper diagonal $i - j = -1$

Index ($A[i][j]$)

case 1 : if $i - j = 1$ index = $i - 1$

$| i - j | \leq 1$

case 2 : if $i - j = 0$ index = $n - 1 + i - 1$

$M[i, j] = \text{Non Zero}$ if $ i - j \leq 1$
$M[i, j] = \text{Zero}$ if $ i - j > 1$

case 3 : if $i - j = -1$ index = $2n - 1 + i - 1$

$5 + 4 + 4$
$n + n - 1 + n - 1$
$3n - 2$

SQUARE BAND MATRIX (Same as TRI DIAGONAL matrix)

When there are more than one diagonals below the main diagonal and the number of lower and upper diagonal is equal.

TOEPLITZ 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

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

row

column

$$M[i, j] = M[i-1, j-1]$$

No of elements we want to store: $n + n-1$

Index A[i][j]

case 1: if $i <= j$ Index = $j-1$

case 2 : if $i > j$ Index = $n + i - j - 1$

CREATING A DYNAMICALLY ALLOCATED ARRAY

```
int * A, n;  
printf (" Enter dimension");  
scanf ("%d", &n);  
A = (int *) malloc (n * sizeof (int));  
A = new int [n]; C++
```



SPARSE MATRIX

Having many number of non-zero elements

Methods for storing sparse matrix

1. Coordinate list / Three column representation
2. Compressed sparse row

1. Coordinate list / Three column representation

	1	2	3	4	5	6	7	8	9	row	column	element
1	0	0	0	0	0	0	0	3	0	8	9	8
2	0	0	8	0	0	10	0	0	0	1	8	3
3	0	0	0	0	0	0	0	0	0	2	3	8
4	4	0	0	0	0	0	0	0	0	2	6	10
5	0	0	0	0	0	0	0	0	0	4	1	4
6	0	0	2	0	0	0	0	0	0	6	3	2
7	0	0	0	6	0	0	0	0	0	7	4	6
8	0	9	0	0	5	0	0	0	0	8	2	9
	8 x 9 72 elements									8	5	5
	72 x 2 = 144 bytes											

2. Compressed sparse row

A [3, 8, 10, 4, 2, 6, 9, 5] → Non Zero elements

IA [0, 1, 3, 3, 4, 4, 5, 6, 8] → Row Numbers

→ Cumulative sum of number of non zero elements in each row

JA [8, 3, 6, 1, 3, 4, 2, 5]

No of column in which each non zero element is located

8 + 9 + 8 = 25 x 2 = 50 bytes

1. Coordinate List / Three column representation (ADDITION)

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

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

$$B = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 3 & 0 & 0 & 5 & 0 \\ 3 & 0 & 0 & 2 & 0 & 0 & 7 \\ 4 & 0 & 0 & 0 & 9 & 0 & 0 \\ 5 & 8 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

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

$$C = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 0 & 0 & 0 & 6 & 0 & 0 \\ 2 & 0 & 10 & 0 & 0 & 5 & 0 \\ 3 & 0 & 2 & 2 & 5 & 0 & 7 \\ 4 & 0 & 0 & 0 & 9 & 0 & 0 \\ 5 & 12 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

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

Q

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

4x5 mxn

0	1	2	3	4	5
i	4	1	2	3	4
j	5	3	1	4	1
k	5	7	2	5	9

CREATING SPARSE MATRIX PROGRAM

Struct Element

{

int ij;
int jj;
int xj;

}

Struct sparse

{

int m;
int n;
int num;
Struct Element *e;

}

void main()

{

Struct sparse s;
create(&s);

}

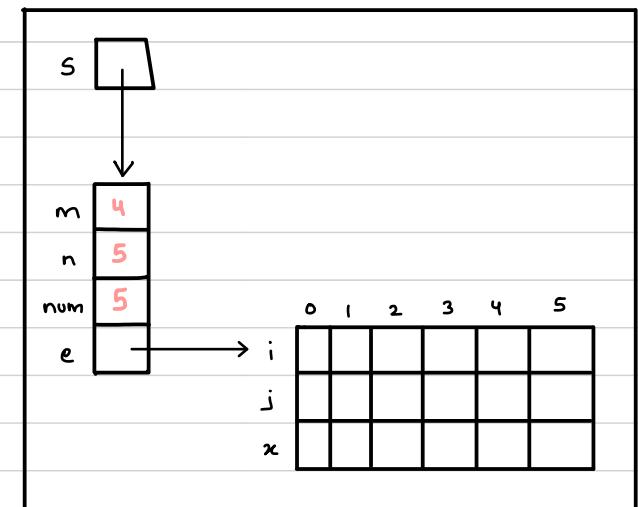
s->e = (Struct Element *) malloc
(s->num * sizeof(Struct Element))

void create (struct sparse *s)

{

int ij;
printf(" Enter dimensions");
scanf (" %d %d ", &s->m, &s->n);
printf(" Enter number of non-zero elements");
scanf (" %d ", &s->num);
s->e = new Elements [s->num];
printf(" Enter all elements ");

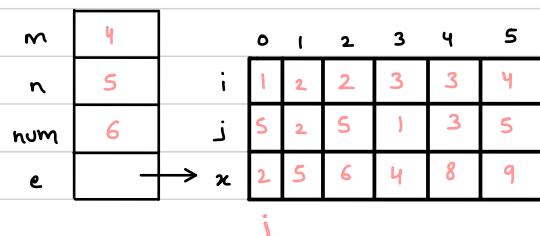
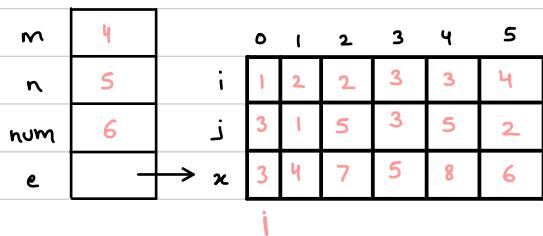
for (i=0; i < s->num; i++)
scanf(" %d %d %d ", &s->e[i].i,
&s->e[i].j,
&s->e[i].x);



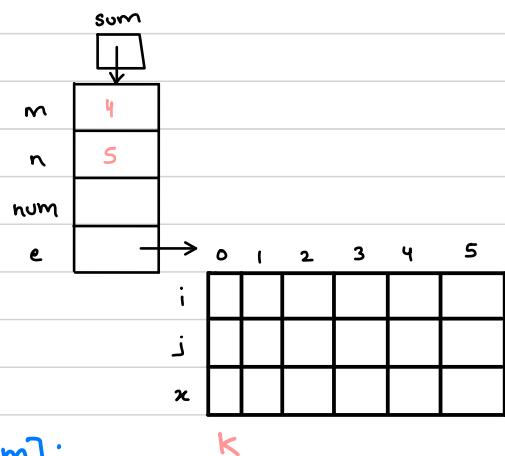
ADDING SPARSE MATRIX PROGRAM

	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

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



```
add ( struct sparse *S1 , struct sparse S2 )
{
    struct Sparse *Sum;
    if ( S1->m! = S2->m || S1->n! = S2->n )
        return 0;
    Sum = new sparse;
    Sum->m = S1->m;
    Sum->n = S1->n;
    Sum->e = new Element [S1->num + S2->num];
```



```
while (i < s1->num && j < s2->num)  
{
```

$\text{if } (\text{s1.e[i].i} < \text{s2.e[j].i})$
 $\text{sum} \rightarrow \text{e[k++]} = \text{s1} \rightarrow \text{e[i++]}$;

else if ($s_1 \rightarrow e[i].i > s_2 \rightarrow e[i].i$)
 $sum \rightarrow e[k++] = s_2 \rightarrow e[i]$

```
else  
{
```

```

if ( s1 → e[i] . j < s2 → e[j] . j )
    sum → e[k++ ] = s1 → e[i++ ];
else if ( s1 → e[i] . j > s2 → e[j] . j )
    sum → e[k++ ] = s2 → e[j++ ];

```

else
{

$\text{sum} \rightarrow e[k] = S1 \rightarrow e[i++];$
 $\text{sum} \rightarrow e[k++] \cdot x += S2 \rightarrow e[j++] \cdot x;$

POLYNOMIAL REPRESENTATION

1. Polynomial Representation
 2. Evaluation of Polynomial
 3. Addition of two Polynomials

$$p(x) = 3x^5 + 2x^4 + 5x^2 + 2x + 7$$

coeff	3	2	5	2	7
exp	5	4	2	1	0

$$n = 5$$

Strut Term

3

int coeff;
int Exp;

Strut Poly

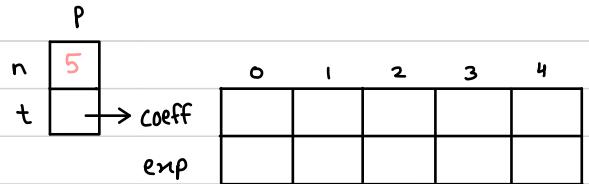
{

```
int n;  
Struct Team *t;
```

```

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 : y.d ", i-1);
    scanf("y.d %d", &p.t[i].co
}

```



EVALUATION

```
Struct Poly P;  
x=5; sum=0;
```

```

for(i=0 ; i<p.n ; i++)
    sum += p.t[i].coeff * pow(x, p.t[i].Exp);

```

POLYNOMIAL ADDITION

$$p_1(x) = 5x^4 + 2x^2 + 5$$

$$p_2(x) = 6x^4 + 5x^3 + 9x^2 + 2x + 3$$

P		0	1	2	3	4
n	3					
t						
coeff	5	2	5			
exp	4	2	0			
i						

P		0	1	2	3	4
n	5					
t						
coeff	6	5	9	2	3	
exp	4	3	2	1	0	
j						

while (i < p1.n && j < p2.n)

{

 if (p1.t[i] · Exp > p2.t[j] · Exp)
 p3.t[k++] = p1.t[i++];

 else if (p2.t[j] · Exp > p1.t[i] · Exp)
 p3.t[k++] = p2.t[j++];

 else

{

 p3.t[k] · Exp = p1.t[i] · Exp;
 p3.t[k] · coeff = p1.t[i] · coeff + p2.t[j] · coeff;

}

}

P		0	1	2	3	4
n	5					
t						
coeff	11	5	11	2	8	
exp	4	3	2	1	0	
k						

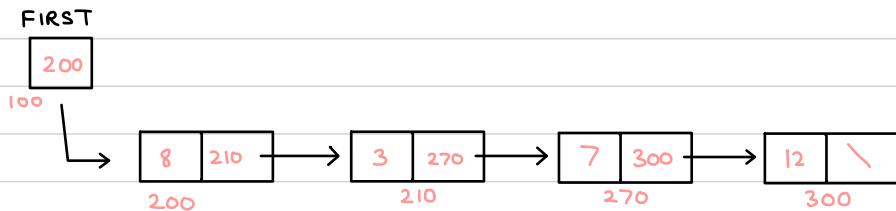


LINKED LIST

1. Problem with arrays : Fixed Size

Q What is a linked list?

Ans Linked list is a collection of nodes where each node contain data and pointers to next node.



Struct Node
{

SELF REFERENTIAL STRUCTURE

```

int data;
Struct Node * next;
}
  
```

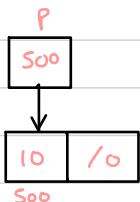
$\frac{2}{\text{same as datatype}}$
 $\frac{2}{4 \text{ bytes}}$

```

Struct Node * p;
p = (Struct Node *)malloc(sizeof(Struct Node));
p = new Node; C++
  
```

```

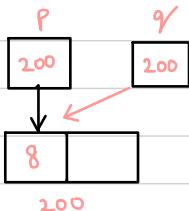
p → data = 10;
p → next = 0;
  
```



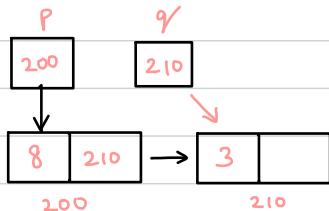
X ————— X

Struct Node * p, * .

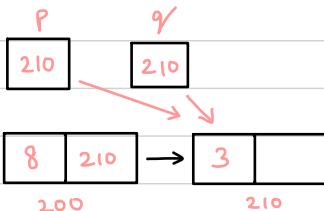
1. q ← p



2. q = p → next



3. p = p → next



```
struct Node *p = NULL;
```

```
if (p == NULL)  
if (p == 0)  
if (!p)  
if (p->next == NULL)
```



To check if pointer
is not pointing anywhere

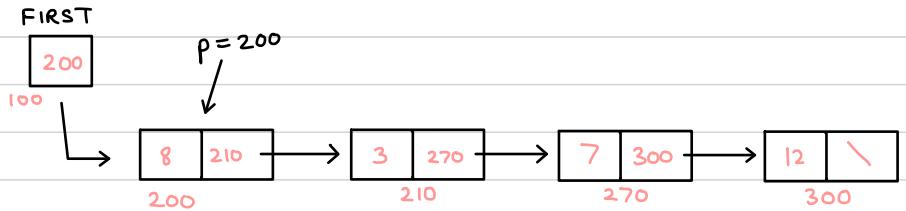
```
if (p != NULL)  
if (p != 0)  
if (p)  
if (p->next != NULL)
```



To check if pointer
is not NULL.

TRaversing THROUGH LINKED LIST

```
struct Node * p = first;  
while (p != 0)  
{  
    p = p->next;  
}
```



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

```
struct Node
```

```
{
```

```
    int data;
```

```
    struct Node *next;
```

```
} *first = NULL;
```

Check `struct Node * first = NULL` in main

```
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;
    }
}

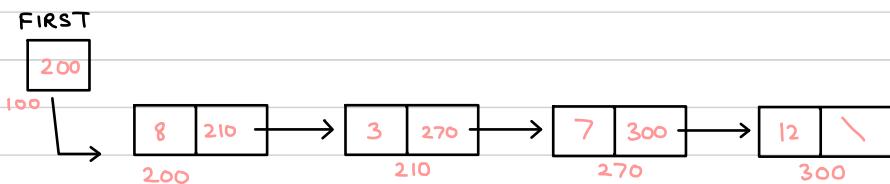
```

```

void main()
{
    struct Node *temp;
    int A[] = { 3,5,7,10,25, 8 ,32,2};
    create (A,8);
    Display (first);
}

```

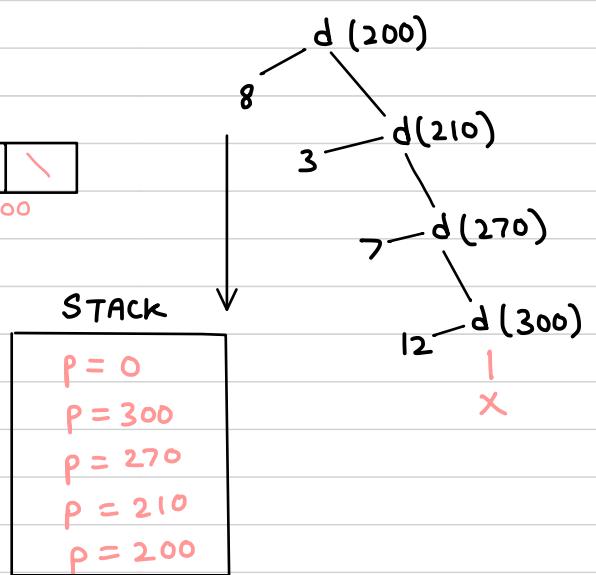
RECURSIVE DISPLAY OF LINKED LIST



```

void Display( struct Node *p)
{
    if (p!=NULL)
    {
        printf(" %d ", p->data);
        Display ( p->next);
    }
}

```

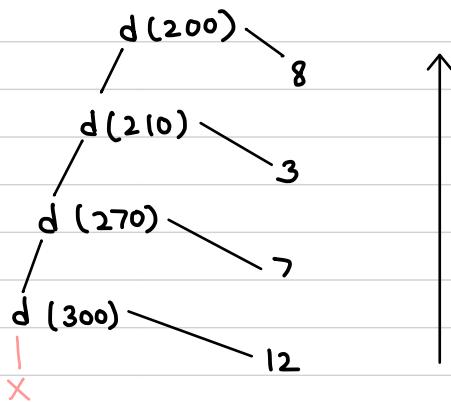


```

void Display (struct Node *p)
{
    if (p != NULL)
    {
        Display (p->next);
        printf ("%d", p->data);
    }
}

```

$O(n)$



COUNTING NODES IN LINKED LIST

```

int count (struct Node *p)
{
    int c=0;
    while (p!=0)
    {
        c++;
        p=p->next;
    }
    return (c);
}

```

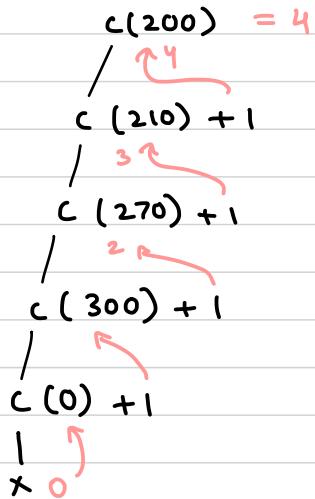
$O(n)$

RECURSIVE FUNCTION FOR COUNTING NUMBER OF NODES

```

int count (struct Node *p)
{
    if (p==0)
        return 0;
    else
        return count (p->next)+1;
}

```



SUM OF ALL ELEMENTS IN A LINKED LIST

```

int add (struct Node *)
{
    int sum=0;
    while (p)
    {
        sum = sum + p->data;
        p = p->next;
    }
    return sum;
}

```

USING RECURSION

```

int Add (struct Node *p)
{
    if (p==0)
        return 0;
    else
        return Add (p->next) + p->data;
}

```

$O(n)$

MAXIMUM ELEMENT IN A LINKED LIST

```
int max( struct Node *p)
{
    int m = -32768;
    while (p)
    {
        if (p->data > m)
            m = p->data;
        p = p->next;
    }
    return (m);
}
```

RECURSION

```
int max( Node *p)
{
    int x = 0;
    if (p == 0)
        return MIN_INT;
    else
    {
        x = max(p->next);
        if (x > p->data)
            return x;
        else
            return p->data;
    }
}
```

SEARCHING IN A LINKED LIST

Binary search is not suitable for linked list as we cannot go directly in the middle of the list

LINEAR SEARCH

```
Node* Search( struct Node *p , int key)
{
    while (p != NULL)
    {
        if (key == p->data)
            return (p);
        p = p->next;
    }
    return NULL;
}
```

RECURSIVE

```
Node* Search( Node *p , int key)
{
    if (p == NULL)
        return NULL;
    if (key == p->data)
        return (p);
    return Search(p->next , key);
}
```

IMPROVING SEARCHING

```

Node *search ( Node *p, int key)
{
    Node *q = NULL;

    while ( p != NULL )
    {
        if ( key == p->data )
        {
            q->next = p->next;
            p->next = first;
            first = p;
        }
        q = p;
        p = p->next;
    }
}

```

```

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; i++ )
            p = p->next;

```

```

        if ( p )
        {

```

```

            t = new Node;
            t->data = x;
            t->next = p->next;
            p->next = t;
        }
    }

```

}

}

INSERTING IN A LINKED LIST

Before first Node

```

Node *t = new Node;
t->data = x;
t->next = first;
first = t

```

Pos == 4

```

Node *t = new Node;
t->data = x;
p = first;
for ( i = 0; i < pos - 1; i++ )
    p = p->next;
t->next = p->next;
p->next = t;

```

```

        p = first;
        for ( i = 0; i < pos - 1; i++ )
            p = p->next;
        if ( p )
        {
            t = new Node;
            t->data = x;
            t->next = p->next;
            p->next = t;
        }
    }
}

```

}

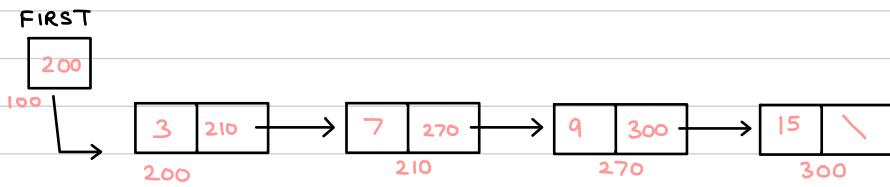
}

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;
    }
}
```

INSERTING IN A SORTED LIST



$p = \text{first};$
 $q = \text{NULL}$ (Tailing Pointer)

```
while (p && p->data < n)
{
```

```
    q = p;
    p = p->next;
}
```

```
t = new Node;
t->data = n;
t->next = q->next;
q->next = t;
```

DELETION FROM LINKED LIST

(1) Deletion of first node

```
Node *p = first;      O(1)  
first = first -> next;  
x = p -> data;  
free(p);
```

(2) Deletion from a given position

```
Node *p = first;      min O(1)  
Node *q = NULL;      max O(n)
```

```
for (i=0 ; i < pos-1; i++)  
{  
    q = p;  
    p = p -> next;  
}
```

```
q -> next = p -> next;  
x = p -> data;  
free(p);
```

CHECK IF LIST IS SORTED

```
int x = -32768;      O(n)  
Node *p = first;  
while (p != NULL)  
{  
    if (p -> data < x)  
        return -1;  
    x = p -> data;  
    p = p -> next;  
}  
return 0;
```

REMOVE DUPLICATES FROM LIST

Node * p = first; O(n)
Node * q = first → next;

```
while ( q != NULL )
{
    if ( p → data != q → data )
    {
        p = q;
        q = q → next;
    }
    else
    {
        p → next = q → next;
        free (q);
        q = p → next;
    }
}
```

* REVERSING A LINKED LIST

- (1) Reversing Elements
- (2) Reversing Links

Reversing links is preferred over reversing elements because maybe there are lots of values in a single node.

(1) Reversing Elements

First create an array A equal to size of length of linked list.

p = first; O(n)
i = 0;

```
while ( p != NULL )
{
    A[i] = p → data;
    p = p → next;
    i++;
}
```

p = first; i --;
while (p != NULL)
{
 p → data = A[i--];
 p = p → next;
}

(2) Reversing Links

TRACE IT!

```
p = first;  
q = NULL;  
r = NULL;  
while (p != NULL)  
{  
    r = q;  
    q = p;  
    p = p->next;  
    q->next = r;  
}  
first = q;
```

] Sliding pointers

REVERSING LINKED LIST USING RECURSION

```
Void Reverse ( Node *q , Node *p )  
{  
    if ( p != NULL )  
    {  
        Reverse ( p , p->next )  
        p->next = q ;  
    }  
    else  
        first = q ;  
}
```

CONCATENATING TWO LINKED LIST

```
p = first; O(n)  
while ( p->next != NULL )  
    p = p->next  
p->next = second;  
second = NULL;
```

MERGING TWO LINKED LIST

We have two sorted linked list and we want to combine it into a single list.

```
if (first → data < second data)           Θ(m+n)  
{
```

```
    third = last = first;  
    first = first → next;  
    last → next = NULL;
```

```
}
```

```
else  
{
```

```
    third = last = second;  
    second = second → next;  
    last → next = NULL;
```

```
}
```

```
while (first != NULL and 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;
```

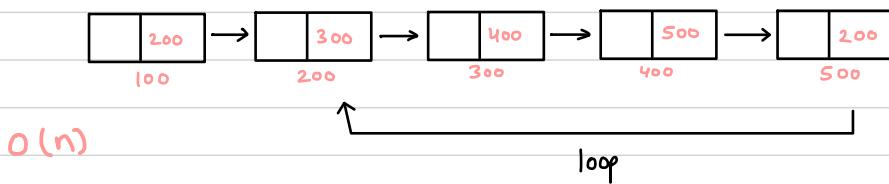
```
}
```

```
}  
if (first != NULL)  
    last → next = first;
```

```
else
```

```
    last → next = second;
```

CHECK FOR LOOP IN LINKED LIST



```
int isloop ( Struct Node *f )
{
```

```
    Struct Node *p, *q;
    p=q=f;
```

```
    do
```

```
{
```

```
    p = p->next;
    q = q->next;
    q = q ? q->next : q;
}
```

while (p != q || p != f);

```
if (p==q)
```

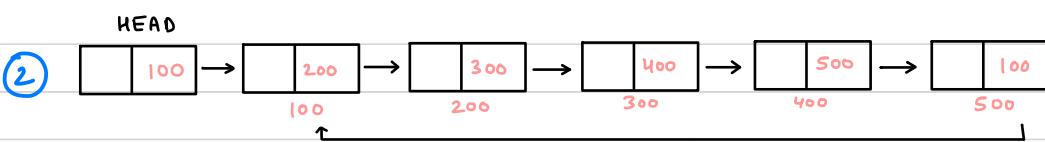
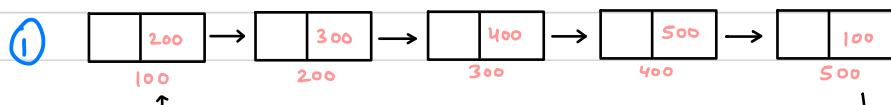
```
    return 1;
```

```
else
```

```
    return 0;
```

```
}
```

CIRCULAR LINKED LIST



DISPLAY CIRCULAR LINKED LIST

```
Void Display ( Node *p )
{
```

```
    do
```

```
        Display (Head);
```

```
{
```

```
        printf (" %d ", p->data);
```

```
        p = p->next;
```

```
}
```

```
while ( p != Head );
```

```
}
```

DISPLAY CIRCULAR LINKED LIST USING RECURSION

```
void Display (Node *p)
{
    static int flag = 0 ;
    if ( p != Head || flag == 0 )
    {
        flag = 1 ;
        printf ("%d", p->data) ;
        Display ( p->next ) ;
    }
}
```

CREATION OF CIRCULAR LINKED LIST

```
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;
    }

    int main()
    {
        int A[] = {2,3,4,5,6};
        create (A, 5);
    }
}
```

INSERTING IN A CIRCULAR LINKED LIST

```
void Insert ( struct Node *p, int index, int x)
{
    struct Node *t;
    int i;

    if (index < 0 || index > length())
        return;

    if (index == 0)
    {
        t = (struct Node *) malloc ( sizeof (struct Node));
        t->data = x;

        if (Head == NULL)
        {
            Head = t;
            Head->next = Head;
        }
        else
        {
            while ( p->next != Head )
                p = p->next;
            p->next = t;
            Head = t;
            t->next = Head;
        }
    }

    else
    {
        for (i=0; i < index-1; i++)
            p = p->next;
        t = (struct Node *) malloc ( sizeof (struct Node));
        t->data = x;
        t->next = p->next;
        p->next = t;
    }
}
```

DELETION IN CIRCULAR LINKED LIST

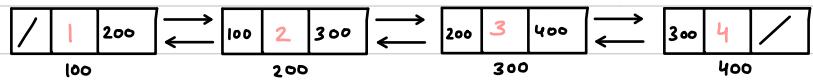
```
int Delete ( struct Node *p, int index )
{
    struct Node *q;
    int i, x;

    if ( index < 0 || index > length ( Head ) )
        return -1;

    if ( index == 1 )
    {
        while ( p->next != Head )
            p = p->next;
        x = Head->data;

        if ( Head == p )           // If there is only one node
        {
            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 );
    }
}
```

INSERT IN A DOUBLY LINKED LIST



Struct Node

{

```
Struct Node * prev;  
int data;  
Struct Node * next;  
} * first = NULL;
```

void create (int A[], int n)

{

```
Struct Node * t, * last;  
int i;
```

```
first = (Struct Node *) malloc ( sizeof (Struct Node));  
first -> data = A[0];  
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
```

}

INSERT IN A DOUBLY LINKED LIST

(1) Insert at first node

```

Node *t = new Node;
t → data = x;
t → prev = NULL;
t → next = first;
first → prev = t;
first = t;

```

(2) Insert at any given position

```

Node *t = new Node
t → data = x;
for (i=0; i<pos-1; i++)
    p = p → next;
t → next = p → next;
t → prev = p;
if (p → next)
    p → next → prev = t
    p → next = t;

```

// To check if a node is available after
p or not

DELETE FROM A DOUBLY LINKED LIST

(1) Deleting first node

```

p = first;
first = first → next;
x = p → data;
delete p;
if (first) // If first is not NULL
    first → prev = NULL;

```

(2) Deleting Node from given index

```

p = first;
for (i=0; i<pos-1; i++)
    p = p → next;
p → prev → next = p → next;
if (p → next)
    p → next → prev = p → prev;
x = p → data;
delete p;

```

if p → next is not NULL

REVERSING A DOUBLY LINKED LIST

```

p = first;
while (p)
{
    temp = p->next;
    p->next = p->prev;
    p->prev = temp;
    p = p->prev

    if (p != NULL && p->next == NULL)
        first = p;
}

```

FINDING MIDDLE NODE IN A LINKED LIST

```

p = q = first;
while (q)
{
    q = q->next;
    if (q)
        q = q->next;
    if (p)
        p = p->next;
}

```

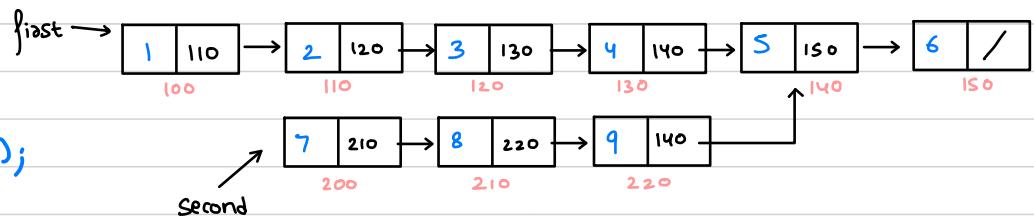
Two pointers p and q will move at the same time.
q will move 2 nodes and p will move 1 node.

FINDING INTERSECTION POINT OF TWO LINKED LIST

```

p = first
while (p != NULL)
    push (&stk1, p);
p = second;
while (p != NULL)
    push (&stk2, p);
while (stacktop(stk1) == stacktop(stk2))
{
    p = pop (&stk1);
    pop (&stk2);
}

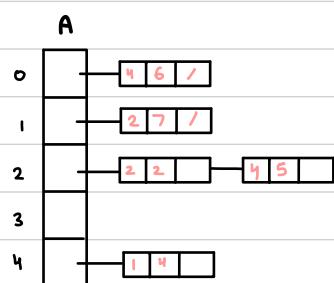
```





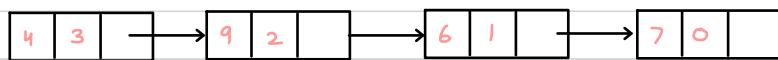
SPARSE MATRIX USING LINKED LIST

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



POLYNOMIAL REPRESENTATION USING LINKED LIST

$$p(x) = 4x^3 + 9x^2 + 6x + 7$$





Recursion uses stack STACK

↳ LIFO: Last In First Out

It is a collection of elements that follow LIFO for insertion and deletion.

ADT Stack (Abstract data type)

- Data :
1. Space for storing elements
2. Top pointer

Implementation of stack using

1. Array
2. Linked List

- Operations:
1. push(x)
2. pop()
3. peek(index)
4. StackTop()
5. isEmpty()
6. isFull()

IMPLEMENTATION OF STACK USING ARRAY

Struct stack

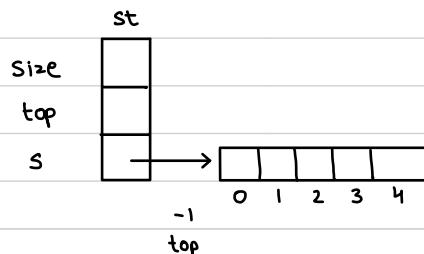
{

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]; *for heap memory*
st.top = -1;

}



Stack empty - if ($\text{top} == -1$)
Stack full - if ($\text{top} == \text{size} - 1$)

push() $O(1)$

```
void push (stack *st, int x)
{
    if (st->top == st->size - 1)
        printf ("Stack Overflow");
    else
    {
        st->top++;
        st->s[st->top] = x;
    }
}
```

pop() $O(1)$

```
int pop (stack *st)
{
    int x = -1;
    if (st->top == -1)
        printf ("Stack Underflow");
    else
    {
        x = st->s[st->top];
        st->top--;
    }
    return x;
}
```

peek()

```
int peek (stack st, int pos)
{
    int x = -1;
    if (st.top - pos + 1 < 0)
        printf ("Invalid Position");
    else
        x = st.s[st.top - pos + 1];
    return x;
}
```

pos	Index = Top - pos + 1
1	3
2	2
3	1
4	0

isEmpty()

```
int isEmpty (stack st)
{
    if (st.top == -1)
        return 1;
    else
        return 0;
}
```

StackTop()

```
int StackTop (stack st)
{
    if (st.top == -1)
        return -1;
    else
        return st.s[st.top];
}
```

isFull()

```
int isFull (stack st)
{
    if (st.top == -1)
        return 1;
    else
        return 0;
}
```

STACK USING LINKED LIST

```
Struct Node
{
```

```
    int data;
    Struct Node *next;
}
```

Empty : if (top == NULL)
Full : Node *t = new Node;
if (t == NULL)

push()

```
void push( int x)
{
    Node * t = new Node;

    if (t == NULL)
        printf("Stack Overflow");
    else
    {
        t->data = x;
        t->next = top;
        top = t;
    }
}
```

peek()

```
int Peek( int pos)
{
    int i;
    Node *p = top;

    for (i=0; p != NULL && i < pos-1; i++)
        p = p->next;
```

pop()

```
int pop()
{
    Node *p;
    int x=-1;

    if (top == NULL)
        printf("Stack is empty");
    else
    {
        p = top;
        top = top->next;
        x = p->data;
        free(p);
    }

    return x;
}
```

```
if (p != NULL)
    return p->data;
else
    return -1;
```

Stack top()

```
int stacktop()
{
    if (top)
        return top->data;
    return -1;
}
```

isFull

```
int isFull()
{
    Node *t = new Node;
    int r = t ? 1 : 0;
    free(t);
    return r;
}
```

isEmpty

```
int isEmpty()
{
    return Top ? 0 : 1;
}
```

↑ False
 ↑ True

Parenthesis Matching

exp	((a	+	b)	*	((-	d))	/	0
	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];
```

] initializing of stack

```
for (i=0; exp[i] != '\0'; i++)
{
    if (exp[i] == '(')
        push(&st, exp[i]);
    else if (exp[i] == ')')
        if (isEmpty(st))
            return false;
        else
            pop(&st);
}
return isEmpty(st) ? true : false;
```

ASCII

(40
)	41
[91
]	93
{	123
}	125

INFIX TO POSTFIX CONVERSION

1. What is postfix
2. Why postfix
3. Precedence
4. Manual Conversion

1. Infix: Operand Operator Operand
 $a+b$

2. Prefix: Operator Operand Operand
 $+ab$

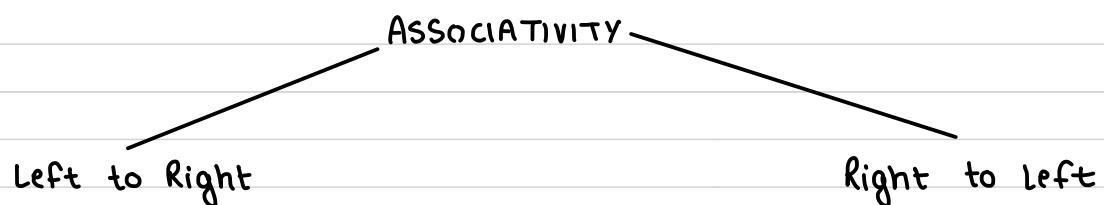
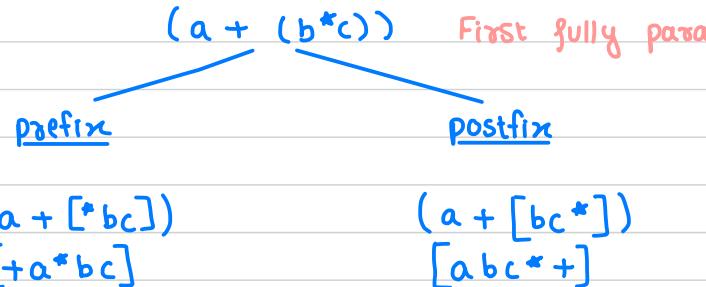
3. Postfix: Operand Operand Operator
 $ab+$

SYMBOL	PRECEDENCE	ASSOCIATIVITY
$+, -$	1	L-R
$\times, /$	2	L-R
\wedge	3	R-L
\neg	4	R-L
$()$	5	L-R

Eg

$a + b * c$

Unary minus



$a + b + c - d$

$a = b = c = 5$

$((a+b)+c)-d$

$(a = (b = (c = 5)))$

Power operator Example

a^b^c

$(a^b)^c$

Postfix: (a^b^c)
 $abc^{^n}$

Unary Operators Example

(1) $-a$ negation of a

pre : $-a$

post : $a-$

$(-(-a))$

(2) $*p$

pae : $*p$

post : p^*

$(*(*p))$

(3) $n!$

pae : $!n$

post : $n!$

(4) $\log x$

pre : $\log x$

post : $x \log$

Example:

$$\begin{aligned}
 & -a + b * \log n \\
 & -a + b * \log [n!] \\
 & -a + b * [n! \log] \\
 & [a-] + b * [n! \log] \\
 & [a-] + [bn! \log^*] \\
 & a - bn! \log^* +
 \end{aligned}$$

INFIX TO POSTFIX CONVERSION

$a + b * c - d / e$

Symbol	Stack	Postfix
a		a
+	+	a
b	+	ab
*	*, +	ab
c	*, +	abc
-	-	abc *+
d	-	abc *+d
/	/, -	abc *+d
e	/, -	abc *+ de

SYMBOL	PRECEDENCE	ASSOCIATIVITY
+, -	1	L-R
*, /	2	L-R

$abc * + de / -$ Ans

PROGRAM

infix a + b * c - d / e \0
 0 1 2 3 4 5 6 7 8 9

```
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;
```

```
int pre(char x)
{
    if(x == '+' || x == '-')
        return 1;
    else if(x == '*' || x == '/')
        return 2;
    return 0;
}
```

```
int isOperand(char x)
{
    if(x == '+' || x == '-' || x == '*' || x == '/')
        return 0;
    else
        return 1;
}
```

Q

$$\begin{aligned}
 & ((a+b)*c) - d^a e^f \\
 & ([ab+]^*c) - d^a e^f \\
 & [ab+c^*] - d^a e^f \\
 & [ab+c^*] - d^a [ef^a] \\
 & [ab+c^*] - [def^{aa}] \\
 & ab+c^* def^{aa} -
 \end{aligned}$$

SYMBOL	OUT STACK PRE	IN STACK PRE
+,-	1	2
* , /	3	4
^	6	5
(7	0
)	0	?

closing bracket ←
cannot be pushed into
Stack

because of
R-L associativity

EVALUATION OF POSTFIX

35 * 62 / + 4 -

SYMBOL	STACK	OPERATION
3	3	
5	5,3	
*	5 * 3	
	15	
6	6,15	
2	2,6,15	
/	6 / 2	
	3,15	
+	15 + 3	
	18	
4	4,18	
-	18 - 4	
	14	

$$n = 6 + 5 + 3 * 4$$

$$n = 65 + 34 *$$

*
Here + gets executed first instead of * because precedence and associativity are meant for parenthesisation, they don't decide which operator gets executed first.

PROGRAM FOR EVALUATION OF POSTFIX

postfix 3 5 * 6 2 / + 4 - 10
 0 1 2 3 4 5 6 7 8 9

```
int Eval( char *postfix )
```

```
{  
    struct stack st;  
    int i, x1, x2, r;
```

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

```
    {  
        if ( isoperand( postfix[i] ) )  
            push( &st, postfix[i] - '0' );
```

```
        else
```

```
        {  
            x2 = pop( &st );  
            x1 = pop( &st );
```

```
        switch( postfix[i] )
```

```
        {
```

```
            case '+': r = x1 + x2; push( &st, r ); break;
```

```
            case '-': r = x1 - x2; push( &st, r ); break;
```

```
            case '*': r = x1 * x2; push( &st, r ); break;
```

```
            case '/': r = x1 / x2; push( &st, r ); break;
```

```
        }
```

```
}
```

```
    return pop( &st );
```

```
}
```

because operand will be pushed
into the stack in its ASCII
value because postfix expression
is in char.

For eg : 3

$$'51' - '48' = 3$$



QUEUE

↳ FIFO

(First In First Out)

Queue ADT

- Data: (1) Space for storing elements
 (2) Front : for deletion
 (3) Rear : for insertion

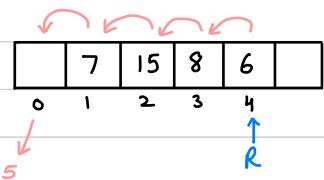
- Operations: (1) enqueue (x)
 (2) dequeue ()
 (3) isEmpty ()
 (4) isFull ()
 (5) first()
 (6) last()

- (1) Array
 (2) Linked List

QUEUE USING ARRAY

1. Queue using single pointer
2. Queue using front and rear.
3. Drawbacks of queue using array.

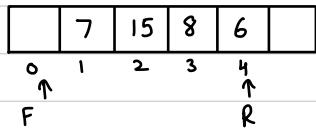
1. Queue Using Single Pointer



Insert - $O(1)$

Delete - $O(n)$

2. Queue using front and rear



initially $\text{front} = \text{rear} = -1$

Insertion : increment rear and insert $O(1)$

Deletion : increment front and delete $O(1)$

Empty : if ($\text{front} == \text{rear}$)

Full : if ($\text{rear} == \text{size} - 1$)

PROGRAM FOR QUEUE USING ARRAY

Struct Queue

{

```
int size;
int front;
int rear;
int *Q;
```

}

int main ()

{

```
Struct Queue q;
printf(" Enter Size : ");
scanf ("%d", &q.size);
q.Q = new int[q.size];
q.front = q.rear = -1;
```

}

void enqueue (Queue *q, int x)

{

```
if (q->rear == q->size-1)
    printf (" Queue is full ");
```

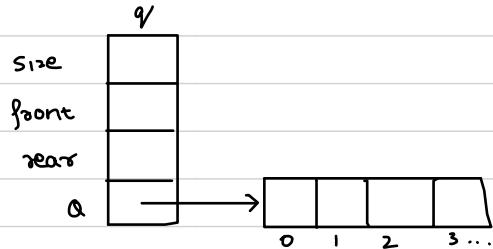
else

{

```
    q->rear++;
    q->Q[q->rear] = x;
```

}

}



Void dequeue (Queue *q)

{

int x = -1;

if (q->front == q->rear)
 printf (" Queue is empty ");

else

{

q->front++;

x = q->Q[q->front];

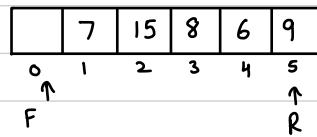
}

return x;

}

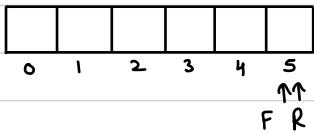
DRAWBACK OF QUEUE USING ARRAY

1. We cannot utilize space of deleted element.



2. Every location can be used only once

3. A situation where queue is empty and full



USING SPACE AGAIN SOLUTION

(1) Resetting Pointers

Whenever Front and Rear are pointing at same place, initialize them as -1

(2) Circular Queue

In circular queue, front and rear initialize with array's first position.

```
void enqueue( struct Queue *q, int x)
```

```
{
```

```
if((q->Rear+1) % q->size == q->front) To check if rear's next is front  
printf("Queue is Full");
```

```
else
```

```
{
```

```
q->Rear = (q->rear + 1) % q->size;  
q->Q[q->rear] = x;
```

```
}
```

```
}
```

$$\text{rear} = (\text{rear} + 1) \% \text{size}$$

$$0 \quad (0 + 1) \% 7 = 1$$

$$1 \quad (1 + 1) \% 7 = 2$$

$$2 \quad (2 + 1) \% 7 = 3$$

$$3 \quad (3 + 1) \% 7 = 4$$

$$4 \quad (4 + 1) \% 7 = 5$$

$$5 \quad (5 + 1) \% 7 = 6$$

$$6 \quad (6 + 1) \% 7 = 0$$

```
void dequeue( struct Queue *q)
```

```
{
```

```
int x = -1
```

```
if(q->front == q->rear)
```

```
printf("Queue is Empty");
```

```
else
```

```
{
```

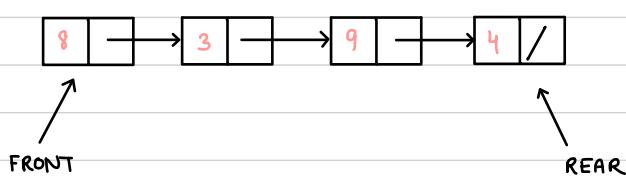
```
q->front = (q->front + 1) % q->size;  
x = q->Q[q->front];
```

```
}
```

```
return x;
```

```
}
```

QUEUE USING LINKED LIST



Empty : if (front == NULL)
 Full : Node *t = new Node;
 if (t == NULL)
 // No more nodes can be created
 i.e heap is full

void enqueue (int x) O (1)

```

{
    Node *t = new Node;
    if (t == NULL)
        printf(" Queue is Full");
    else
    {
        t->data = x;
        t->next = NULL;
        if (front == NULL)
            front = rear = t;
        else
        {
            rear->next = t;
            rear = t;
        }
    }
}
  
```

int dequeue() O (1)

```

{
    int x=-1;
    Node *p;
    if (front == NULL)
        printf(" Queue is Empty");
    else
    {
        p = front;
        front = front->next;
        x = p->data;
        free(p);
    }
    return x;
}
  
```

DE Queue

→ Double Ended Queue

QUEUE

	Insert	Delete
front	✗	✓
rear	✓	✗

DEQUEUE

	Insert	Delete
front	✓	✓
rear	✓	✓

INPUT RESTRICTED DEQUEUE

	Insert	Delete
front	✗	✓
rear	✓	✓

OUTPUT RESTRICTED DEQUEUE

	Insert	Delete
front	✓	✓
rear	✓	✗

PRIORITY QUEUES

1. Limited Set of priorities
2. Element Priority

1. Limited Set of priorities

Priorities = 3

Element →	A	B	C	D	E	F	G	H	I	J
Priority →	1	1	2	3	2	1	2	3	2	2

Priority Queues

Q_1	A	B	F			
Q_2	C	E	G	I	J	
Q_3	D	H				

DELETION

For deletion, elements of Q_1 will be deleted first, then Q_2 and then Q_3

Elements will be deleted in FIFO

INSERTION

2. Element Priority

Elements → 6, 8, 3, 10, 15, 2, 9, 17, 5, 8

- Element is itself a priority
- Smaller number higher priority

1. Insert in same order $O(1)$

Delete max priority by searching it $O(n)$

2. Insert in increasing order of priority $O(n)$

Delete the last element of array $O(1)$

QUEUE using 2 STACKS

assume stacks to be
implemented using linked
list

enqueue (int x)
{

 push (ds1, x);

int dequeue ()
{

 int x = -1;
 if (isEmpty (s2))
 {

 if (isEmpty (s1))
 {

 printf ("Queue Empty");
 return x;

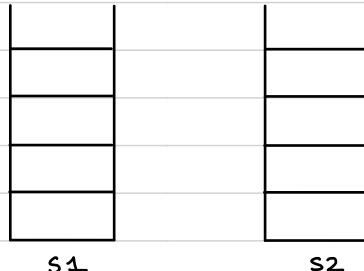
 }

 else
 {

 while (!isEmpty (s1))
 push (ds2, pop (ds1));

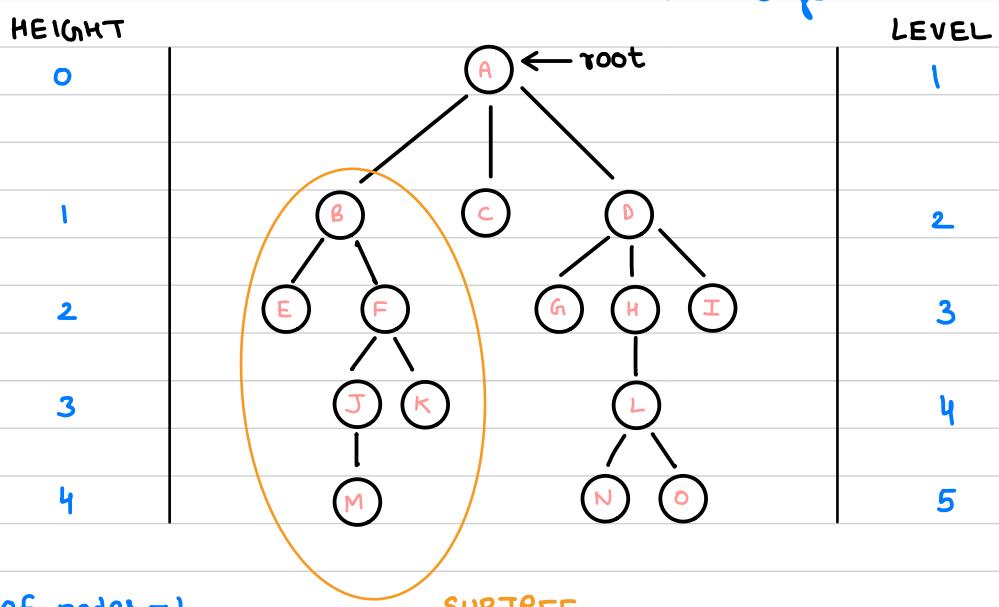
 }

 }
 return pop (ds2);





TREES → Tree is a collection of nodes and edges.



$$\text{No of edges} = \text{No of nodes} - 1$$

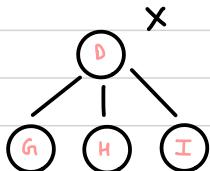
TERMINOLOGY

- (1) Root: The first node on the top
- (2) Parent: A node is a parent to its very next descendants.
- (3) Child
- (4) Sibling: Children of same parent G, H, I
- (5) Descendants: Set of all those nodes which can be reached from a particular node.
E, F, J, K, M are descendants of B.
- (6) Ancestors: For any node, all the nodes along the path from that node to root node
For M, J, F, B, A are ancestors.
- (7) Degree of a node: No of children it is having L-2
- (8) Internal Nodes / Non-Leaf Nodes / Non-Terminal Nodes: Nodes with degree > 0
- External Nodes / Leaf-Nodes / Terminal Nodes: Nodes with degree 0
- (9) Levels (10) Height
- (10) Forest: Collection of trees.

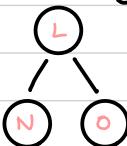
BINARY TREE

degree {2}

children - 0, 1, 2



BINARY TREE ✓



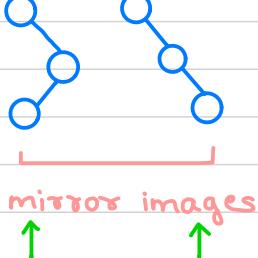
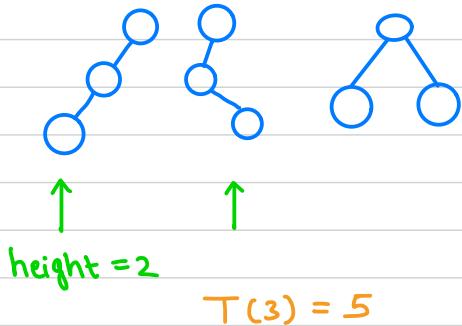
NUMBER OF BINARY TREES USING N NODES

(1) Unlabelled Nodes

(2) Labelled Nodes

$$n = 3$$

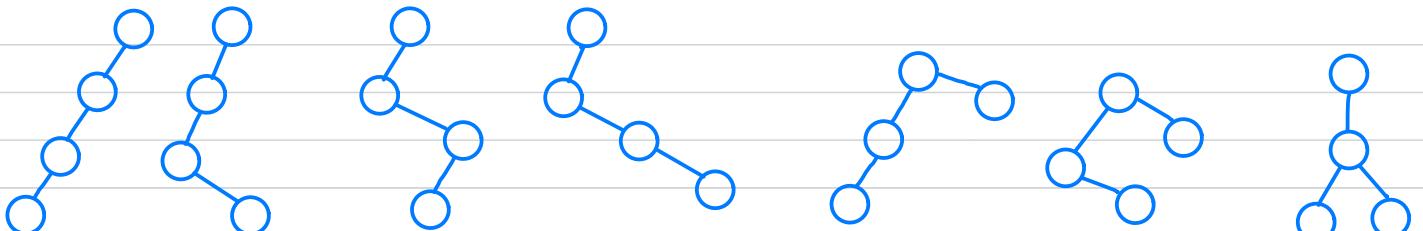
No of trees with maximum height = $4 = 2^2$
(2)



$$n = 4$$



No of trees with maximum height = $8 = 2^3$



$$T(4) = 14$$

+ Other 7 mirror images

$$T(n) = \frac{2^n C_n}{n+1}$$

Catalan Number

No of trees with maximum height = 2^{n-1}

2nd Method for finding Catalan Number

n	0	1	2	3	4	5	6
$T(n) = \frac{2^n C_n}{n+1}$	1	1	2	5	14	42	

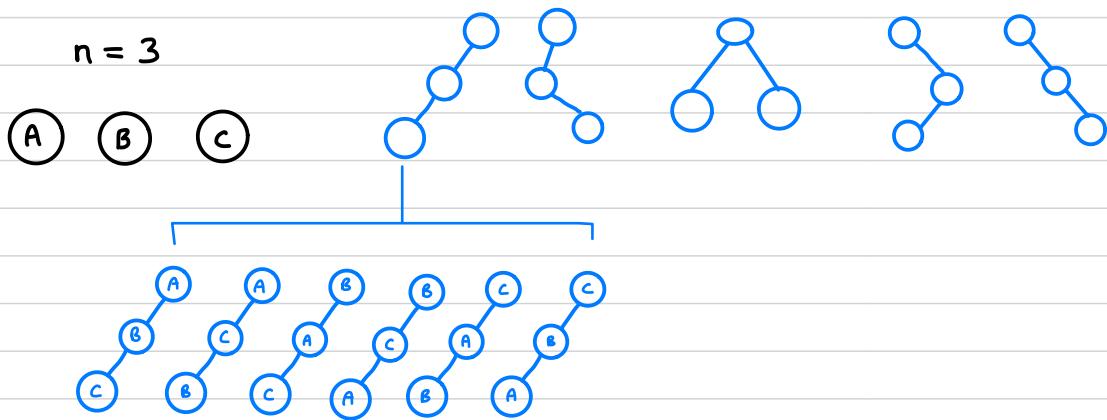
$$T(6) = 1 \times 42 + 1 \times 14 + 2 \times 5 + 5 \times 2 + 14 \times 1 + 42 \times 1$$

$$T(6) = T(0) \times T(5) + T(1) \times T(4) + T(2) \times T(3) + T(3) \times T(2) + T(4) \times T(1) + T(5) \times T(0)$$

$$= 132$$

$$T(n) = \sum_{i=1}^n T(i-1)^* T(n-i)$$

(2) Labelled Nodes

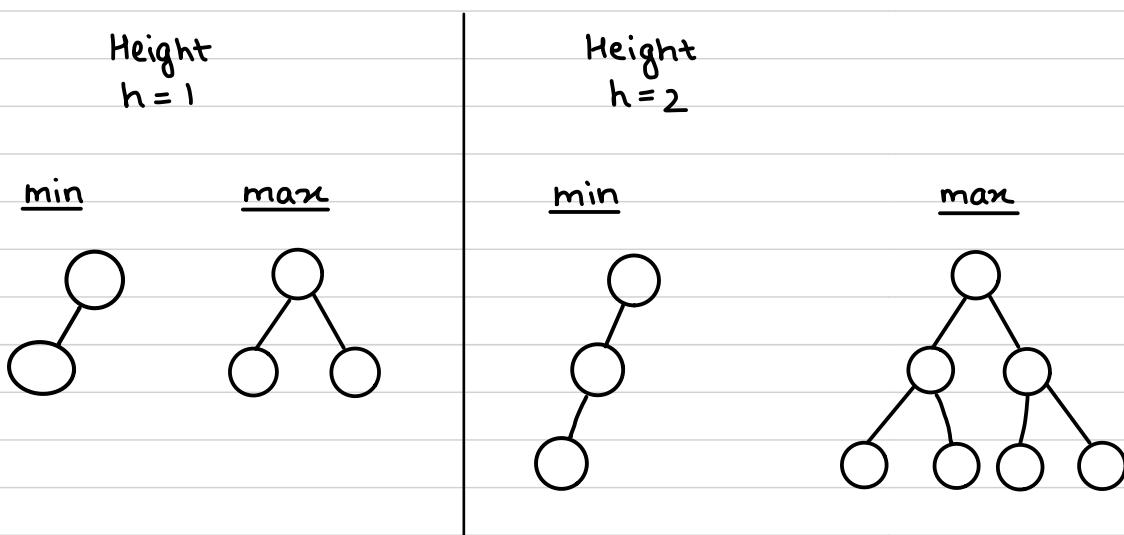


$$6 = 3^1$$

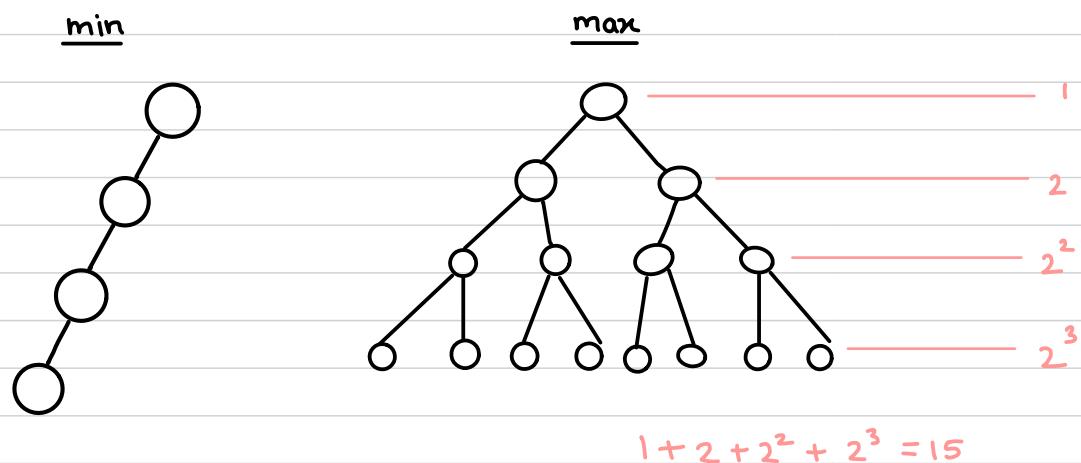
$$T(n) = \frac{2^n C_n}{n+1} * n!$$

↑ ↑
Shapes Filling

Height vs Node



Height $h = 3$



Minimum Number of Nodes

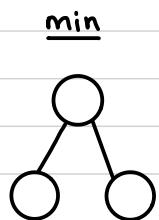
$$\text{min nodes} = h+1$$

Maximum Number of Nodes

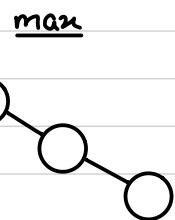
$$a + ar + ar^2 + ar^3 + ar^k = \frac{a(r^{k+1}-1)}{r-1}$$

$$1 + 2 + 2^2 + 2^3 + \dots + 2^h = \frac{1 \cdot (2^{h+1}-1)}{2-1} = 2^{h+1}-1$$

Nodes $n = 3$



$$h = 1$$



$$h = 2$$

If nodes are given

(1) Max height $h = n - 1$

Can be obtained by previous formula

(2) For minimum height

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

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

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

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

Height of a binary tree

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

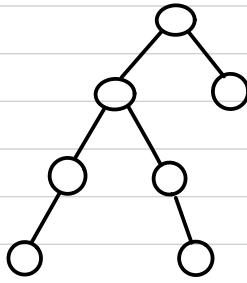
$O(\log n)$

$O(n)$

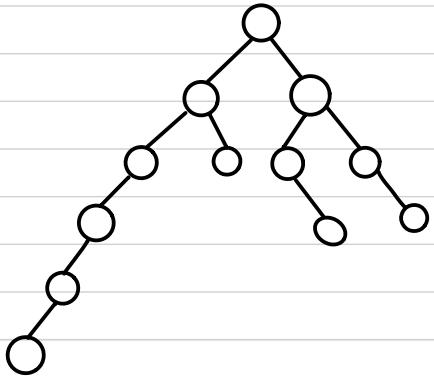
Number of Nodes in a binary tree

$$h+1 \leq n \leq 2^{h+1} - 1$$

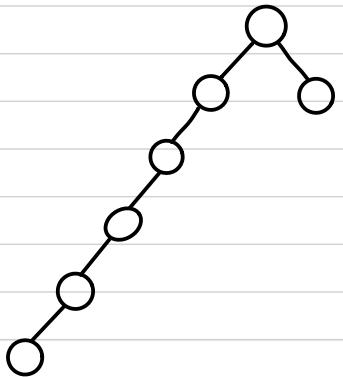
INTERNAL NODES VS EXTERNAL NODES IN A BINARY TREE



$$\begin{aligned} \text{deg}(2) &= 2 \\ \text{deg}(1) &= 2 \\ \text{deg}(0) &= 3 \end{aligned}$$



$$\begin{aligned} \text{deg}(2) &= 3 \\ \text{deg}(1) &= 5 \\ \text{deg}(0) &= 4 \end{aligned}$$



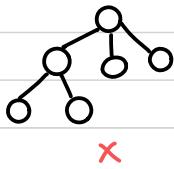
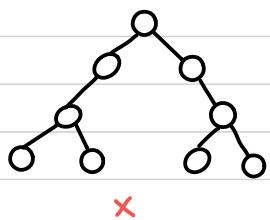
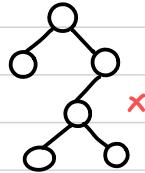
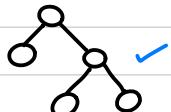
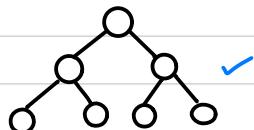
$$\begin{aligned} \text{deg}(2) &= 1 \\ \text{deg}(1) &= 4 \\ \text{deg}(0) &= 2 \end{aligned}$$

$$\boxed{\text{deg}(0) = \text{deg}(2) + 1}$$

True in binary tree

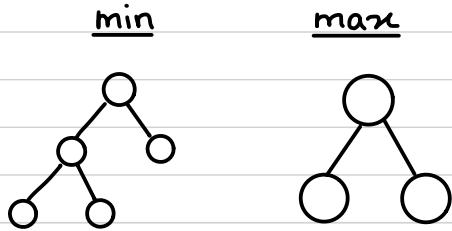
STRICT / PROPER / COMPLETE BINARY TREES

$$\text{deg} = \{0, 1, 2\}$$

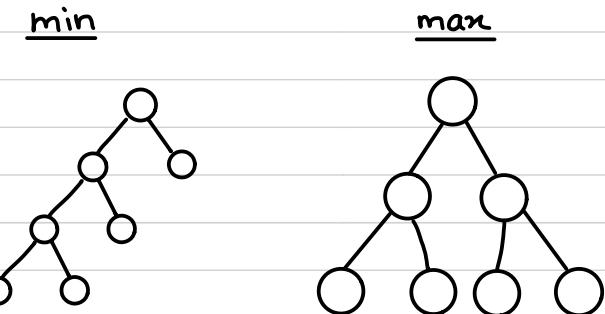


Height vs Node (Strict Binary Tree)

Height
 $h=1$



Height
 $h=2$

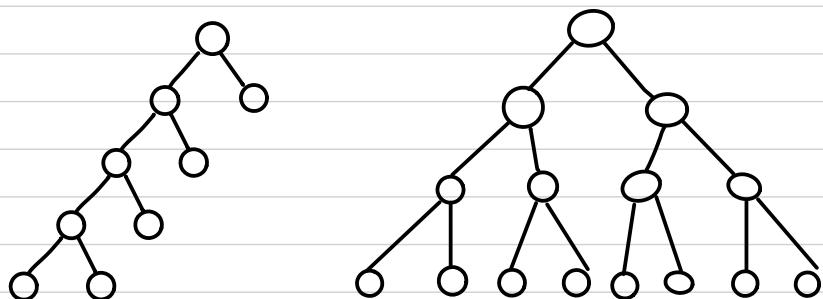


Height $h=3$

min

max

Only minimum number changed



If height 'h' is given

If 'n' nodes are given

$$\text{Min Nodes } n = 2h+1$$

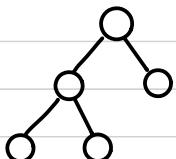
$$\text{Max Nodes } n = 2^{h+1}-1$$

$$\text{Minimum height } h = \log_2(n+1)-1$$

$$\text{Max height } h = \frac{n-1}{2}$$

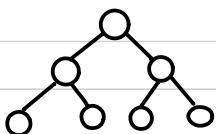
$$\log_2(n+1)-1 \leq h \leq \frac{n-1}{2}$$

INTERNAL NODES VS EXTERNAL NODES IN A BINARY TREE (STRICT)



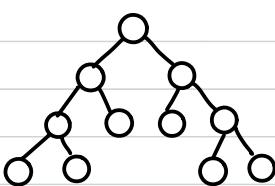
$$i=2$$

$$e=3$$



$$i=3$$

$$e=4$$



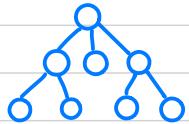
$$i=5$$

$$e=6$$

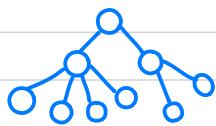
$$e=i+1$$

n-ary Trees

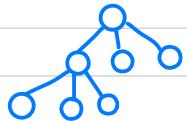
3-ary Tree {0,1,2,3}



4-ary Tree {0,1,2,3,4}



Strict 3-ary Tree {0,1,3}

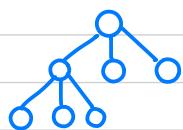


Strict n-ary trees

$h = 2$

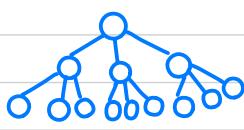
If height ' h ' is given

min



$$\begin{aligned}n &= 3 + 3 + 1 \\n &= 2 \times 3 + 1 \\i &= 2 \\e &= 5\end{aligned}$$

max



$$\begin{aligned}j &= 5 \\e &= 9\end{aligned}$$

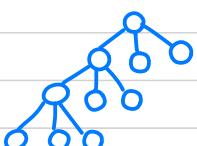
$$\begin{aligned}\text{Min Nodes } n &= nh + 1 \\ \text{Max Nodes } n &= \frac{m^{h+1} - 1}{m - 1}\end{aligned}$$

If 'n' nodes are given

$$\begin{aligned}\text{Minimum height } h &= \log_m[n(m-1) + 1] - 1 \\ \text{Max height } h &= \frac{n-1}{m}\end{aligned}$$

$h = 3$

min



$$\begin{aligned}i &= 3 \\e &= 7\end{aligned}$$

max



$$\begin{aligned}\Rightarrow 1 + 3 + 3^2 + 3^3 \\ \Rightarrow 1 + n + n^2 + n^3 \\ \Rightarrow \frac{(n^{h+1} - 1)}{n - 1}\end{aligned}$$

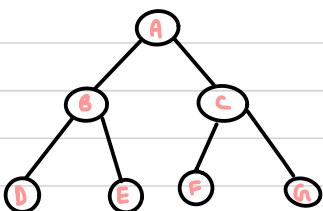
$$e = 2^i + 1$$

$$e = (n-1)i + 1$$

REPRESENTATION OF BINARY TREE

1. Array Representation
2. Linked Representation

1. Array Representation



A	B	C	D	E	F	G
1	2	3	4	5	6	7

Element	Index	L. child	R. child
A	1	2	3
B	2	4	5
C	3	6	7
	i	$2^* i$	$2^* i + 1$

Element \rightarrow i

L Child \rightarrow $2^* i$

R Child \rightarrow $2^* i + 1$

Parent \rightarrow $\lfloor \frac{i}{2} \rfloor$

2. Linked Representation

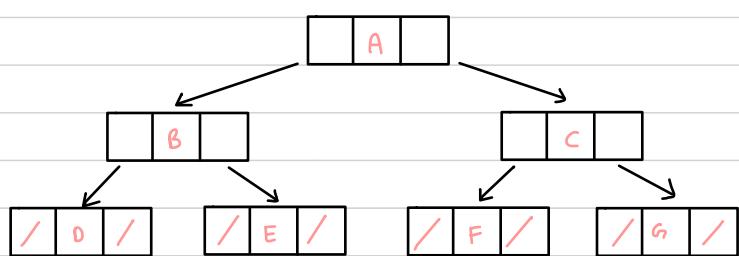
NODE



Struct Node
{

```

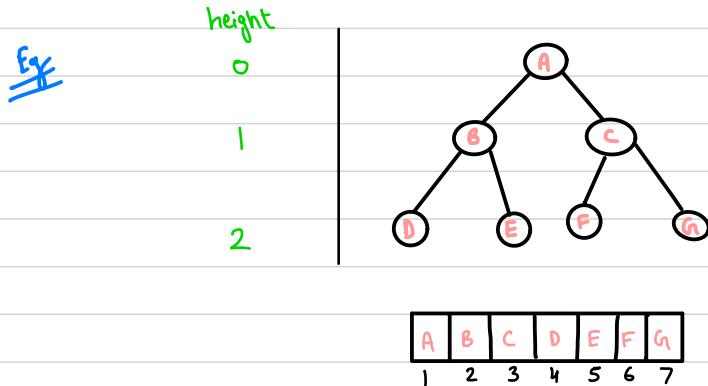
Struct Node *lchild;
int data;
Struct Node *rchild;
}
  
```



$$n = 7$$

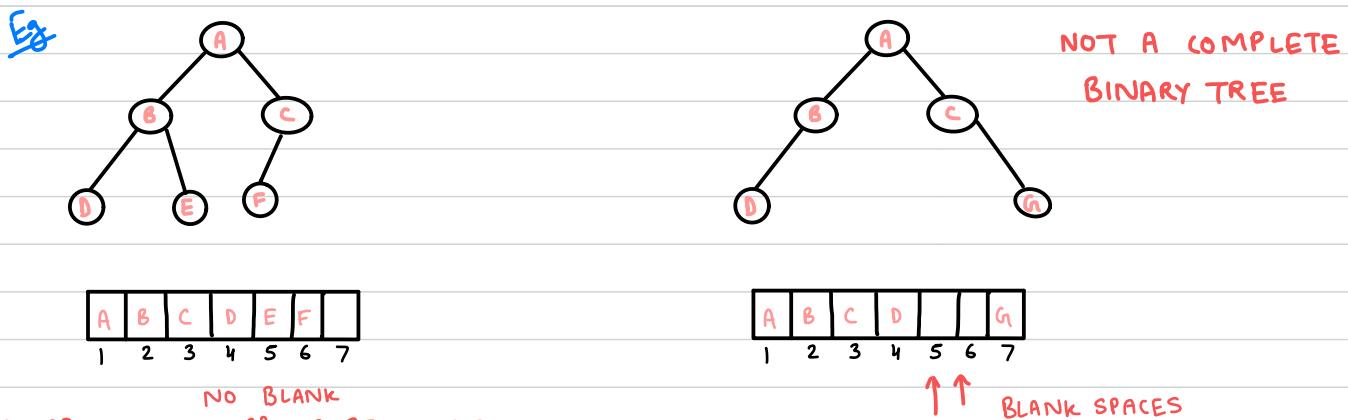
nulls = n + 1

Full Binary Tree: A binary tree of height h having maximum number of nodes.



Complete Binary Tree: A complete binary tree of height h will be a full binary tree upto height $h-1$

A full binary tree is always a complete binary tree.



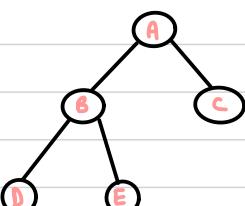
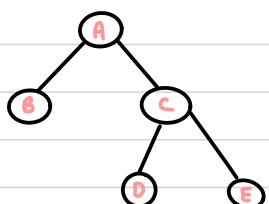
degree
 $\{0, 2\}$

NO BLANK
SPACES BETWEEN ELEMENTS

STRICT VS COMPLETE

↓
Complete

almost
complete



A	B	C		D	E	
1	2	3	4	5	6	7

strict ✓
complete ✗

A	B	C	D	E		
1	2	3	4	5	6	7

strict ✓
complete ✓

TREE TRAVERSALS

Preorder : NLR

Node , Left , Right

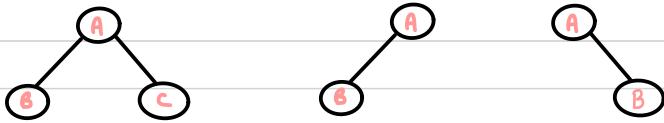
Inorder : LNR

Left , Node , Right

Postorder : LRN

Left , Right , Node

Level Order : level by level



Preorder ABC

Inorder BAC

Postorder BCA

Level ABC

AB

BA

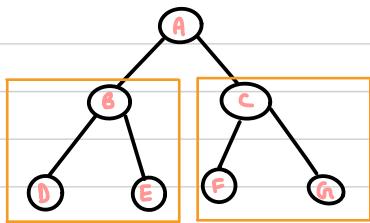
BA

AB

AB

BA

AB



Preorder A , (B,D,E) , (C,F,G)

A,B,D,E,C,F,G

Inorder (D,B,E) , A , (F,C,G)

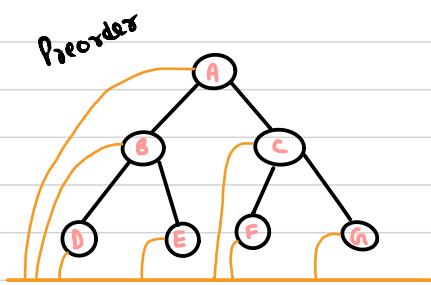
D,B,E,A,F,C,G

Postorder (D,E,B) , (F,G,C) , A

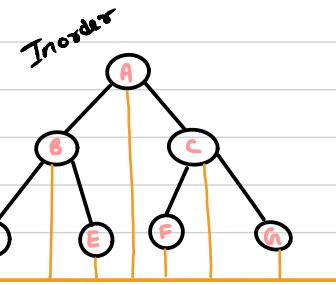
D,E,B,F,G,C,A

Level A,B,C,D,E,F,G

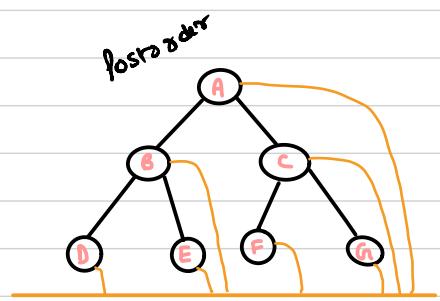
BINARY TREE TRAVERSAL EASY METHOD 1



A, B, D, E, C, F, G

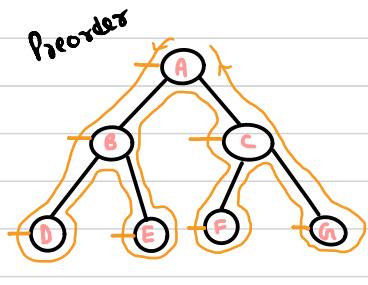


D, B, E, A, F, C, G

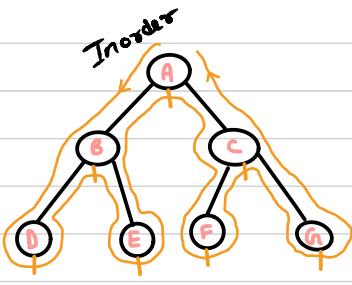


D, E, B, F, G, C, A

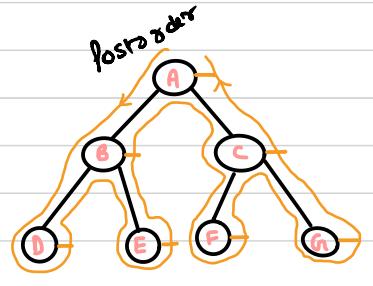
BINARY TREE TRAVERSAL EASY METHOD 2



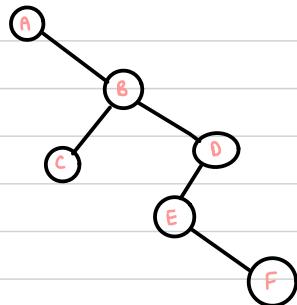
A, B, D, E, C, F, G



D, B, E, A, F, C, G



D, E, B, F, G, C, A



Preorder : A, B, C, D, E, F

Inorder : A, C, B, E, F, D

Postorder: C, F, E, D, B, A

PROGRAM TO CREATE BINARY TREE

```
void create()
```

```
{
```

```
Node *p, *t;  
int x;  
Queue q; initializing a queue  
printf("Enter root value");  
scanf("%d", &x);  
root = malloc(...); initializing root  
root -> data = x;  
root -> lchild = root -> rchild = 0;  
enqueue(root); Storing address of root in queue
```

```
while (!isEmpty(q))
```

```
{
```

```
p = dequeue(dq); take out a value from queue into p  
printf("Enter left child");  
scanf("%d", &x);
```

```
if (x != -1)
```

```
{
```

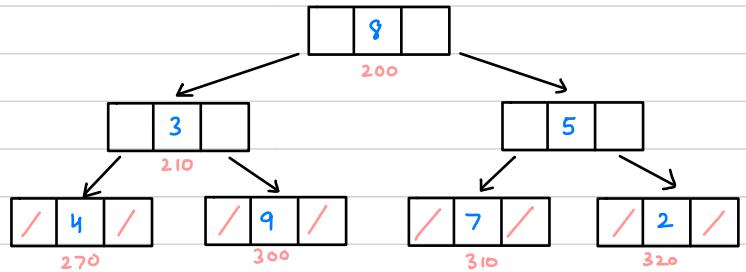
```
t = malloc(...);  
t -> data = x; t -> lchild = t -> rchild = 0;  
p -> lchild = t; p -> rchild = t for right child everything  
enqueue(t); else will be same
```

```
}
```

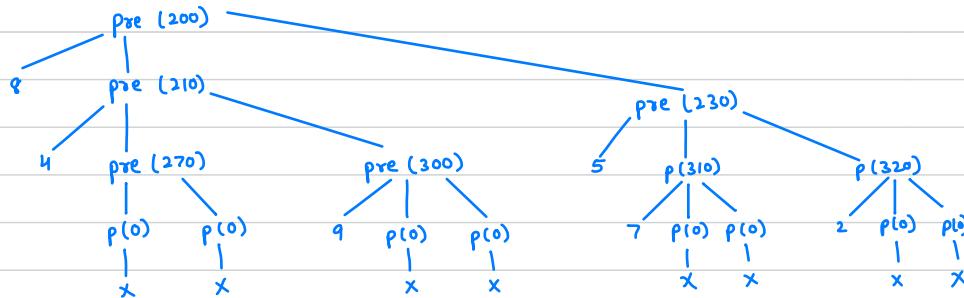
```
}
```

RECURSIVE PREORDER TREE TRAVERSAL

```
void Preorder (Node *t)
{
    if ( t != NULL )
    {
        printf( "%d", t->data );
        Preorder ( t->lchild );
        Preorder ( t->rchild );
    }
}
```



TRACING TREE



ACTIVATION RECORDS
ARE CREATED USING
STACK

RECURSIVE INORDER TREE TRAVERSAL

```
void Inorder ( Node *t )
{
    if ( t != NULL )
    {
        Inorder ( t->lchild );
        printf( "%d", t->data );
        Inorder ( t->rchild );
    }
}
```

RECURSIVE Postorder Tree TRAVERSAL

```
void Postorder ( Node *t )
{
    if ( t != NULL )
    {
        Postorder ( t->lchild );
        Postorder ( t->rchild );
        printf( "%d", t->data );
    }
}
```

ITERATIVE PREORDER TRAVERSAL

```
void Preorder ( Node *t )
{
```

```
    struct stack st;
```

to check for empty stack

```
    while ( t != NULL || !isEmpty ( st ) )
    {
```

```
        if ( t != NULL )
        {
```

```
            printf ( "%d ", t->data );
```

Inorder (Node *t)

push (&t, t); push address of t in stack

t = t->lchild;

```
}
```

```
        else
        {
```

t = pop (&st); pop out address

t = t->rchild; from stack

```
}
```

push (&st, t);

t = t->lchild;

```
}
```

ITERATIVE POSTORDER TRAVERSAL

```
void Inorder ( 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 )
        {
```

push (&st, -temp);

t = (Node *) temp ->rchild;

```
}
```

```
        else
        {
```

printf ("%d ", (Node *) temp ->data);

t = NULL;

```
}
```

```
}
```

```
}
```

LEVEL ORDER TRAVERSAL

```

void Levelorder (Node *p)
{
    Queue q;
    printf("%d", p->data);
    enqueue (&q, p);

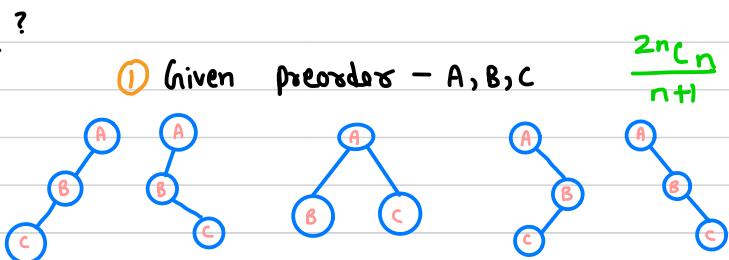
    while( ! isEmpty (q) )
    {
        p = dequeue (&q);
        if (p->lchild)
        {
            printf("%d", p->lchild->data);
            enqueue (&q, p->lchild);
        }

        if (p->rchild)
        {
            printf("%d", p->rchild->data);
            enqueue (&q, p->rchild);
        }
    }
}

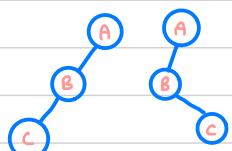
```

CAN WE GENERATE TREE FROM TRAVERSALS ?

$n = 3$



② Given preorders A, B, C more than one posorder C, B, A



CONCLUSION

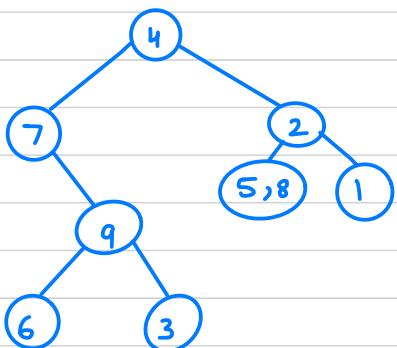
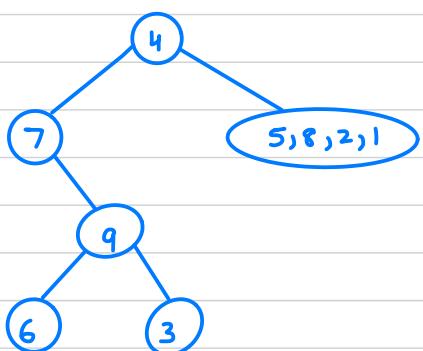
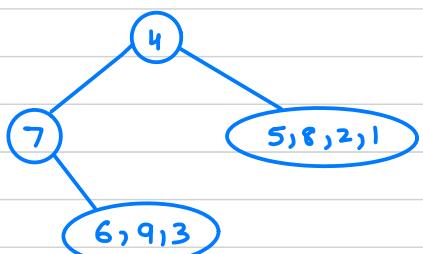
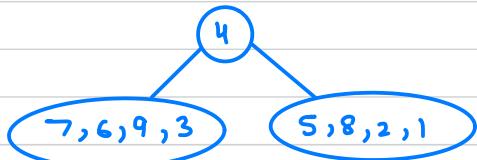
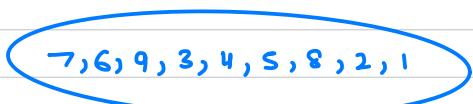
If we are given either preorders, inorder, postorder we cannot generate unique tree.

If we are given both preorder and postorder, then also unique tree cannot be generated

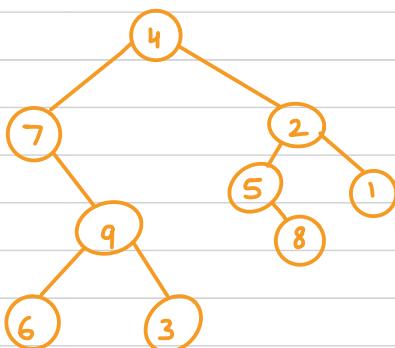
- ① preorder + inorder for generating unique tree
- ② postorder + inorder

Q Preorder - 4, 7, 9, 6, 3, 2, 5, 8, 1
Inorder - 7, 6, 9, 3, 4, 5, 8, 2, 1

Sol



Unique Tree



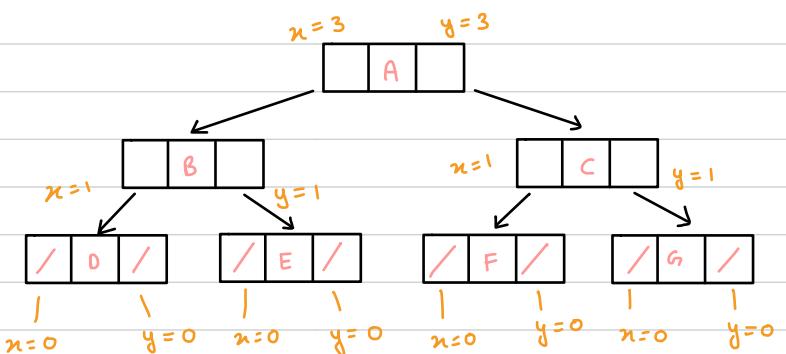
COUNTING NO OF NODES IN A BINARY TREE

7

```

int count ( Node *p )
{
    int x,y;
    if ( p != NULL )
    {
        x = count ( p ->lchild );
        y = count ( p ->rchild );
        return x + y + 1;
    }
    return 0;
}

```



COUNTING LEAF NODES

COUNTING NODES WITH DEGREE 2

```

int count ( Node *p )
{

```

COUNTING NODES WITH DEGREE 2 OR 1

```

    int x,y;
    if ( p == NULL )
        x = count ( p ->lchild );
        y = count ( p ->rchild );

```

COUNTING NODES WITH DEGREE 1

```

        if ( p ->lchild == NULL && p ->rchild == NULL )
            return x + y + 1;

```

```

        else
            return x + y;

```

}

if (p ->lchild != NULL && p ->rchild != NULL)

return 0;

if (p ->lchild != NULL || p ->rchild != NULL)

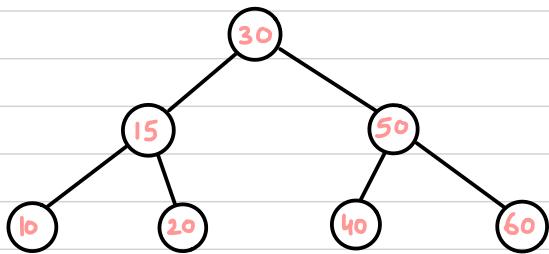
if ((p ->lchild != NULL && p ->rchild == NULL) ||
 (p ->lchild == NULL && p ->rchild != NULL))

XOR

if (p ->lchild != NULL ^ p ->rchild != NULL)



BINARY SEARCH TREE



- Left subtree is smaller
- Right subtree is greater
- Useful for searching (in less number of comparisons)
- No duplicates
- Inorder gives sorted order
- For n number of nodes, catalan number of trees can be generated

$$T(n) = \frac{2^n C_n}{n+1}$$

- BST is represented using linked representation.
- Can also be represented using arrays.

SEARCHING IN A BINARY SEARCH TREE

(Searching takes maximum time same as height of tree) $O(\log n)$

(1) RECURSIVE SEARCH (*tail recursion*)

```

Node * Rsearch(Node *t, int key)
{
    if (t == NULL)           // If element is not
        return NULL;          found
    if (key == t->data)
        return t;
    else if (key < t->data)
        return Rsearch(t->lchild, key);
    else
        return Rsearch(t->rchild, key);
}
  
```

$O(h)$

$\log n \leq h \leq n$

(2) ITERATIVE VERSION $O(\log n)$

```

Node * Search(Node *t, int key)
{
    while (t != NULL)
        if (key == t->data)
            return t;
}
  
```

```

else if (key < t->data)
    t = t->lchild;
else
    t = t->rchild;
}
return NULL;
  
```

INSERTING IN BINARY SEARCH TREE

```
void Insert(Node *t, int key)
{
    Node *r = NULL, *p;

    while (t != NULL)
    {
        if (key == t->data)
            return; // To avoid duplication
        else if (key < t->data)
            t = t->lchild;
        else
            t = t->rchild;
    }

    p = new Node;
    p->data = key;
    p->lchild = p->rchild = NULL;

    if (p->data < r->data)
        r->lchild = p;
    else
        r->rchild = p;
}
```

RECURSIVE INSERT IN BST

```
Node * Insert(Node *p, int key)
{
    if (p == NULL)
    {
        t = new Node;
        t->data = key;
        t->lchild = t->rchild = NULL;
    }

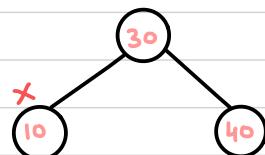
    if (key < p->data)
        p->lchild = insert(p->lchild, key);
    else if (key > p->data)
        p->rchild = insert(p->rchild, key);

    return p;
}
```

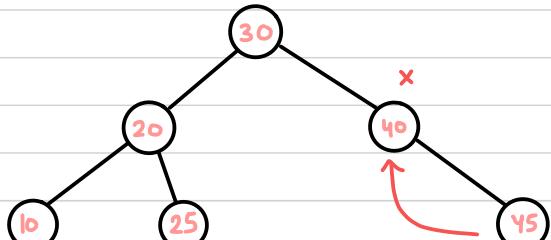
```
void main()
{
    Node *root = NULL;
    root = insert(rroot, 30);
    insert(rroot, 20);
    insert(rroot, 25);
}
```

DELETING FROM BST

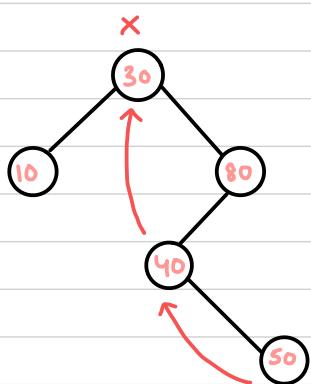
CASE 1 :



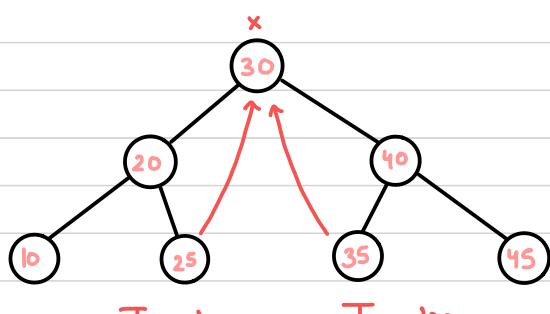
CASE 2 :



CASE 4 :



CASE 3 :



// Multiple changes to be made

```
int Height ( struct Node *p )
{
    int x, y;
    if ( p == NULL )
        return 0;
    x = Height ( p ->lchild );
    y = Height ( p ->rchild );
    return x>y? x+1: y+1;
}
```

```
struct Node * InPre( struct Node *p )
{
    while ( p && p ->lchild != NULL )
        p = p ->lchild;
    return p;
}
```

```
struct Node * InSucc( struct Node *p )
{
    while ( p && p ->rchild != NULL )
        p = p ->rchild;
    return p;
}
```

```

Struct Node * Delete( struct Node *p , int key)
{
    struct Node *q;

    if (p == NULL)
        return NULL;

    if ((p->lchild == NULL) && (p->rchild == NULL)) // Leaf Node
    {
        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 = InPre(p->lchild);
            p->data = q->data;
            p->lchild = Delete(p->lchild, q->data);
        }
        else
        {
            q = InSucc(p->rchild);
            p->data = q->data;
            p->rchild = Delete(p->rchild, q->data);
        }
    }
    return p;
}

```

GENERATING BINARY SEARCH TREE

Void Createpre(int pre[], int n) O(n)

{ Stack st;

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] < stacktop(stk) → data)

{

t = new Node;

t → data = pre[i++];

t → lchild = t → rchild = NULL;

p → rchild = t;

p = t;

}

else

{

p = pop(↓stk);

}

}

}

BST can be generated using

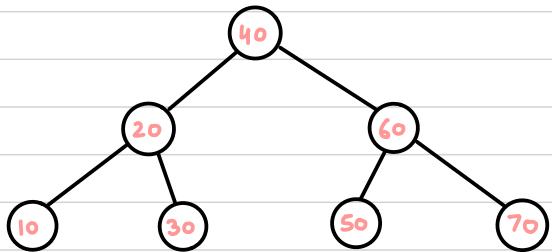
• Preorder + Inorder

• Postorder + Inorder

Sorted preorder of BST gives its Inorder
/postorder

DRAWBACK OF BINARY SEARCH TREE

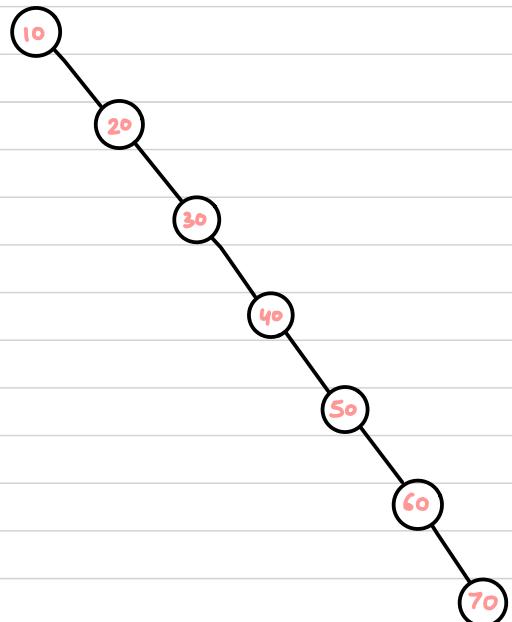
KEYS: 40, 20, 30, 60, 50, 10, 70



0
1
2

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

KEYS: 10, 20, 30, 40, 50, 60, 70



0
1
2
3
4
5
6

$$h = n-1$$
$$O(n)$$

There is no control over height of binary search tree.

It only depends on user's input or sequence of insertion

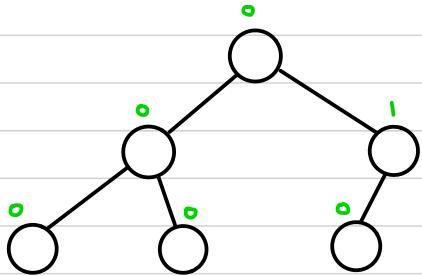


AVL TREES

No of edges

Height Balanced Binary Search Trees

Balance factor = Height of left subtree - Height of right subtree
 $\{-1, 0, 1\}$ → Balanced Tree



Balanced Tree

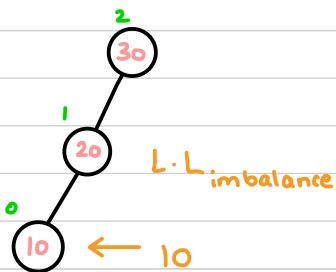
Rotations for insertion in AVL trees

- (1) LL Rotation] Single Rotation
- (2) RR Rotation
- (3) LR Rotation] Double Rotation
- (4) RL Rotation

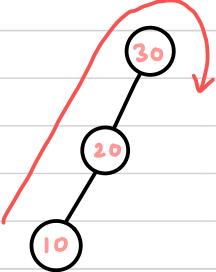
INITIAL



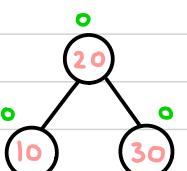
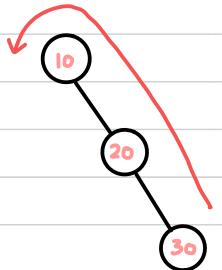
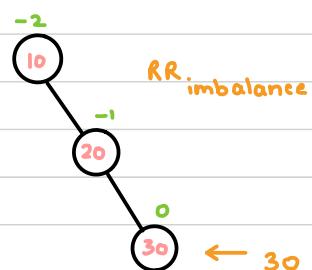
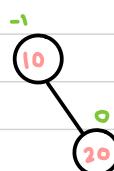
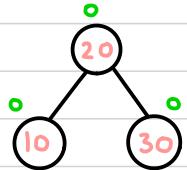
AFTER INSERTION

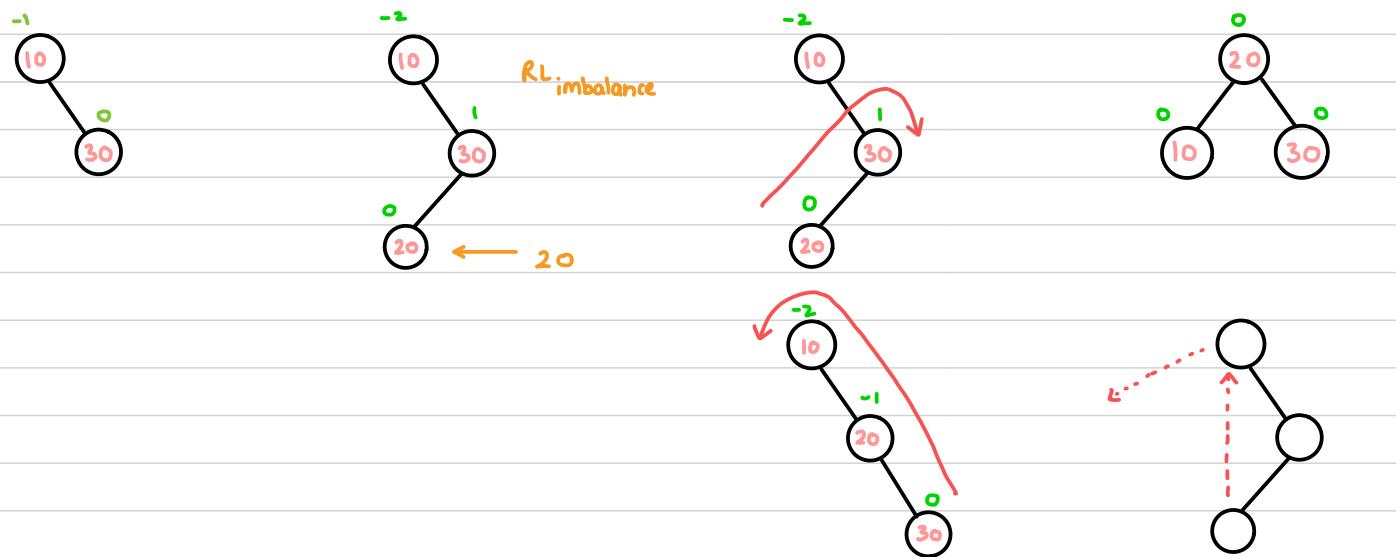
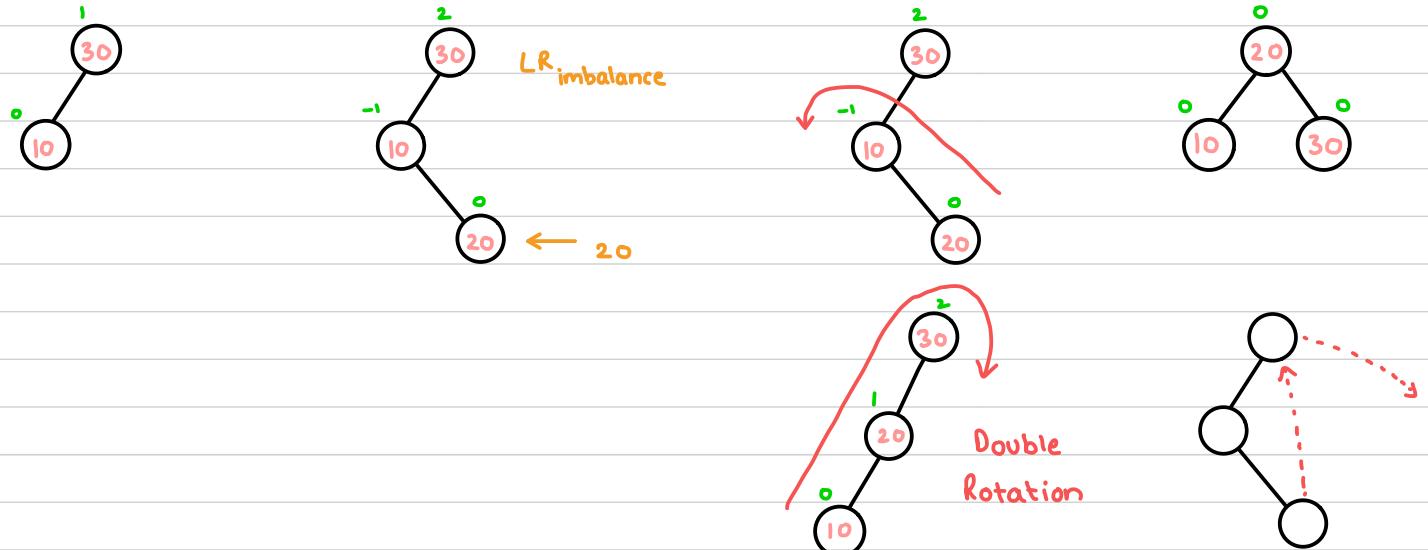


PERFORM ROTATION

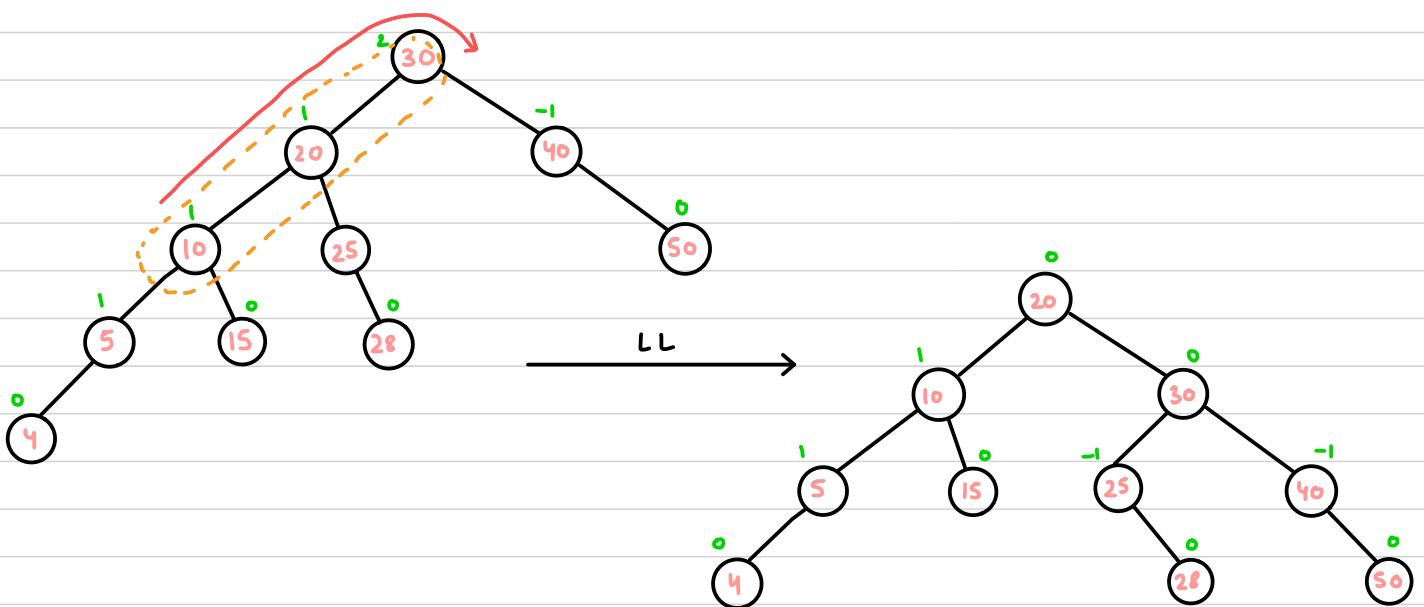


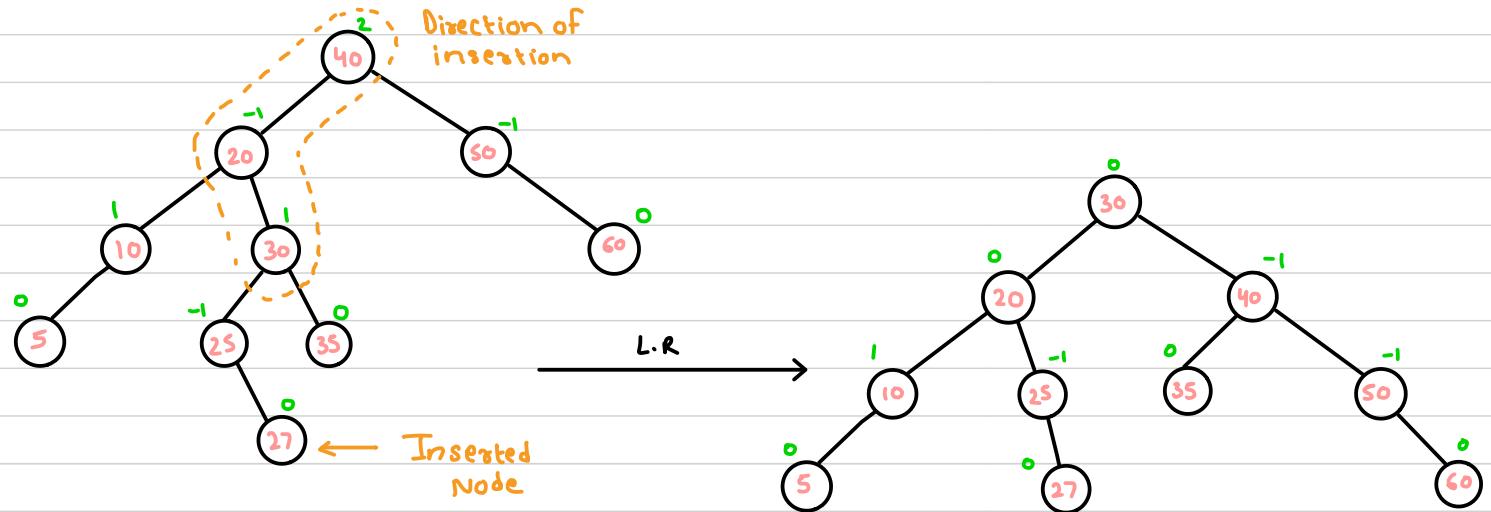
AFTER ROTATION





FORMULA OF ROTATION FOR INSERTION





PROGRAM FOR LL ROTATION

```
Struct Node  
{  
    Struct Node *lchild;  
    int data;  
    int height; // We will set height for  
    Struct Node *rchild; each and every Node  
}* root = NULL;
```

```

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->child) == -1)
    return RRRotation(p);
else if (BalanceFactor(p) == -2 || BalanceFactor(p->child) == 1)
    return RLRotation(p);

return p;
}

int NodeHeight (struct Node *p)
{
    int hl, hr; // height of left subtree (HL), height of right subtree (HR)
    hl = p == p->lchild ? p->lchild->height : 0;
    hr = p == p->rchild ? p->rchild->height : 0;
    → NOT NULL
    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)
```

```

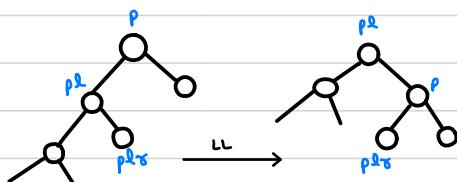
Struct Node * pl = p->lchild;
Struct Node * plr = pl->lchild;

```

```

pl->rchild = p;
p->lchild = plr;
p->height = NodeHeight (p);
pl->height = NodeHeight (plr);

```



```

if(zroot == p) // If rotation was performed on
    zroot = pl; root node, zroot needs to be
    updated.

```

```
return pl;
```

```
}
```

```
Struct Node* LRRotation(Struct Node* p)
```

```
{
```

```
Struct Node* pl = p->lchild;
```

```
Struct Node* plx = pl->rchild;
```

```
pl->rchild = plx->lchild;
```

```
p->lchild = plx->rchild;
```

```
plx->lchild = pl;
```

```
plx->rchild = p;
```

```
pl->height = NodeHeight(pl);
```

```
p->height = NodeHeight(p);
```

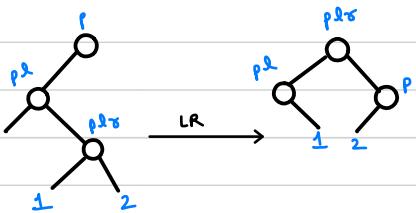
```
plx->height = NodeHeight(plx);
```

```
if(zroot == p)
```

```
zroot = plx;
```

```
return plx; // New zroot;
```

```
}
```



DELETION FROM AVL TREES WITH ROTATION

1. L 1 Rotation

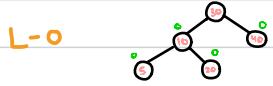
2. L -1 Rotation

3. L 0 Rotation

4. R 1 Rotation

5. R -1 Rotation

6. R 0 Rotation



// Other three will be
mirror images

HEIGHT VS NODES OF AVL TREES

If height is given:

- Max nodes $n = 2^h - 1$ // Not $2^{h+1} - 1$ because height is starting from 1.
- Min nodes $n = \text{Look in table}$

$$h=1$$

$$n=1$$



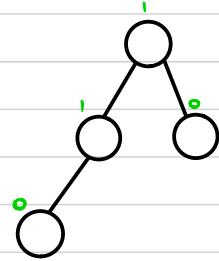
$$h=2$$

$$n=2$$



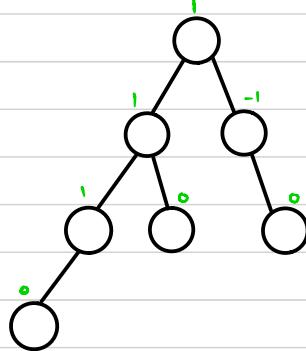
$$h=3$$

$$n=4$$



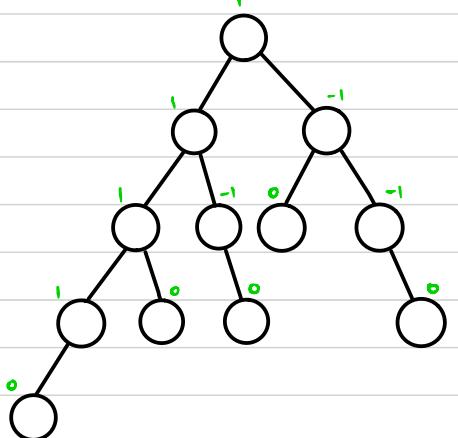
$$h=4$$

$$n=7$$



$$h=5$$

$$n=12$$



h	1	2	3	4	5	6	7
n	1	2	4	7	12	20	33

$\downarrow + \downarrow + 1 \uparrow$

$$N(h) = \begin{cases} 0 & h=0 \\ 1 & h=1 \\ N(h-2) + N(h-1) + 1 & \text{otherwise} \end{cases}$$

// formula same as Fibonacci series

\hookrightarrow balanced series

If 'N' Nodes are given find:

- Min height = $\log_2(n+1)$
- Max height = Look in table

Foreg for 13 nodes $h=5$ // Look from node towards height
19 nodes $h=5$



SEARCH TREES

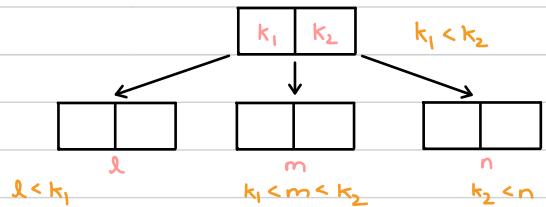
2 - 3 TREES

↗ degree

- Multiway Search Tree (M Way Search Tree)
- Degree 3 (2-3 trees are Multiway Search Trees with degree 3)
- B Trees (These are height Balanced Search Trees)
- Rules

- All leaf nodes at same level
- Every node must have $\lceil \frac{n}{2} \rceil$ children

$$\lceil \frac{3}{2} \rceil = 2$$



- Cannot have duplicates

CREATION OF 2-3 TREE

KEYS : 20 , 30 , 40 , 50 , 60 , 10 , 15 , 70 , 80 , 90

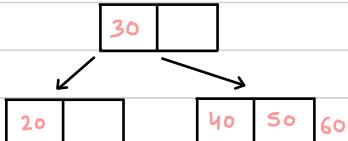
20 , 30



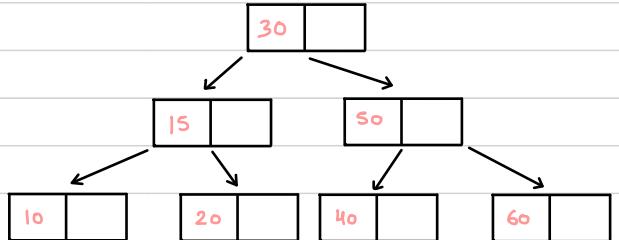
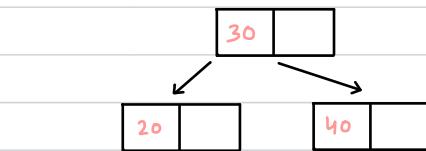
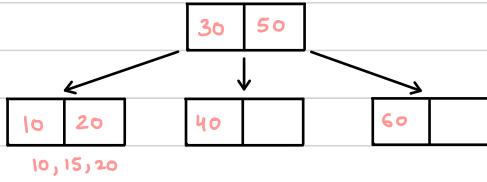
40

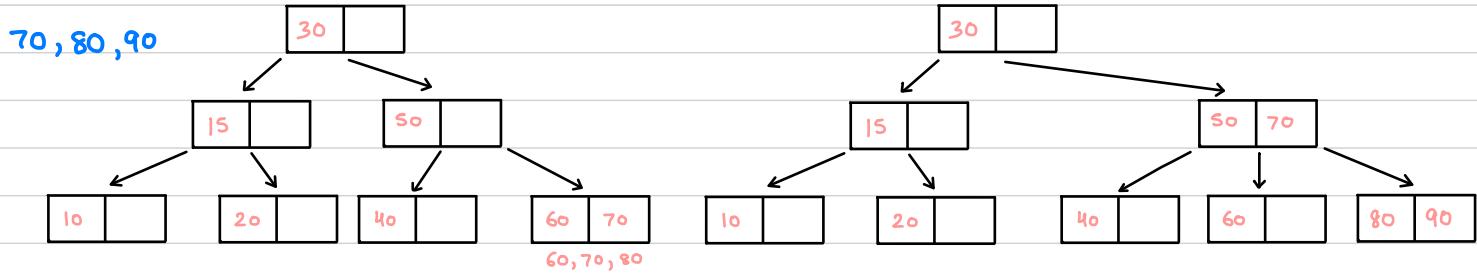


50, 60



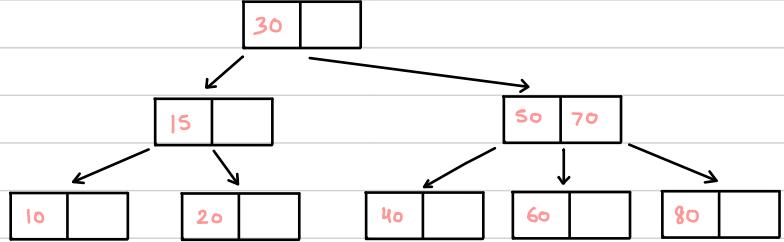
10 , 15



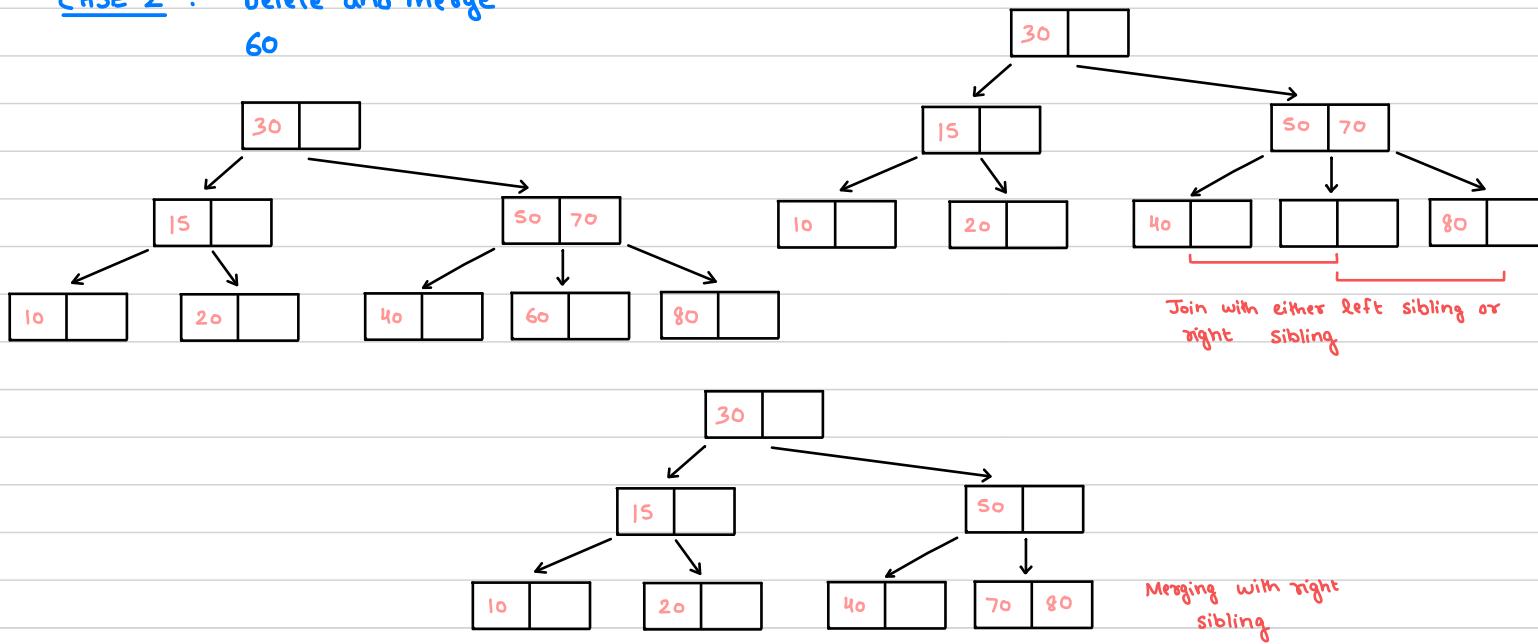


DELETING FROM 2-3 TREE

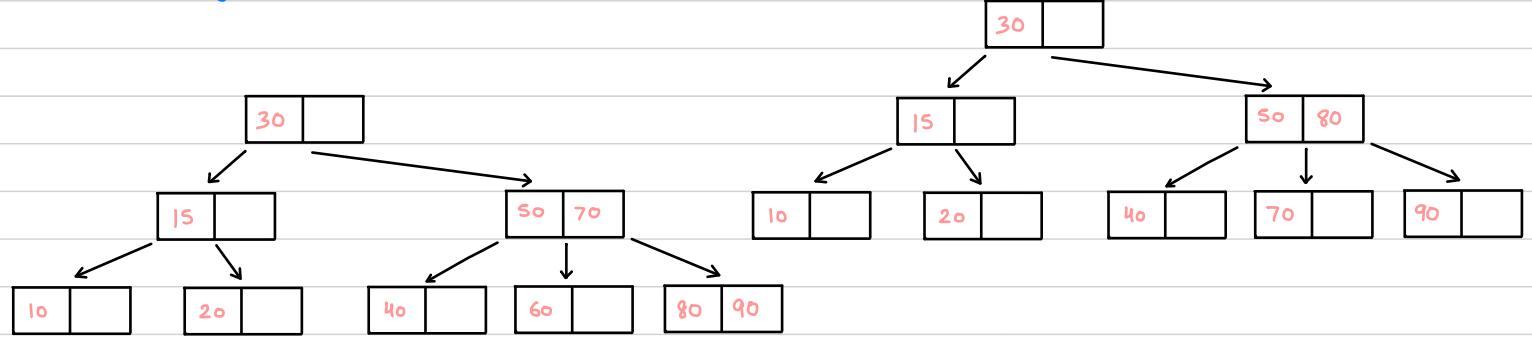
CASE 1 : Simply Delete
90



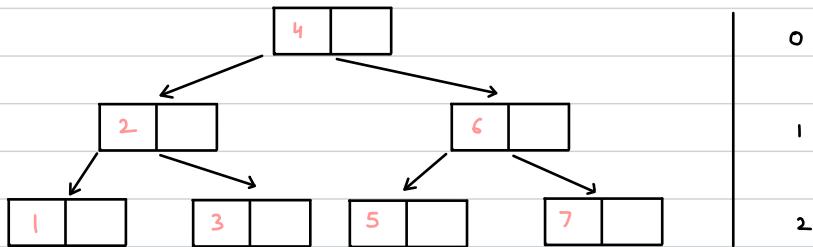
CASE 2 : Delete and merge
60



CASE 3 : Borrow
60



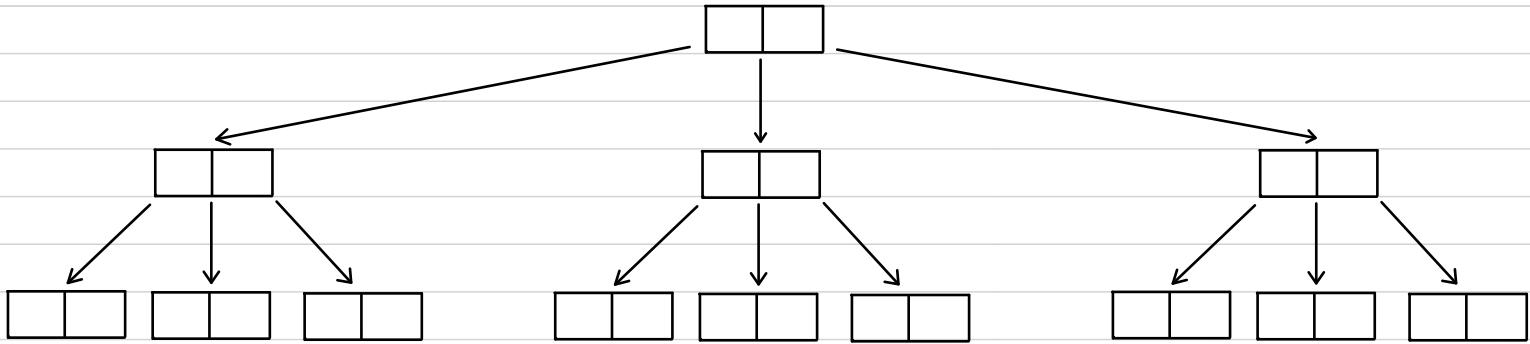
ANALYSIS



Tree with min nodes
for given height

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

$$\text{Max } h = \log_2(n-1) - 1 \\ = O(\log_2 n)$$



$$\text{Max } n = 1 + 3 + 3^2 \dots$$

$$= \frac{3^{h+1} - 1}{3 - 1}$$

$$\text{Min } h = \log_3[n(3-1) + 1] - 1$$

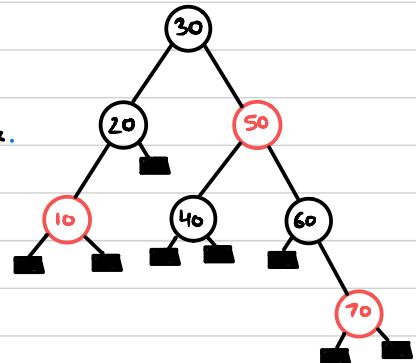
$$O(\log_3 n)$$

Minimum as well as maximum height is $\log n$

These trees are used for DBMS softwares
because a node can have more than one value

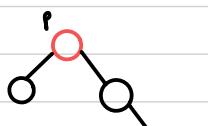
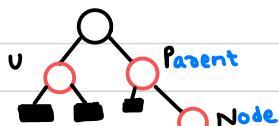
RED BLACK TREE

- It is a height balanced Binary Search Tree, similar to 2-3-4 tree.
- Every node is either Red or Black.
- Root of a Tree is Black
- NULL is also Black.
- Number of Blacks on paths from Root to leaf are same.
- No 2 consecutive Red, Parent and children of red are Black.
- New inserted Node is Red.
- Height in $\log n \leq h \leq 2\log n$



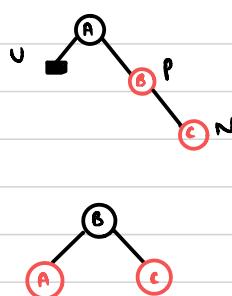
CREATION OF RED BLACK TREE

Uncle is Red (for Node)



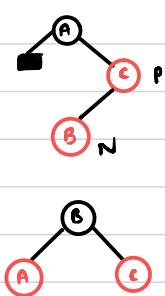
RECOLOURING

Zig - Zig (LL/RR)



Uncle is Black

Zig - Zag (RL/LR)



ROTATION

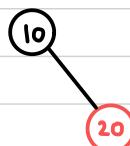
KEYS : 10 , 20 , 30 , 50 , 40 , 60 , 70 , 80 , 4 , 8

INSERT

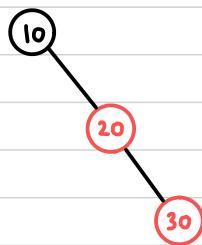
10



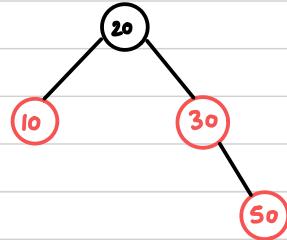
20



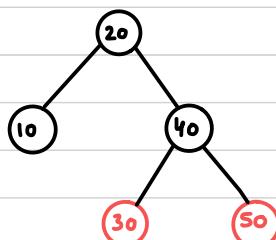
30



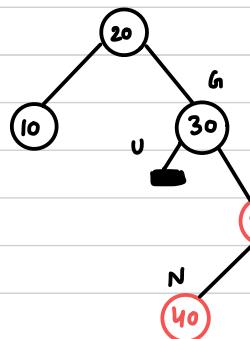
50



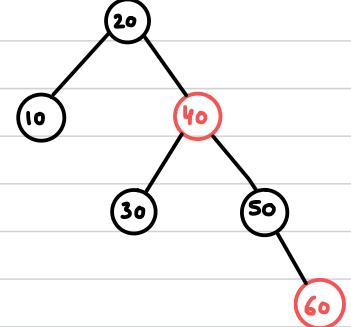
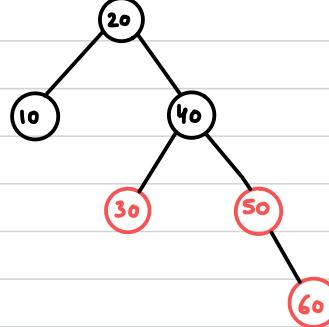
Root must be black



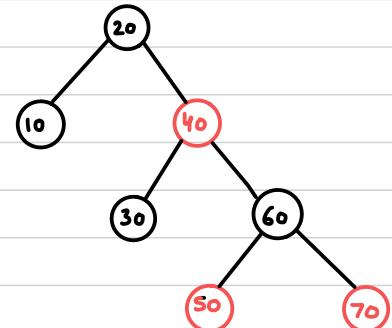
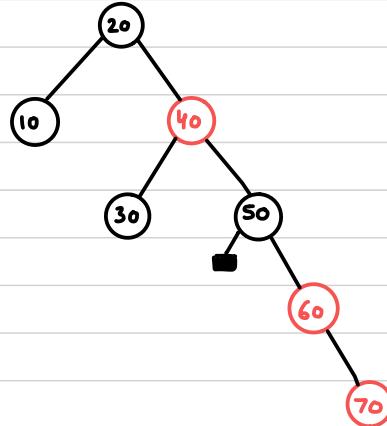
40



60

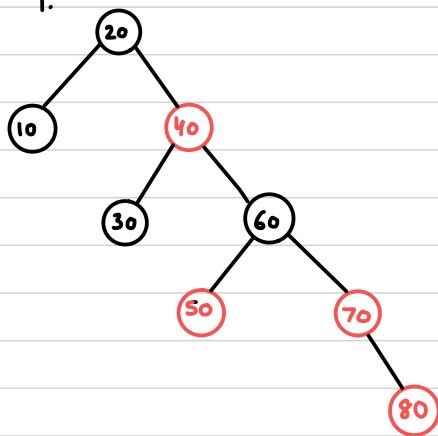


70

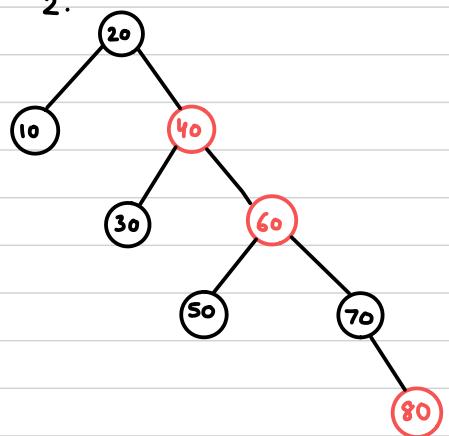


80

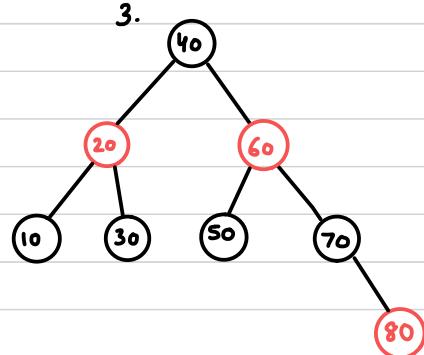
1.



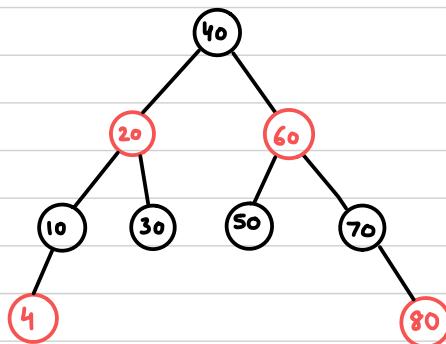
2.



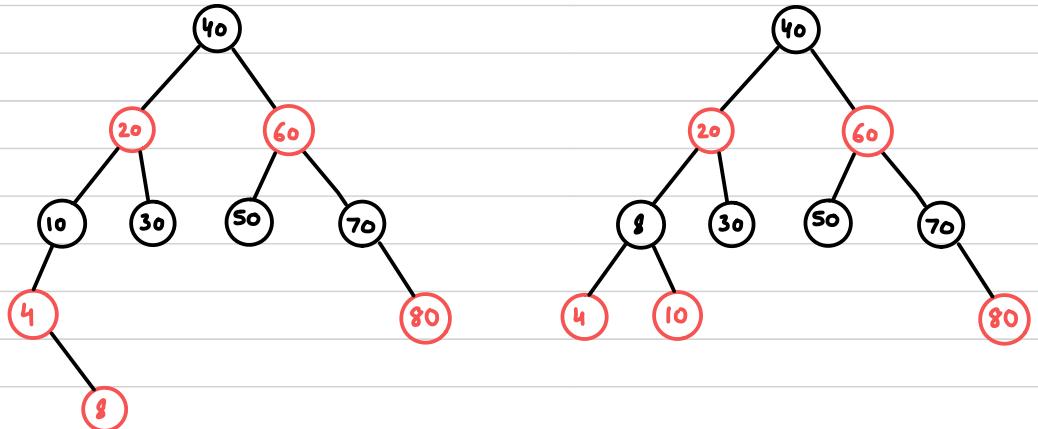
3.



4

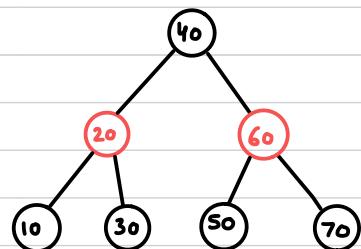
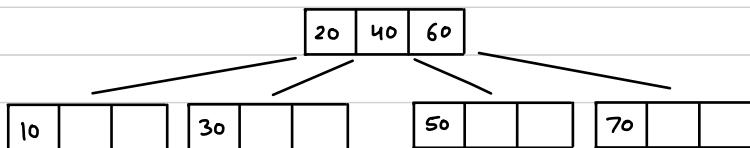
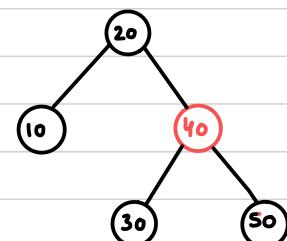
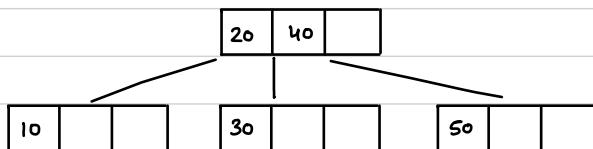
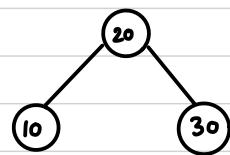
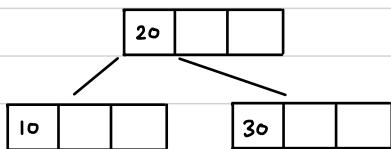


5



height only red/black = $\log n$
height red and black = $2\log n$

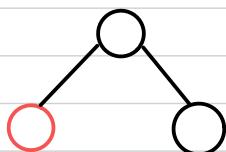
2-3-4 TREES VS RED BLACK TREES



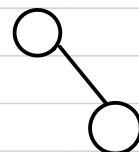
RED BLACK TREE DELETION CASES

CASE 1 : Deleted Node is Red Node

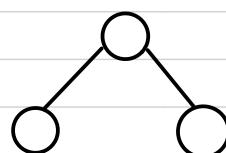
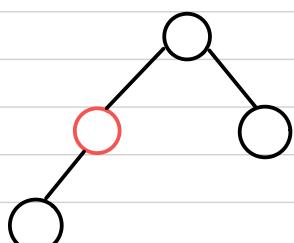
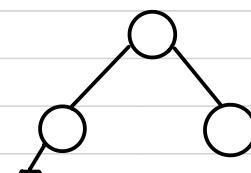
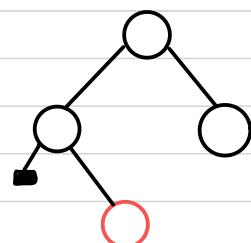
Before Deletion



After Deletion

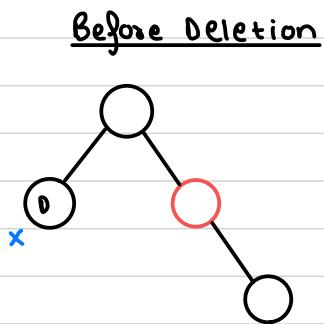


Simply delete as it is a leaf node and red

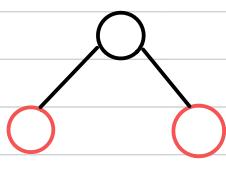


Simply delete because if red node is deleted, the path of black nodes remains unchanged

CASE 2 : Node is black and sibling is red

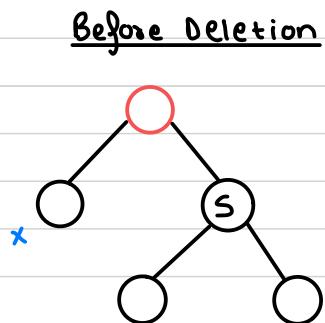


After Deletion

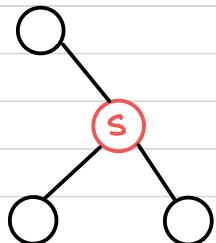


Perform rotation

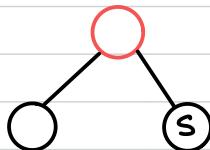
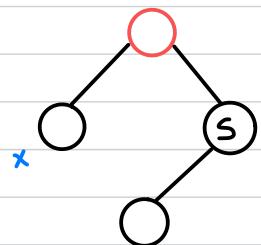
CASE 3 : Node is black and sibling is also black



After Deletion



Change sibling to
red and parent to black
Recolour



Perform rotation

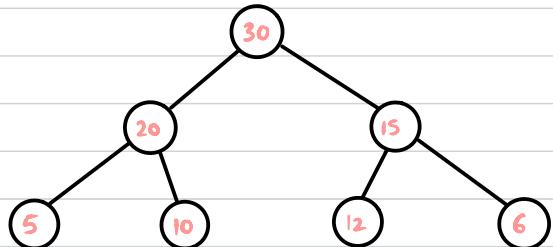


- what is a heap?
- Insert in a heap
- Deleting from heap
- Heap Sort
- Heapify
- Priority Queues

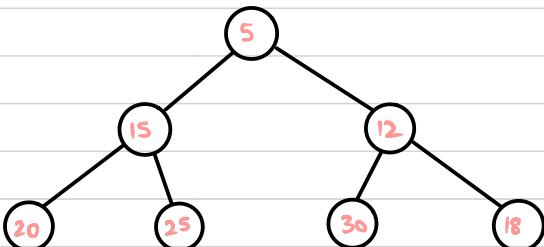
HEAP

It is a complete binary tree

Max Heap



Min Heap



30	20	15	5	10	12	6			
1	2	3	4	5	6	7	8	9	10

- For complete binary tree, there should not be any free space between the elements of array

Node at index i

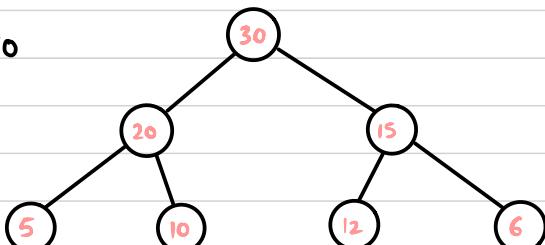
Left child at index 2^*i

Right child at index $2^*i + 1$

- Every node should have an element greater than or equal to all its descendants (duplicates can be there)

INSERTING IN HEAP

Element to insert = 40



30	20	15	5	10	12	6			
1	2	3	4	5	6	7	8	9	10

Step 1: Insert element at next free index in array

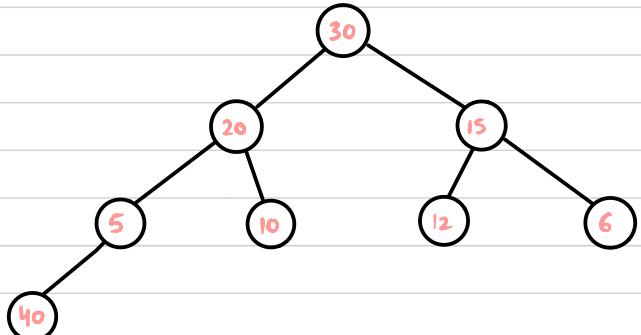
30	20	15	5	10	12	6	40		
1	2	3	4	5	6	7	8	9	10



Step 2: Insert the element in the tree

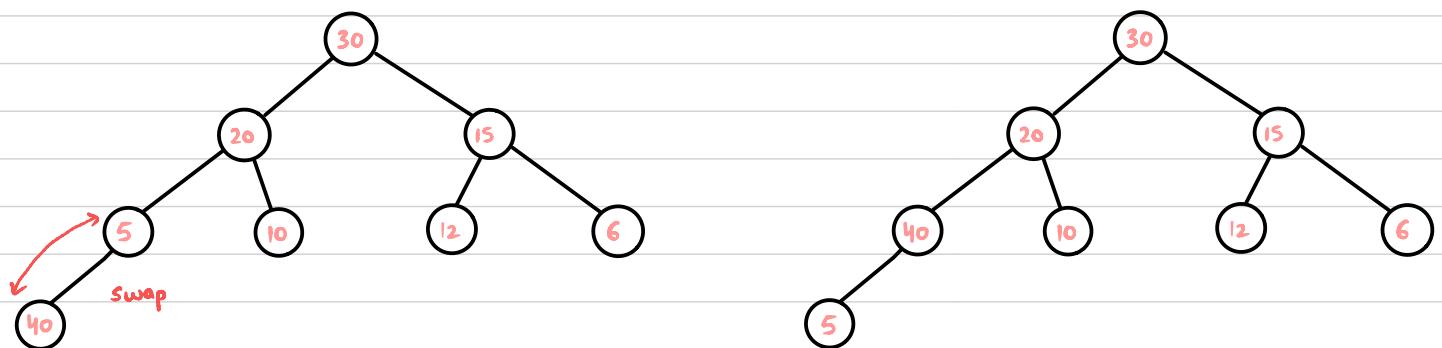
$$\text{Parent of } 40 = \frac{8}{2} = 4$$

30	20	15	5	10	12	6	40		
1	2	3	4	5	6	7	8	9	10

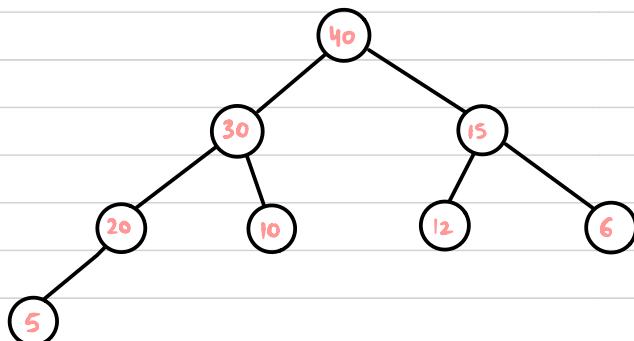


Step 3: Now tree does not satisfy max heap condition

Compare 40 with its parent/ancestor, if node is found greater than ancestor, Swap.



Repeat until condition is satisfied



Check this condition in array also. Compare node with parent ($\frac{i}{2}$), if found greater than parent, replace

40	30	15	20	10	12	6	5		
1	2	3	4	5	6	7	8	9	10

PROGRAM

```
void Insert (int A[], int n)      O(logn)
{
    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()  O(nlogn)   1 element - logn
{
    int A[] = {0, 10, 20, 30, 25, 5, 40, 35};
    int i;
    for (i=2; i<=7; i++)  ← because first element is the heap and next element onwards,
                           insertion in heap is done.
        Insert (A, i);
}
```

Heapify : Faster method to implement creation of binary heap

← Separate Topics
↓

Element Priority

Elements → 6, 8, 3, 10, 15, 2, 9, 17, 5, 8

- Element is itself a priority
- Smaller number higher priority

1. Insert in same order O(1) O(logn)

Delete max priority by searching it O(n) O(logn)

] After implementing using heap

Heap is used to implement priority queues

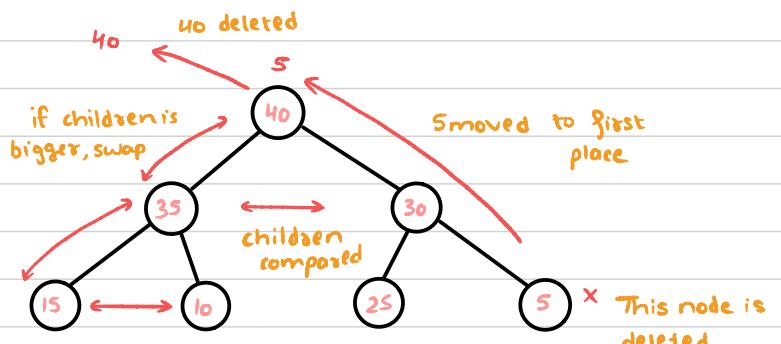
DELETION FROM HEAP Only the root node can be deleted from heap i.e. the first node

```
void Delete (int A[], int n)
{
    int x, i, j;
    x = A[n];
    A[1] = A[n]; // Smallest or last node is copied at first place
    i = 1; j = 2 * i; // i is at index 1 and j is at its left child

    while (j < n - 1)
    {
        if (A[j + 1] > A[j]) // Right child is greater than left child
            j = j + 1;

        if (A[i] < A[j]) // Parent is smaller than child
        {
            swap(A[i], A[j]);
            i = j;
            j = 2 * j;
        }
        else
            break;
    }

    A[n] = x;
}
```



HEAP SORT

while deleting, store the deleted element at vacant position

Through this, you will get sorted array in ascending order after deleting all the elements

40	35	30	15	10	25	5			
1	2	3	4	5	6	7	8	9	10

5	35	30	15	10	25				
1	2	3	4	5	6	7	8	9	10

- STEPS
- nlogn • Create heap of 'n' elements
 - nlogn • Delete 'n' elements 1 by 1
- $O(n \log n)$

35	15	30	5	10	25	40			
1	2	3	4	5	6	7	8	9	10



SORTING TECHNIQUES

1. Bubble Sort
2. Insertion Sort
3. Selection Sort
4. Heap Sort
5. Merge Sort
6. Quick Sort
7. Tree Sort

8. Shell Sort

9. Count Sort
10. Bucket / Bin Sort
11. Radix Sort

$O(n^2)$

$O(n \log n)$

$O(n^{3/2})$

$O(n)$

Comparison based
Sorts

There is no best sorting technique.

You have to choose the technique which matches your requirements

Index Based Sort

// Faster but memory consumption
is high

CRITERIA FOR ANALYSIS

1. Number of comparisons
2. Number of swaps
3. Adaptive // If already the array is sorted, it should make minimum comparisons
4. Stable
5. Extra Memory required

1. List sorted on basis of name

Name : A B C D E F G

Marks : 5 8 6 4 6 7 10

Duplicate elements

2. List sorted on basis of marks

Name : D A C F F B G

Marks : 4 5 6 6 7 8 10

Here also 'C' should come before 'E'

This type of algorithms are
useful in databases

If the sorting algorithm is preserving
the order of duplicate elements in the
sorted list then that algorithm is called
'stable'.

1. BUBBLE SORT

A [8 | 5 | 7 | 3 | 2]
 0 1 2 3 4 $n = 5$

1st 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

Largest element
is sorted in first pass

4 comp
4 Swap

2nd Pass

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

3 comp
3 swap

3rd Pass

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

2 comp
2 swap

4th Pass

3	2		
2	3		
5	5		
7	7		
8	8		

1 comp
1 swap

No of passes : 4
 $: (n-1)$

→ Not actual numbers of swaps but maximum number of swaps

No of comparison : $1+2+3+4$
 $: 1+2+3+4.....(n-1)$
 $: \frac{n(n-1)}{2} O(n^2)$

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[i], A[j+1]);
 flag=1;

}

if (flag == 0)
 break;

}

• Called as bubble sort as lighter or smaller elements come up same as bubbles

• Performing selected number of passes can give same number of largest elements as number of passes performed

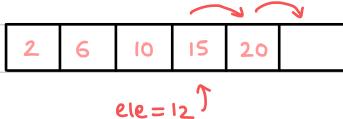
• Bubble Sort is adaptive

• It is stable

• 'k' passes give 'k' numbers of largest element

2. INSERTION SORT

Inserting element in a sorted array at a sorted position



// Start comparison from last element

EXAMPLE : A

8	5	7	3	2
0	1	2	3	4

// Assume only first element to be sorted
n = 5

1st Pass

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

1 Comp
1 swap
Take out one element and compare and insert

3rd Pass

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

3 comp
3 swap

2nd Pass

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

2 comp
2 swap

4th Pass

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

4 comp
4 swap

No of passes : (n-1)

```
void InsertionSort(int A[], int n)
{
```

No of comparison : $\frac{n(n-1)}{2}$ O(n²)

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

No of swaps : $\frac{n(n-1)}{2}$ O(n²)

```
j = i-1;
```

```
x = A[i];
```

```
while(j > -1 && A[j] > x)
```

```
A[j+1] = A[i];
```

```
j--;
```

```
A[j+1] = x;
```

- Insertion Sort is adaptive as no swapping is done if array is sorted. Only 'n' comparisons are done which take minimum time. It is adaptive by nature (no flag used)

```
}
```

- Insertion sort is stable
- Used in linked list.

3. SELECTION SORT

↳ Selecting a position in the array and finding the smallest element suitable for that.

A

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

 $n = 6$

1st Pass 2nd Pass 3rd Pass 4th Pass 5th Pass

0 8 $\leftarrow i$	0 2	0 2	0 2	0 2	0 2
1 6	1 6 $\leftarrow i$	1 3	1 3	1 3	1 3
2 3	2 3 $\leftarrow k$	2 6 $\leftarrow i$	2 4	2 4	2 4
3 2 $\leftarrow k$	3 8	3 8	3 8 $\leftarrow i$	3 5	3 5
4 5	4 5	4 5	4 5 $\leftarrow k$	4 8 $\leftarrow i$	4 6
5 4	5 4	5 4 $\leftarrow k$	5 6	5 6 $\leftarrow k$	5 8

No of comparison: $\frac{n(n-1)}{2}$ O(n²)

No of swaps : n-1 O(n)

• k passes give k number of smallest elements

• Selection Sort is not adaptive

• Selection Sort is not stable

• Selection Sort performs minimum number of swaps

void SelectionSort(int A[], int n)

{

int i, j, k;

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

{

for (j=k=i; j < n; j++)

{

if (A[j] < A[k])

k = j;

swap (A[i], A[k]);

}

}

5. MERGING

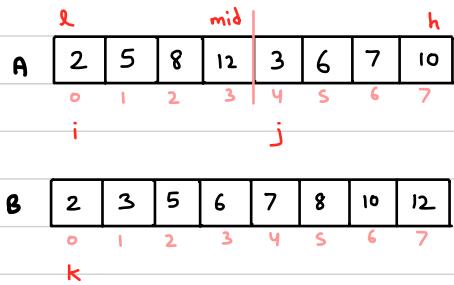
- Merging two lists in third array
- Merging two lists in single array
- Merging multiple list

MERGING TWO LISTS IN THIRD ARRAY

$O(m+n)$

	A(m)	B(n)	C	
0	2	i 0 4	j 0 2	k
1	10	1 9	1 4	
2	18	2 19	2 9	
3	20	3 25	3 10	
4	23		4 18	
			5 19	
			6 20	
			7 23	
			8 25	

MERGING TWO LISTS IN SINGLE ARRAY



```
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 < m; i++) // for remaining elements
        c[k++] = A[i]; // in either of list
```

// Only one of these loops

```
    for (j = n; j < n; j++)
        c[k++] = B[j];
```

```
void Merge (int A[], int l, int mid, int h)
{
```

```
    int i, j, k;
    int B[h+1];
    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++];
```

```
    for (i = l; i <= mid; i++)
```

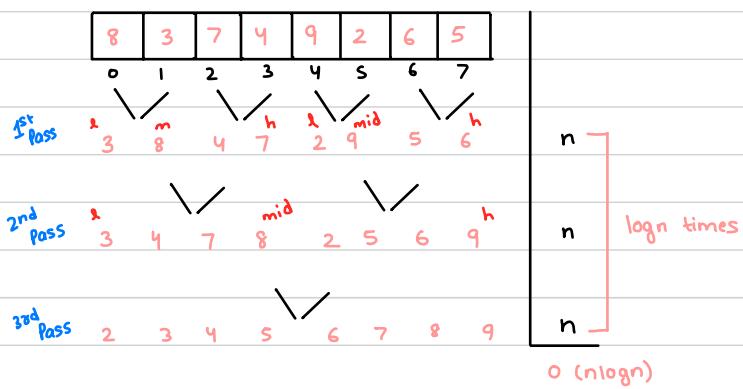
```
        B[k++] = A[i];
```

```
    for (j = mid + 1; j <= h; j++)
```

```
        B[k++] = A[j];
```

ITERATIVE MERGESORT

We consider each of these elements to be a sorted list



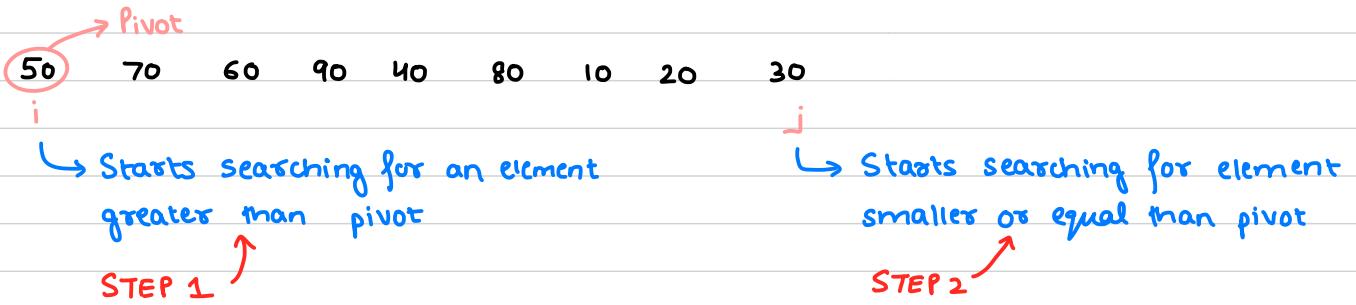
RECURSIVE MERGE SORT O(n log n)

```
void RMergesort (int A[], int l, int h)
{
    if (l < h)
    {
        mid = ⌊(l+h)/2⌋;
        RMergesort (A, l, mid);
        RMergesort (A, mid+1, h);
        Merge (A, l, mid, h);
    }
}
```

```
void IMergesort (int A[], int h)
```

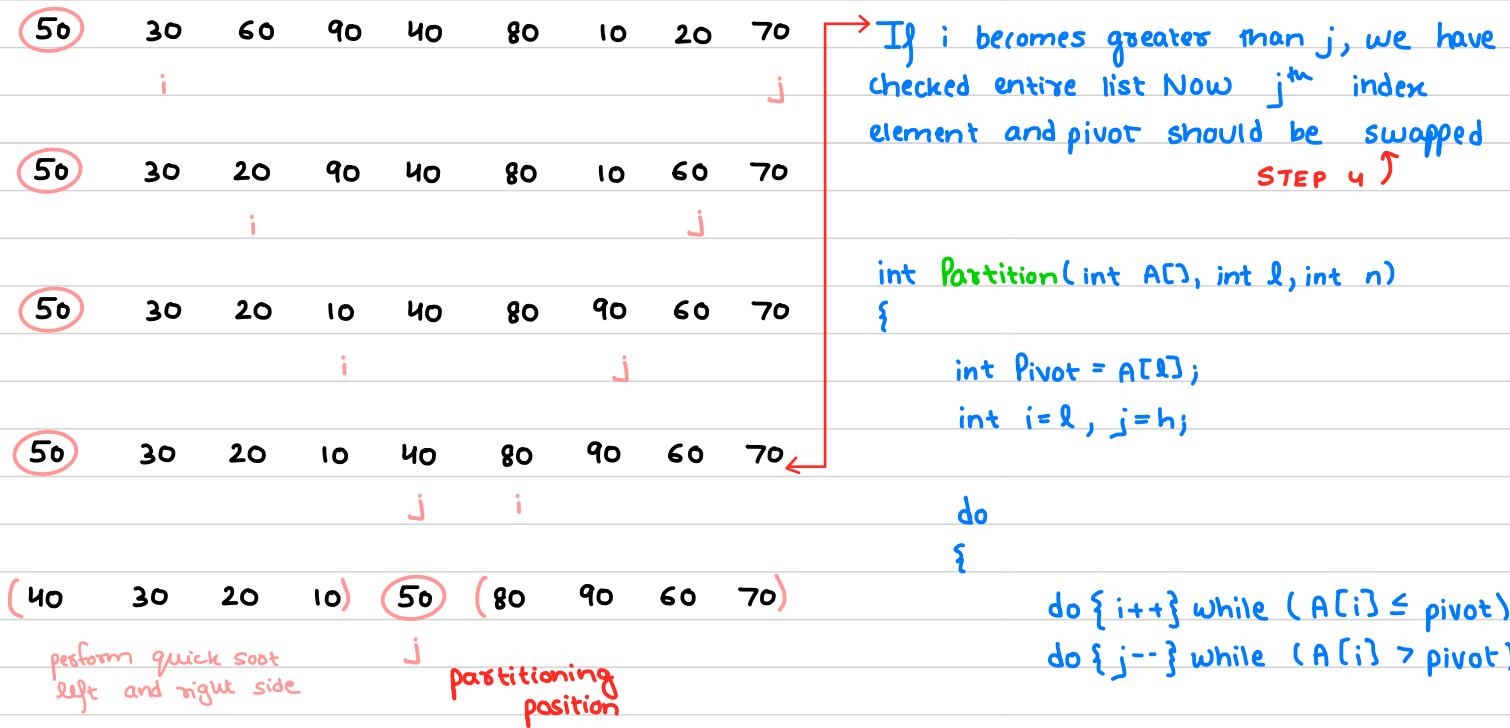
```
{
    int p, i, l, mid, h;
    for (p=2; p <= n; p=p*2)
    {
        for (i=0; i+p-1 < n; i=i+p)
        {
            l = i;
            h = i+p-1;
            mid = ⌊(l+h)/2⌋;
            Merge (A, l, mid, h);
        }
    }
    if (p/2 < n)
        Merge (A, 0, p/2, n-1);
    // If numbers of elements are odd
}
```

6. QUICK SORT → Selecting a element and then finding its position



PARTITIONING PROCEDURE

When larger and smaller elements found,
exchange them ↪ STEP 3



Best Case : If partitioning is in middle
 $O(n \log n)$

Worst Case: If partitioning is on any end
 $O(n^2)$

Avg case : $O(n \log n)$

- best case: sorted list
(first make middle element as pivot)
 - worst case: partitioning on any end
 $\Theta(n^2)$

```

int Partition( int A[], int l, int n)
{
    int Pivot = A[l];
    int i=l, j=h;

    do
    {
        do { i++ } while ( A[i] <= pivot);
        do { j-- } while ( A[i] > pivot);

        if ( i < j)
            swap( A[i], A[j]);

    } while ( i < j);

    swap( A[l], A[j]);
    return j;
}

```

8. SHELL SORT

Used for large arrays

```
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;
            }
            A[j+gap] = temp;
        }
    }
}

void main()
{
    int A[] = { 11,13,7,12,16,9,24,5,10,3 },n=10;
    ShellSort (A,n);
}
```

9. COUNT SORT

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

C	0	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	20

Void CountSort(int A[], int n)

{

int max, i, j;
int *c;

max = findMax(A, n);

C = new int[max + 1];

C = (int *)malloc(sizeof(int) * (max + 1));

for (i = 0; i < max + 1; i++) n
 C[i] = 0;

for (i = 0; i < n; i++) n
 C[A[i]]++;

i = 0, j = 0;

while (i < max + 1) n
 {

 if (C[i] > 0)
 {
 A[j++] = i;
 C[i] --;

}

 else
 i++;

}

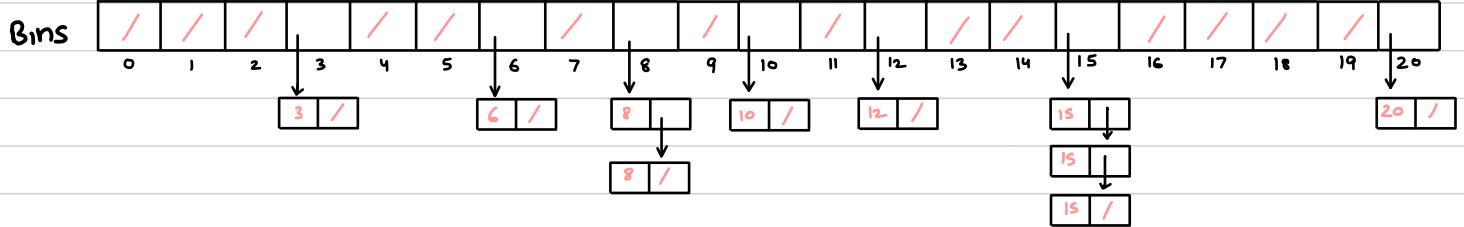
O(n)

Fastest sorting method but
high memory consumption

10. BUCKET / BIN SORT

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

ARRAY OF LINKED LIST



```
void BinSort(int A[], int n)
```

```
{
```

```
    int max, i, j;
```

```
    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]); // Insert at end of linked list
```

```
i=0, j=0;
```

```
while (i < max + 1)
```

```
{
```

```
    while (Bins[i] != NULL)
```

```
{
```

```
        A[j++] = Delete(Bins[i]); O(n)
```

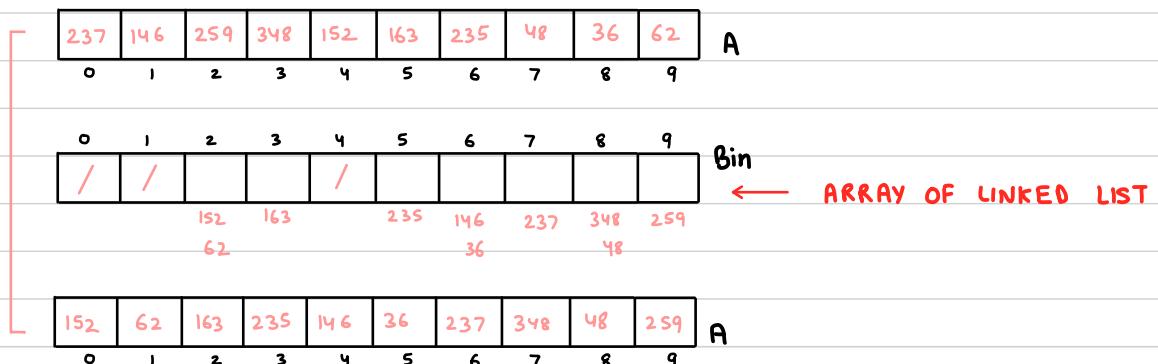
```
}
```

```
    i++;
}
```

```
}
```

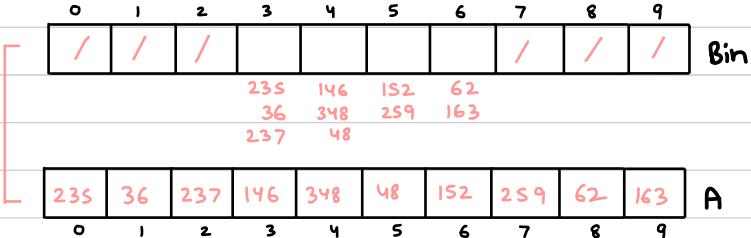
11. RADIX SORT

1st Pass
Elements arranged
on basis of
one's digit

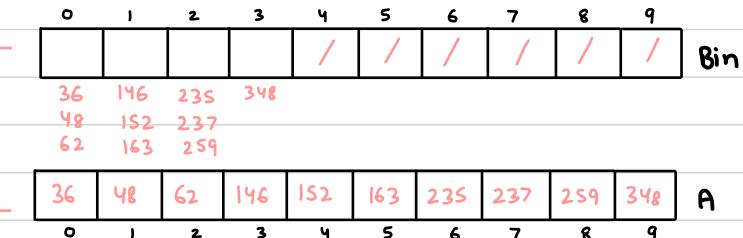


// Take out the elements in
FIFO way and store them

2nd Pass
two's digit



3rd Pass
three's digit



two digit numbers
third digit considered
as 0

$$1. [A[i] / 1] \% 10$$

Decimal numbers require 10 sized bin (0-9)
Binary numbers require 2 sized bin (0,1)
Octal numbers require 8 sized bin (0-7)

$$2. [A[i] / 10] \% 10$$

$O(n)$ and minimum storage required
Storage depends on maximum number
of digits of largest element

$$3. [A[i] / 100] \% 10$$



HASHING TECHNIQUE

- Why hashing?

There are three methods of searching

1. Linear Search - $O(n)$
2. Binary Search - $O(\log n)$ // But in binary search, elements must be arranged in sorted order, so we have to do some extra work
3. Hashing - $O(1)$

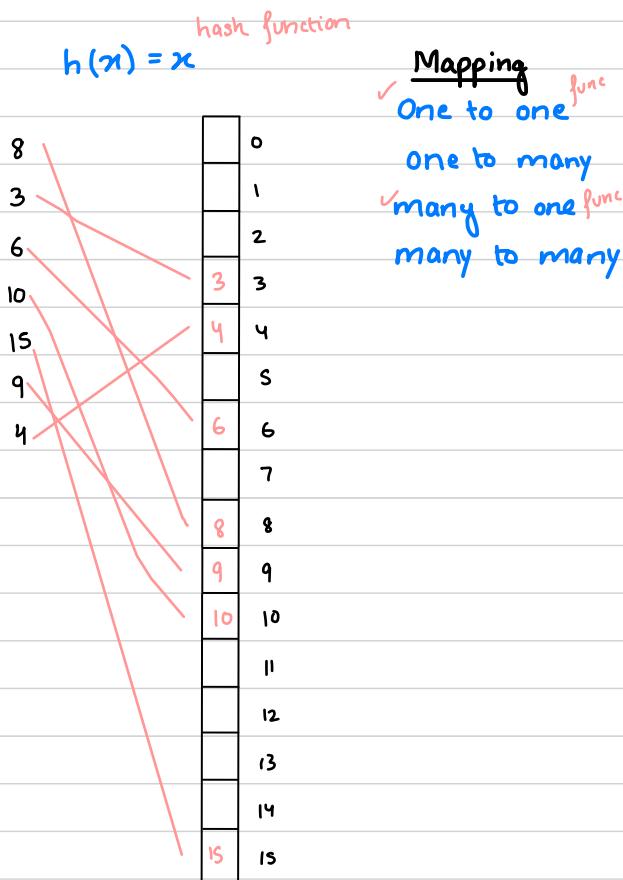
→ It is the fastest method, but requires very large space as it depends on maximum element

DRAWBACK

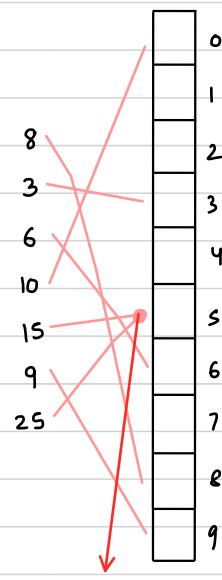
A	3	6	8	8	10	12	15	15	15	20
	0	1	2	3	4	5	6	7	8	9

C	0	0	0	3	0	0	6	0	8	0	10	0	12	0	0	15	0	0	0	0	20
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

- Ideal Hashing



$h(x) = x \% 10$ gives remainder many-one



- How to avoid COLLISION?

Open Hashing // NO space restriction

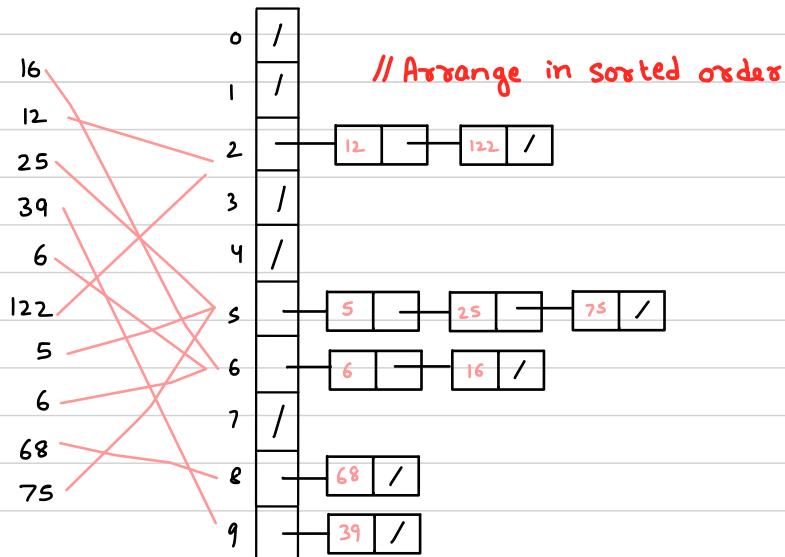
- Chaining

Closed Hashing // Limited Space

- Open Addressing // If collision, store element elsewhere in free space
 - 1. Linear Probing
 - 2. Quadratic Probing
 - 3. Double Hashing

Methods for storing in free space

- CHAINING



Arrange on basis of first digit

Average Successful Search

$$h(x) = x \mod 10 \leftarrow \text{Hash Function}$$

$$t = 1 + \frac{\lambda}{2}$$

$n = 100$ No of keys $\text{size} = 10$ No of index $\lambda = \frac{n}{\text{size}}$ LOADING FACTOR

Average Unsuccessful Search

$$t = 1 + \lambda$$

If keys are : 5, 35, 95, 145, 175, 265, 845

Then modify your hash function

PROGRAM

```
void Insert( struct Node *H[], int key )  
{
```

```
    int index = hash(key);  
    SortedInsert( &H[index], key );
```

```
}
```

```
int hash( int key )
```

```
{
```

```
    return key % 10;
```

```
}
```

```
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 <= q && p->data < x )
```

```
{
```

```
        q = p;
```

```
        p = p->next;
```

```
}
```

```
    if ( p == *H )
```

```
{
```

```
        t->next = *H;
```

```
*H = t;
```

```
}
```

```
else
```

```
{
```

```
        t->next = q->next;
```

```
        q->next = t;
```

```
}
```

```
struct Node
```

```
{
```

```
    int data;
```

```
    struct Node *next;
```

```
};
```

```
void main()
{
    struct Node *HashTable[10], *temp;
    int i;

    for (i=0; i<10; i++)
        HashTable[i] = NULL;

    Insert(HashTable, 12);
    Insert(HashTable, 22);
    Insert(HashTable, 42);

    temp = Search(HashTable[hash(21)], 21);
}
```

```
struct Node *Search(struct Node *p, int key)
{
    while (p != NULL)
    {
        if (key == p->data)
            return p;
        else
            p = p->next;
    }
    return NULL;
}
```

LINEAR PROBING

$$h(x) = x \cdot 10$$

30	0
29	1
	2
45	3
23	4
43	5
25	6
43	7
74	8
19	9
29	

$$h'(x) = (h(x) + f(i)) \cdot 10 \quad // \text{For probing}$$

$$f(i) = i \\ \text{where } i = 0, 1, 2, \dots$$

In case of collision,
insert at next free space

$$h'(25) = (h(25) + f(0)) \cdot 10 \\ = (5 + 0) \cdot 10 = 5$$

$$h'(25) = (h(25) + f(1)) \cdot 10 \\ = (5 + 1) \cdot 10 = 6$$

$$h'(25) = (h(25) + f(2)) \cdot 10 \\ = (5 + 2) \cdot 10 = 7$$

$$h'(29) = (h(29) + f(0)) \cdot 10 \\ = (9 + 0) \cdot 10 = 9 \\ = (9 + 1) \cdot 10 = 0 \\ = (9 + 2) \cdot 10 = 1$$

Cyclic behaviour

For searching : key : 45 ✓ $45 \cdot 10 = 5$ found at index 5

key : 74 ✓ $74 \cdot 10 = 4$ • Not found at index 4

- Search Next index until 74 is found or blank space is found
- Found at index 8

// Search takes more than constant time

Key : 40 ✗

$40 \cdot 10 = 0$ • Not found at index 0

- At index 2, blank space found
- Element does not exist

ANALYSIS

$$\lambda = \frac{n}{\text{size}} = \frac{9}{10} = 0.9$$

DRAWBACK

** $\lambda \leq 0.5$ // Table should be almost half filled so, there are blank spaces and searching can be made faster

Average Successful Search

$$t = \frac{1}{\lambda} \ln \left(\frac{1}{1-\lambda} \right)$$

Average Unsuccessful Search

$$t = \frac{1}{1-\lambda}$$

DELETION IS NOT EASY IN LINEAR PROBING

PROGRAM

```
int hash( int key )
{
    return key % 10;
}
```

```
int probe( int H[], int key )
{
    int index = hash(key);
    int i = 0;

    while ( H[(index + i) % 10] != 0 )
        i++;
    return (index + i) % 10;
}
```

```
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;

    while ( H[(index + i) % 10] != key )
        i++;
    return (index + i) % 10;
}
```

```
Void main()
{
    struct Node *HashTable[10], *temp;
    int i;

    for (i=0; i<10; i++)
        HashTable[i] = NULL;

    Insert(HashTable, 12);
    Insert(HashTable, 22);
    Insert(HashTable, 42);

    temp = Search(HashTable, 35);
}
```

QUADRATIC PROBING

The drawback of linear probing is that elements cluster together and form a group. To resolve this issue, quadratic probing is introduced.

0	$h'(x) = (h(x) + f(i)) \cdot 10$ where $f(i) = i^2$ $i=0, 1, 2, \dots$
1	
2	
3	$h'(23) = (h(13) + f(0)) \cdot 10$ = $(3 + 0) \cdot 10 = 3$
4	<u>Average Successful Search</u> $\frac{-\log_e(1-\lambda)}{\lambda}$
5	$h'(43) = (h(43) + f(0)) \cdot 10$ = $(3 + 0) \cdot 10 = 3$
6	<u>Average Unsuccessful Search</u> $\frac{1}{1-\lambda}$
7	= $(3 + 1) \cdot 10 = 4$
8	
9	$h'(13) = (h(13) + f(2)) \cdot 10$ = $(3 + 4) \cdot 10 = 7$

DOUBLE HASHING

0	$h_1(x) = x \cdot 10$
1	$h_2(x) = R - (x \cdot R)$ where R is prime num smaller than size of Hash Table
2	$h'(x) = (h_1(x) + i * h_2(x)) \cdot 10$ where $i = 0, 1, 2, \dots$
3	
4	
5	$h'(25) = (5 + 1 * 3) \cdot 10 = 8$
6	$7 - (25 \cdot 7)$
7	$7 - 4 = 3$
8	$h'(15) = (5 + 1 * 6) \cdot 10 = 1$
9	$7 - (15 \cdot 7)$ $7 - 1 = 6$

$$h'(35) = (5 + 1 * 7) \cdot 10 = 2$$

$$7 - (35 \cdot 7)$$

$$7 - 0 = 7$$

$$h'(95) = (5 + 3 * 3) \cdot 10 = 4$$

$$7 - (95 \cdot 7)$$

$$7 - 4 = 3$$



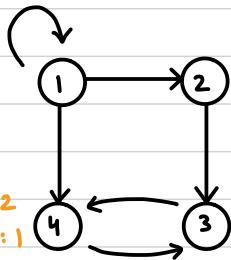
GRAPHS

$$\rightarrow G = (V, E)$$

V = Set of vertices

E = Set of edges

Self loop

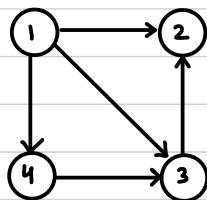


Directed Graph

- Edges are having directions

Indegree: 2
Outdegree: 1

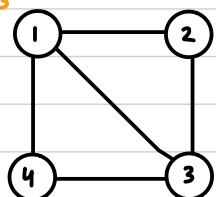
Parallel Edges



Simple Diagraph

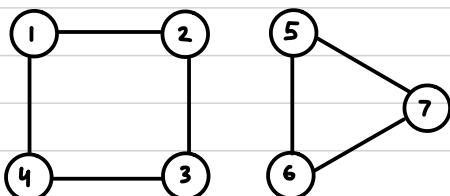
- Without self loop
- Without parallel edges

degree 3



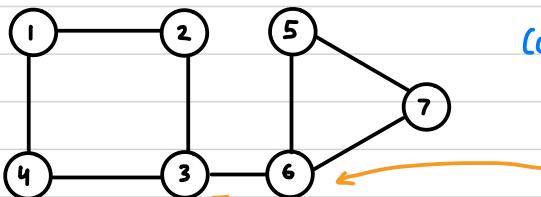
Graph / Non - directed Graph

- Undirected Edges



Non-connected Graph

2 components

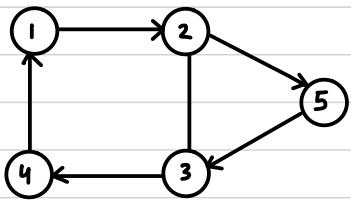


Connected Graph

Articulation Points

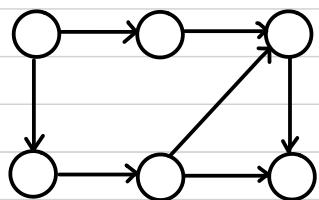
- Those vertices whose removal will divide the graph into multiple components

Connected components



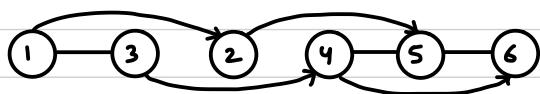
Strongly Connected Graph (Directed Graph)

- All vertices can be reached from any vertex
- There is a path between every pair of vertices
- Path is set of vertices which are connecting pair of vertices
- Cycle is a circular path that is starting from same vertex and ending at same vertex



Directed Acyclic Graph (DAG)

- Directed Graph
- With no cycles
- These can be arranged linearly such that edges are going in only forward direction

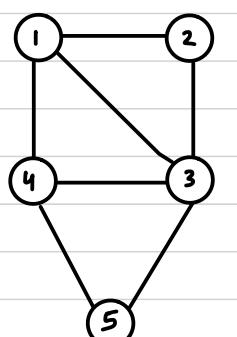


Topological Ordering

REPRESENTATION OF UNDIRECTED GRAPH

- (1) Adjacency Matrix
- (2) Adjacency List
- (3) Compact List

ADJACENCY MATRIX



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

5 × 5

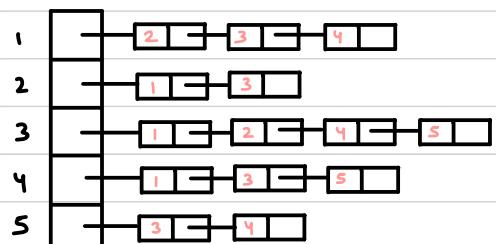
$$|V| = n = 5$$

$$|E| = e = 7$$

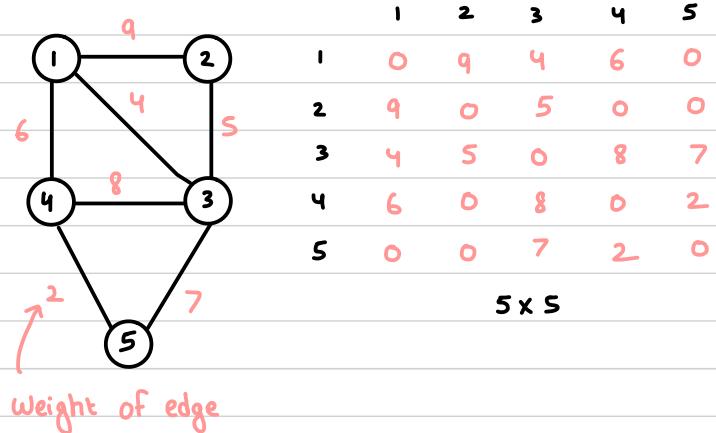
$$(i, j)$$

$$A[i][j] = 1$$

ADJACENCY LIST



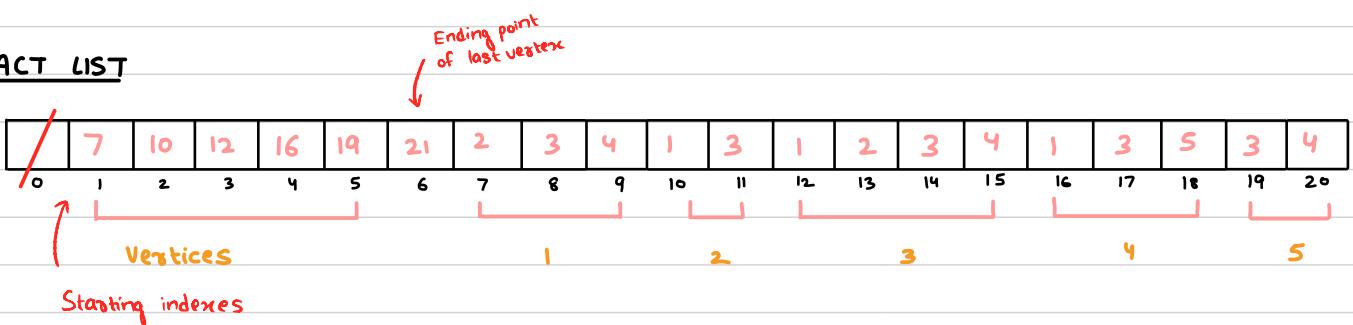
COST ADJACENCY MATRIX (Representation of weight)



COST ADJACENCY LIST

Add another area to node for weight of edge

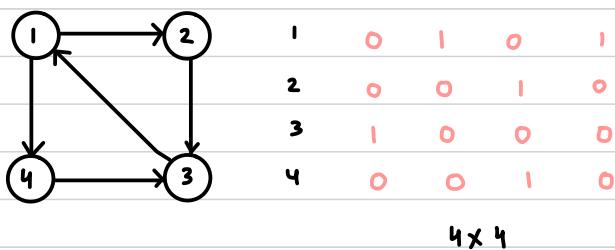
COMPACT LIST



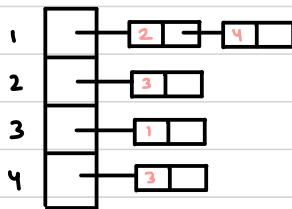
$$|V| + 2|E| + 1 \\ 5 + 2 \times 7 + 1 = 20 \\ 20 + 1 = 21$$

REPRESENTATION OF DIRECTED GRAPH

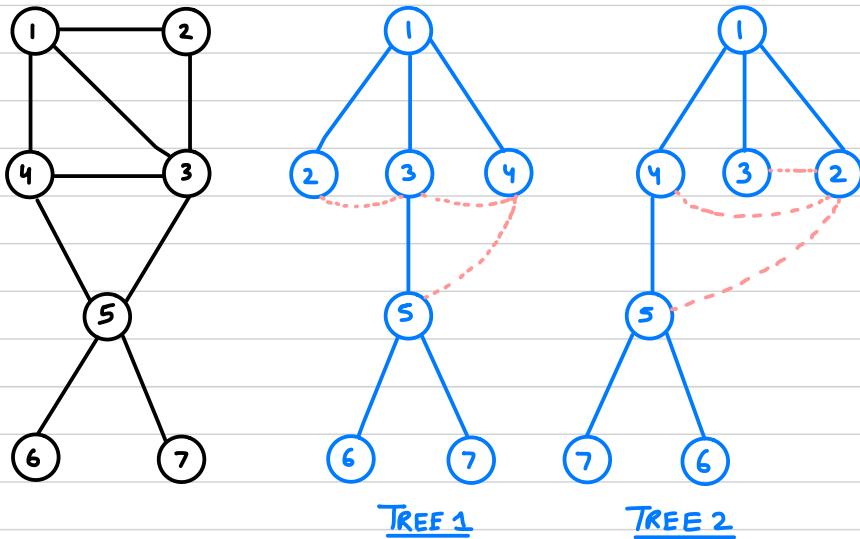
ADJACENCY MATRIX



ADJACENCY LIST



BREADTH FIRST SEARCH (BFS)



Multiple trees can be generated from the same graph

- We can start from any vertex
 - Each vertex should be explored fully
 - Dotted pink lines represent edges that make a complete cycle called cross edges
 - These trees are called BFS Spanning Trees

BFS TREE 1 : 1, 2, 3, 4, 5, 6, 7 // Same as Level Order Traversal in trees

BFS TREE 2 : 1, 4, 3, 2, 5, 7, 6

Starting vertex

8

```
int U, V;  
pointFl("y.d", i);  
visited[i] = 1;  
enqueue(q, i);
```

```
while ( ! isEmpty (q) )  
{
```

$v = \text{dequeue}(q);$ ← Take out vertex from queue

for scanning
row of
matrix —

```
for( V=1 ; V<=n ; V++ )
```

Should not be
visited

```
if (A[v][v] == 1 && visited[v] == 0)
{
```

```
printf(" y. d ",v);  
visited[v]=1;  
enqueue(q,v);
```

3

1

i

```
printf(" %d", v);  
visited[v] = 1;  
enqueue(q, v);
```

QUEUE
9/

--	--	--	--	--	--	--	--

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

DEPTH FIRST SEARCH (DFS)

```

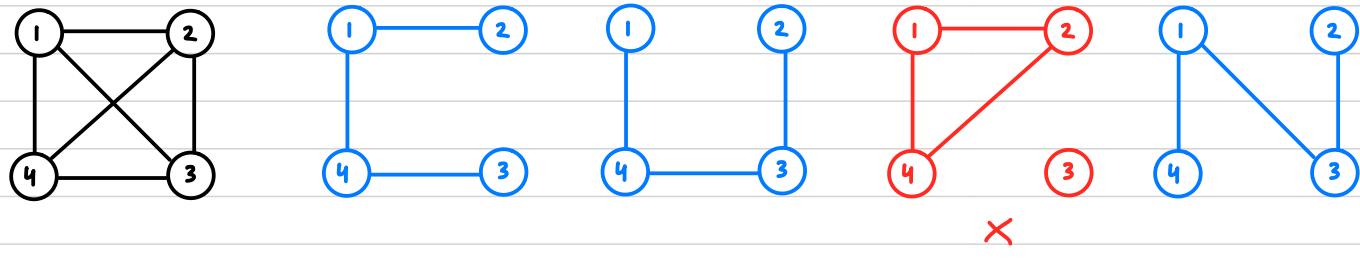
void DFS (int u)
{
    int v;
    if (visited [u] == 0)
    {
        printf (" %d ", u);
        visited [u] = 1;

        for (v = 1; v <= n; v++)
        {
            if (A [u] [v] == 1 && visited [v] != 0)
                DFS (v);
        }
    }
}

```

SPANNING TREES

Spanning tree is a sub graph of a graph having all vertices of a graph and $|V|-1$ edges and there should not be any cycle.



$$|V| = 4$$

$$|E| = 6$$

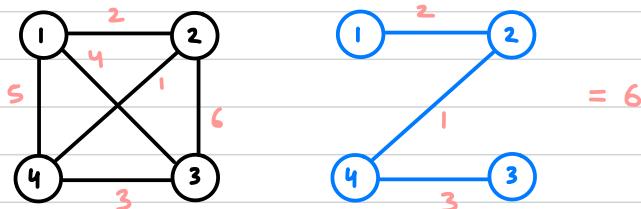
$|E|_C_{|V|-1} = {}^6 C_3$ ways or numbers of spanning trees
 but not include trees forming cycles

$$|E|_C_{|V|-1} - \text{cycles} = {}^6 C_3 - 4 // \text{In this example}$$

$$= 16$$

MINIMUM COST SPANNING TREE

If weights are added to edges of graph, then the spanning tree with minimum cost of weights is called minimum cost Spanning tree.

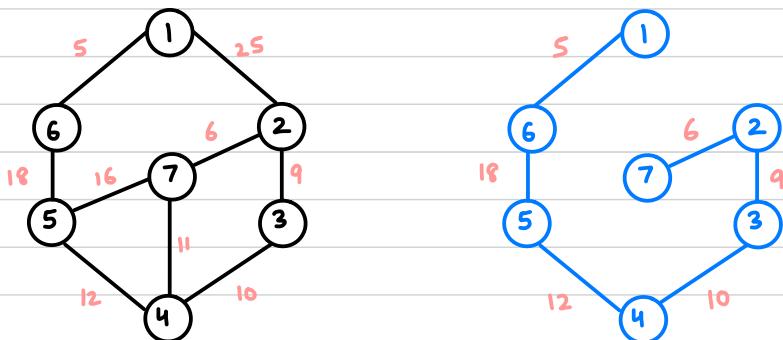


PRIM'S MINIMUM COST SPANNING TREE

$$(|V| - 1) |E|$$

$$ne = n \times n$$

$$O(n^2)$$

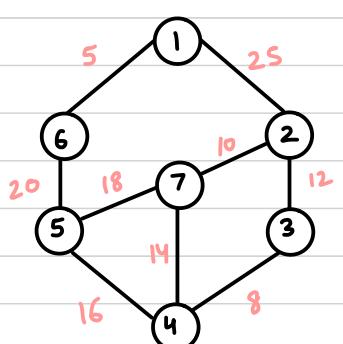


STEP 1 : Select the edge with least weight along with its vertices

STEP 2 : Then compare the edges joined with the vertices and select the one with least weight along with its vertex

STEP 3 : Repeat step 2 until spanning tree is obtained

PROGRAM



COST	0	1	2	3	4	5	6	7
0	-	-	-	-	-	-	-	-
1	-	-	25	-	-	-	5	-
2	-	25	-	12	-	-	-	10
3	-	-	12	-	8	-	-	-
4	-	-	-	8	-	16	-	14
5	-	-	-	-	16	-	20	18
6	-	5	-	-	-	20	-	-
7	-	-	10	-	14	18	-	-

	0	1	2	3	4	5	6	7
near	/	-	-	-	-	-	-	-

t	0					
1		0	1	2	3	4
	0					
	1	0	1	2	3	4
		0	1	2	3	4

// Only either of lower or upper triangular matrix will be sufficient

```

#define I 32767           ← Largest possible integer value
int cost[8][8], near[8], t[2][6];           ← Initialize as {I}
void main() {                                ← Initialize as the given matrix (Replace '-' with 'I')
    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) {
                min = cost[i][j];
                u = i, v = j;
            }
        }
    }
    t[0][0] = u, t[1][0] = v;
    near[u] = near[v] = 0;
    for (i = 1; i <= n; i++) {
        if (near[i] != 0 && cost[i][u] < cost[i][v])
            near[i] = u;
        else
            near[i] = v;
    }
    for (i = 1; i < n - 1; i++) {
        min = I;
        for (j = i; j <= n; j++) {
            if (near[j] != 0 && cost[j][near[j]] < min)
                min = cost[j][near[j]];
        }
        if (min == I)
            break;
        for (j = i + 1; j <= n; j++) {
            if (near[j] == 0 && cost[j][min] < cost[j][near[i]])
                near[j] = min;
        }
    }
}

```

INITIAL PROCEDURE

RECURSIVE PROCEDURE

```

        min = cost[j][near[i]],
        k = j;
    }
}

```

```

t[0][i] = k;
t[1][i] = near[k];
near[k] = 0;

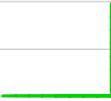
```

```

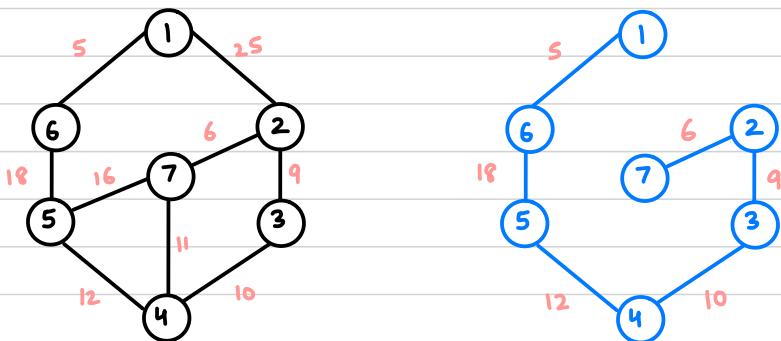
for (j = 1; j <= n; j++)
{
    if (near[j] != 0 && cost[j][k] < cost[j][new[j]])
        near[j] = k;
}

```

Point t;



KRUSKAL'S METHOD



STEP 1 : Select the edge with least weight along with its vertices

STEP 2 : Select the next minimum edge if it is not forming a cycle

DISJOINT SUBSET

$$\mu = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$$

Consider its two subsets

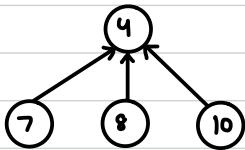
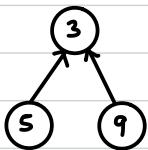
$$A = \{3, 5, 9\}$$

$$B = \{4, 7, 8, 10\}$$

$$A \cap B = \emptyset \quad // \text{Disjoint subset}$$

$$A = \{3, 5, 9\}$$

$$B = \{4, 7, 8, 10\}$$



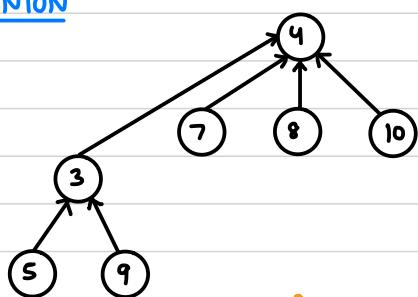
0	1	2	3	-3	-4	3		4	4	3	4
0	1	2	3	4	5	6	7	8	9	10	

No of nodes in negative
(4 is having 4 nodes)

Rest are representing parent

Representation of set

(1) UNION



The parent with more number of nodes

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

Parent of nodes

void Union (int u, int v)

```

{
    if ( s[u] < s[v] )
    {
        s[u] = s[u] + s[v];
        s[v] = u;
    }
    else
    {
        s[v] = s[u] + s[v];
        s[u] = v;
    }
}
  
```

(2) FIND

int Find (int u)

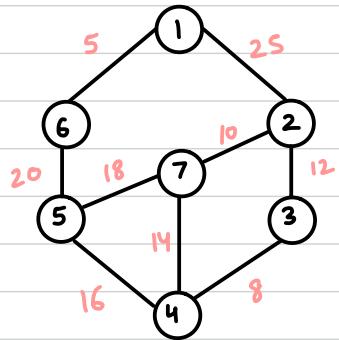
```

{
    int x = u;
    while ( s[x] > 0 )
        x = s[x];
}
  
```

Connecting vertices directly to parent

```

while ( u != x )
{
    v = s[u];
    s[u] = x;
    u = v;
}
  
```



	0	1	2	3	4	5	6	7	8
0	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

edges

X	6	7	4	7	7	-2	-5
0	1	2	3	4	5	6	7

Set

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

included

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

#define I 32767

```
int edges[3][9] = { { 1,1,2,2,3,4,4,5,5 },
{ 2,6,3,7,4,5,7,6,7 },
{ 25,5,12,10,18,16,14,20,18 } };
```

```
int set[8] = {-1};
int included[9] = {0}, t[2][6];
```

```
void main()
{
    int i=0,j,k,n=7,e=9,min,u,v;
```

```
while (i < n-1) // NO of vertices -1
{
```

```
    min = I;
```

For finding minimum cost edge

```
    for (j=0;j<e;j++)
{
```

```
        if ( included[j] == 0 && edges[2][j] < min)
{
```

```
            min = edges[2][j];
            k = j;
```

```
            u = edges[0][j];
```

```
            v = edges[1][j];
```

```
}
```

```
}
```

if ($\text{find}(u) \neq \text{find}(v)$) To check if cycle is
 { forming or not

$t[0][i] = u;$
 $t[1][i] = v;$
 $\text{union}(\text{find}(u), \text{find}(v));$
 $i++;$

}

$\text{included}[k] = 1;$

}

ASYMPTOTIC NOTATION

$$f(n) = \sum_{i=1}^n i = 1+2+3+\dots+n = \frac{n(n+1)}{2} = O(n^2) \quad \text{We can get exact value}$$

$$f(n) = \sum_{i=1}^n i \times 2^i \quad \text{Exact value is not possible that is simplified}$$

So, we use asymptotic notations

Lower Bound	Ω	Omega	less than or equal to
Upper Bound	O	Big O	greater than or equal to
Tight Bound	Θ	Theta	equal