Course : Mastering Data structures & Algorithms using C and C++.

Topics →

1) Arrays
2) Matrices
3) Linked List
4) stack
5) Queues
6) Trees
7) Graph
8) Hashing
9) Recursion
10) sortings

Onlinegdb.

Q. What is data structure?

→ The way you organise the data in the main memory during execution time of a program is called data structure.

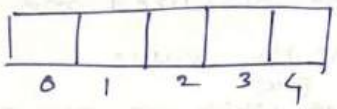
- Physical Data Structure
  - Arrays
  - Matrices
  - Linked List
- Logical Data Structure
  - Stack
  - Queues
  - Trees
  - Graph
  - Hashing
- Recurssion
- Sortiong

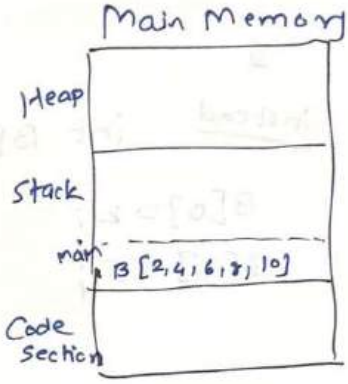
Basic → C++ concepts. → →

① **Arrays** : Arrays are defined as collection of similar data elements.

→ Array is created in the stack memory.

```
int A[5];  → Initialization of Array
int B[5] = {2,4,6,8,10} → Declaration of Array
```





B[2] = 6 B[0] = 2

```
int main ()
{
    int A[10] = {4, 7}

    sizeof (A) → 40        [10 × 4 bytes] =
    A[4] = 0
    A[7] = 0
}
```

```
int main ()
{
    int A[10] = {2, 4, 6, 8, 10, 12};

    for (int x:A)   ← (for each loop).
    {
        cout << x << endl;
    }

    return 0
};
```

Imp.

```
int main()
{
    int n;
    cout << "Enter Size";
    cin >> n;
    int A[n] = {2,4,6,8,10,12};
}
```

It will give error

We cannot Declare ~~initialized~~ variable size array.

instead `int B[n];` It can be initialized and then assign the value.

```
B[0] = 2;
B[1] = 5;
```

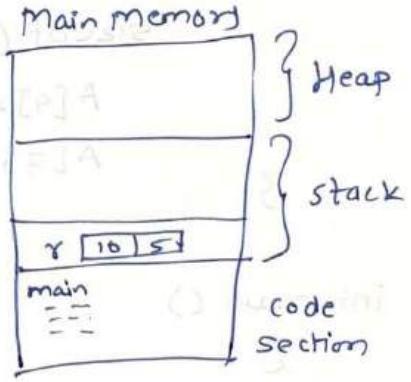
The value/places which are not assign will have garbage value stored.

(2) Structure : It is defined as collection of the similar data items under one name that is grouping the data.

e.g:

```
struct Rectangle
{
    int length;     -4
    int breadth;    -4
}                   8 bytes
```

Initialization / Defination




```
int main()
{
    struct Rectangle r;
    struct Rectangle r={10,5};
```

Declaration

Delaration + initialization

```
r.length = 10;
r.breadth = 5;
```

r
l 10
b 5

```
cout << "Area of Rectangle" r.length x r.breadth)
```

Imp

```
int main()
{
    int n;
    cout << "Enter size";
    cin >> n;
    int A[n]={2,4,6,8,10,12};
}
```

It will give error

We cannot ~~initialized~~ Declare variable size array.

instead `int B[n];` It can be initialized and then assign the value.

```
B[0]=2;
B[1]=5;
```

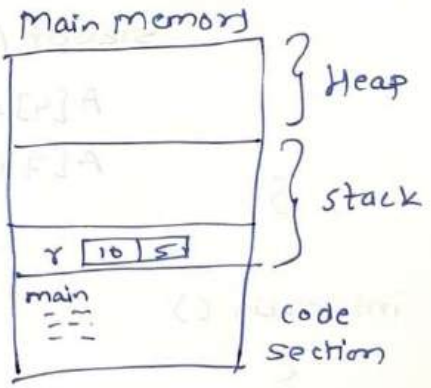
The value places which are not assign will have garbage value stored.

(2) Structure :- It is defined as collection of the similar data items under one name that is grouping the data.

e.g:

```
struct Rectangle
{
    int length;
    int breadth;
};
```

~~Initialization~~ / Defination: length -4, breadth -4, 8 bytes




```
int main()
{
    struct Rectangle r;
    struct Rectangle r={10,5};
```

Declaration / initialization

```
r.length=10;
r.breadth=5;
```

r: length 10, breadth 5

```
cout << "Area of Rectangle" << r.length * r.breadth;
```

Eg: of Structures:

1. Complex No.: → $\sqrt{-1}$

$a+ib$

```
struct complex
{
    int real;
    int img;
};

struct rectangle
{
    ~~long~~ int length;
    int breadth;
} s1, s2, s3;
```

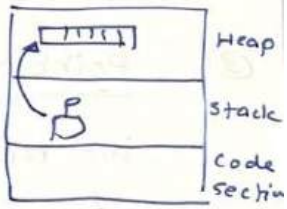
2. student :-)

```
struct student
{
    int roll;
    char name[25];
    char dept[10];
    char address[50];
};
```

(3) Pointers

Pointers: Pointer is an address variable that is meant for storing address of data.

→ Program can access, stack memory and
** code section memory.
** But it can not directly access the heap memory.
So in order to access the heap memory program should have a pointer within itself and with that pointer it can access everything.



→ Program can't access hard disk files directly, because
** hard disk is external. so we need pointer for accessing
* and that pointer should be a file type.

→ Similarly to access, keyboard, monitor, access internet or network connection, all these things are external to a program, so all this things can be accessed with the help of a pointer.

So, pointers are basically used for

① Accessing Heap memory.
② Accessing external resources
③ and for Parameter Passing

e.g.: ①

data variable int a = 10;

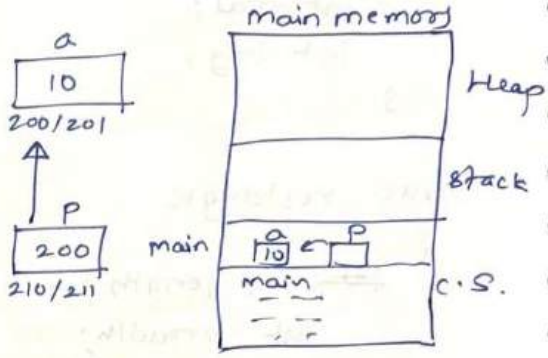
Address variable int *p;

P = &a

cout << a; → 10

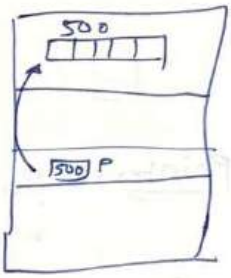
cout << &a; → 200

cout << *a; → 10



②

```
#include <stdlib.h>
int main()
{
    int *p;
C   P = (int *) malloc (5 * sizeof(int));
C++ P = new int[5]
```



③ Pointers to an Array

```
int main()
{
    int A[5] = {2, 4, 6, 8, 10};
    int *p;
      P = A;

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

* pointer to array in heap →

```
int main()
{
    int *p
    p= new int [5];
   p[0]=10; p[1]=15; p[2]=14; p[3]=21; p[4]=31;
    for (int i=0; i<5; i++)
      cout << p[i] <<endl;
      return 0;
   delete [] p;
    return 0;
}
```

** ⇒ whenever you are dynamically allocating the memory then always release the memory after completion/use;

⇒ [delete] → c++ [free] → c

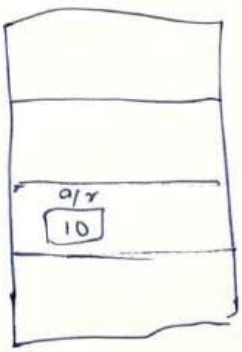
④ Reference

* Reference ⇒ Reference is a nickname given to a variable or alias given to a variable.

** ⇒ Reference is useful in parameter passing in c++.

```
int main()
{
    int a=10;
    int &r=a;
    cout << a; ⟶ 10
      r++
    cout << r; ⟶ 11
    cout << a; ⟶ 11
}
```

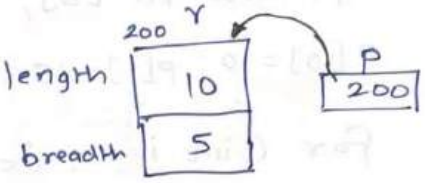
a/r
10 11
200/201



⇒ Reference doesn't consume any memory
It uses same memory of first variable conceptually.

⑤ Pointer to structure

```
struct Rectangle
{
    int length;
    int breadth;
};
```



①
```
int main ()
{
    struct Rectangle r={10,5};
    struct Rectangle *p=&r
        r.length=20
    or  (*p).length = 20;
    or      p->length=20;
}
```

→ Rectangle r={10,5}

→ In C++, writing struct is not mandatory but in C it is mandatory.

②
```
int main ()
{
    struct Rectangle *p
    p= (struct Rectangle *) malloc (sizeof (struct Rectangle))
    p->length=10;
    p->breadth=5;
}
```

Dynamic pointer to struct.

⑥ Functions

Function : function is a piece of code which performs a specific task.

→ Grouping data is a structure

→ Grouping instructions is a function.

Monolithic Programming

```
int main ()
{
   ---
   ---
   ---
   ---
   ---
}
```

Modular / Procedural Programming

```
~~int main()~~
{

Fun1()
{
   ---
   ---
}
Fun2()
{
   ---
   ---
}

int main()
{
   Fun1();
   Fun2();
}
```

Breaking program in small task.

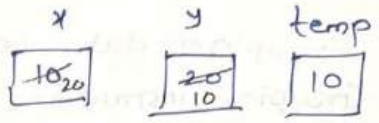
e.g :-

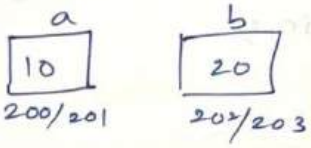
```
int add (int a, int b);        ← Function Declaration
                  ↑ Formal Parameter

int main()
{
   int x, y, z;
   x = 10;
   y = 5;
   z = add (x, y);             ← Function call
                  ↑ Actual Parameter
   cout << " sum is " << z << endl;
}

int add (int a, int b) {       ← Function Defination
   int c;
   c = a + b
   return c;
}
```

* Parameter passing to a function:

① Passing/call by value.

```
void swap ( int x , int y)
{ int temp;
   temp = x;
    x= y ;
    y= temp;
}
int main ()
  {
    int a,b ;
    a=10 ;
    b=20
    swap (a,b);
    cout << a <<b ;
   }
```

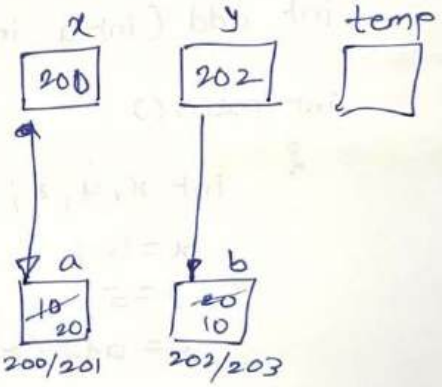




⇒ values of a & b will not get swapped.

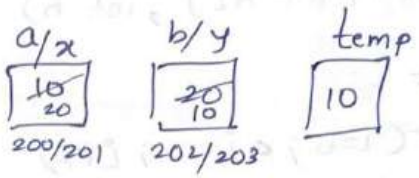
② Passing call by address:

```
void swap (int *x , int *y)
{
   int temp;
   temp=*x ;
   *x= *y;
   *y = temp;
}
   int main()
{  int a,b;
    a=10;
    b=20;
   swap (&a,&b);
   cout << a << b;
}
```



③ Passing/call by Reference:

```
Void swap ( int &x , int &y)
{
   int temp;
   temp = x ;
    x = y
    y = temp;
}
int main ()
 {
   int a,b;
    a=10
    b=20
   swap (a,b);
   cout <<a<<b;
 }
```

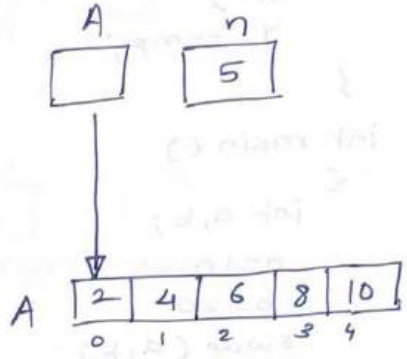


⑦ Array as Parameter

→ Arrays can be passed only by address.

e.g: ① Passing Array as Parameter
or int *A

```
Void fun (int A[] , int n)
{
    int i;
    for(i=0 ; i<n ; i++)
    {
      printf("%d," A[i]
      or cout << A[i] << endl;
    }
}
int main ()
{
    int A[5]={2,4,6,8,10}
    fun(A,5);    ← Array is passed by address
}                  A represents the address
```



② Returning array as from a function:

e.g: int [] fun (int n)
or int* {
    int *p;
    p= new int [n];
    return p;
}

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

⑧ Structure as Parameter

① Call by value:

```
int area (struct Rectangle r1)
{
    r1.length
    return r1.length * r1.breadth;
}
int main()
{
    struct Rectangle r = {10,5};
    cout << area(r);
}
```

← struct is passed by value

If you make changes in formal parameter then actual parameter won't change.

r1.length++

② Call by Reference:

```
int area (struct Rectangle &r1)
{
    return r1.length * r1.breadth;
}
int main()
{
    struct Rectangle r = {10,5}
    cout << area (r)
}
```

← struct is passed by reference

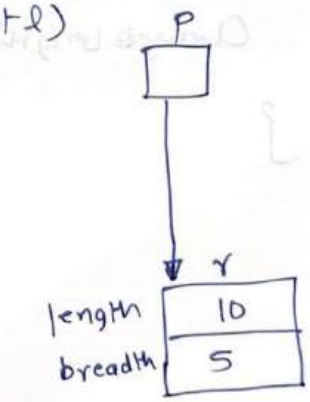
If you make changes in formal parameter then actual parameter will also get change

	r/r1
length	10
breadth	5

③ Call by Address:

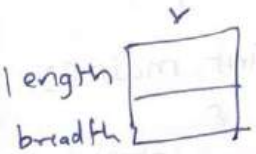
```
void ChangLength (struct Rectangle *p, int l)
{
    p→length = l
}
int main()
{
    struct Rectangle r = {10,5}
    changeLength (&r, 20);
}
```

↳ struct is passed by Address.



* Structure and Functions

```
struct Rectangle
   {
      int length;
      int breadth;
   }
void initialize (struct Rectangle *r, int l, int b)
 {
    r -> length = l;
    r -> breadth = b;
 }
int area (struct Rectangle r)
  {
   return r.length * r.breadth;
  }
void changLength (struct Rectangle *r, int l)
 {
    r -> length = l;
 }
int main()
  {
     struct Rectangle r;
     initialize (&r, 10, 5)
     area (r);
     ChangeLength (&r, 20);
  }
```



* 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;
    }
  int main()
    {
      Rectangle r(10,5);
      r.area ();
      r.changeLength(20);
    }
```

* Recursion *

* Recursion: If a function is calling itself, then it is called Recursive function.

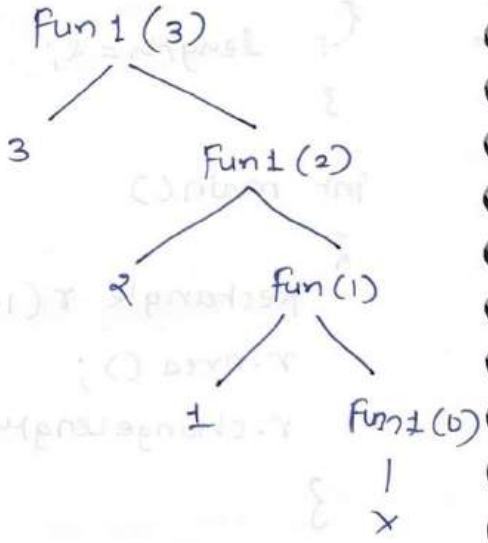
e.g:

```
Type fun (param)
{
    if (< base condition>)
    {
        1. _____
        2. _____
        3. fun (param);
        4. _____
    }
}
```

→ Recursive functions are traced in the form of tree.

e.g:→

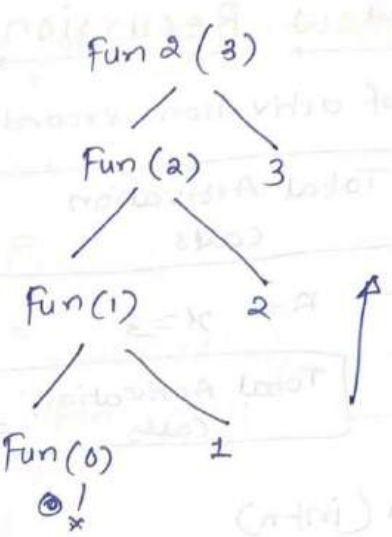
```
void fun1 (int n)
{
    if (n>0)
    {
        1. printf("%d", n)
        2. fun1 (n-1);
    }
}

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



O/p : 3 2 1

(2)

```
void fun2 (int n)
{
  if (n>0)
  {
    fun2 (n-1);
    print ("%d", n);
  }
}
void main()
{
  int x=3;
  fun2 (x);
}
```



O/p = 1 2 3

⇒ Recursion has two phases.
① Calling phase ② Returning phase.

⇒ The difference between loop and Recurssion is Loop has only Ascending (calling) phase but Recurssion has both Ascending as well as Descending phase.

* Generalising Recurssion

```
void fun (int n)
{
  if (n>0)
  {
    1. Calling
    2. Fun (n-1)
    3. Returning.
  }
}
```

Ascending

Descending

* How Recursion Uses Stack

* Numbe rof activation records are depends on Number of rally

Total Activation calls	=	n+1

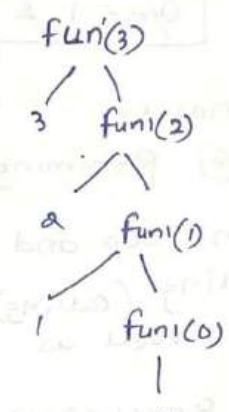
→ O(n)

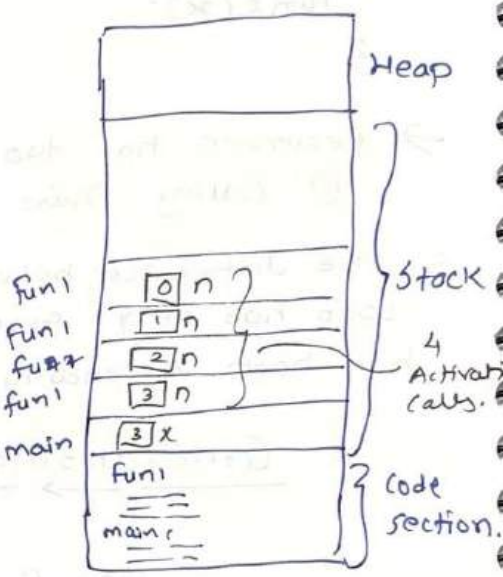
where,
n= value of variable.

e.g: for x=3

Total Activation calls	= 3+1 = 4

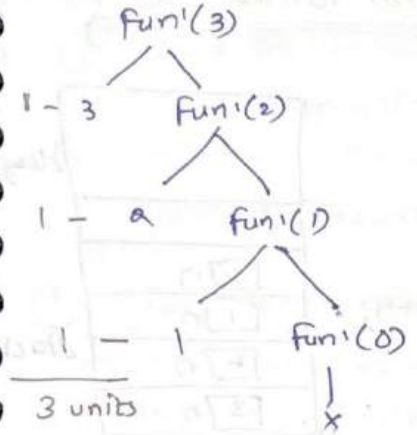
```
Void fun1(int n)
{
  if (n>0)
  {
    printf("%d",n);
    fun(n-1);
  }
}
Void main
{
  int x=3;
  fun1(x);
}
```





⇒ Internally it takes some extra memory for stack.
So the recursive functions are memory consuming

* Recurrence Relation: Time Complexity of Recursion.




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

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

$T(n) = T(n-1) + 1$ — (1)

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

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

$T(n) = T(n-2) + 2$ — (2)

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

$T(n) = T(n-3) + 3$ — (3)

⋮

$\boxed{T(n) = T(n-k) - k}$ — (H)

Assume $n-k=0$ $\therefore n=k$

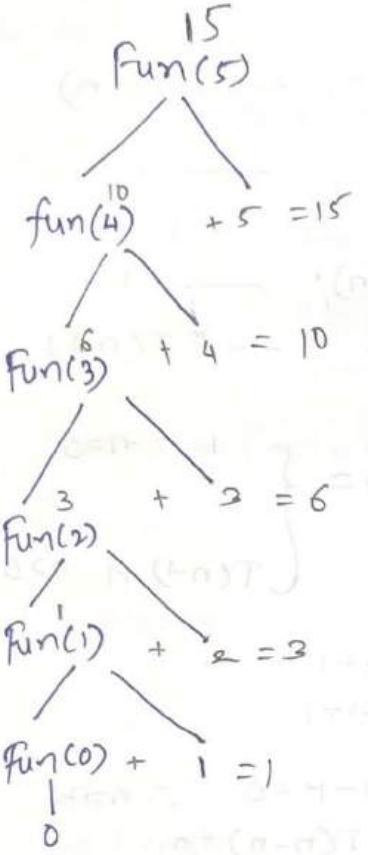
$T(n) = T(n-n) + n$

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

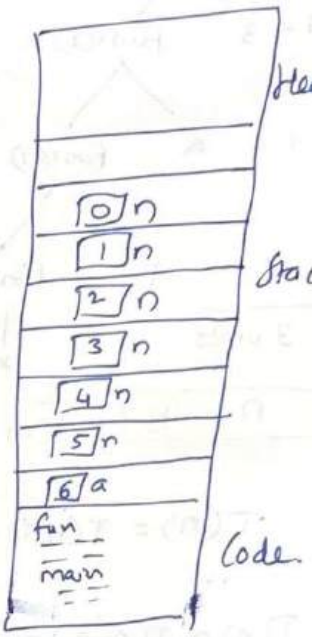
$\boxed{T(n) = 1+n}$ — $O(n)$

* Static Variable in Recurssion *

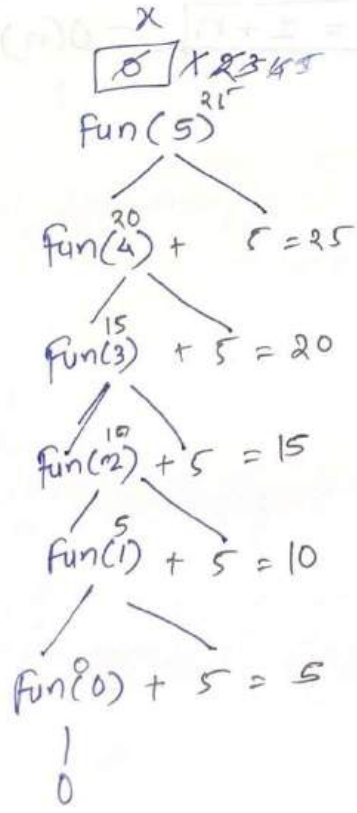
① Local Variable (n)




```
int fun (int n)
{
  if (n>0)
  {
    return fun(n-1)+n;
  }
  return 0;
}
main()
{
  int a=5
  printf("%d, fun(a));
}
```



② Static Variable




```
int fun (int n)
{
  if static int x=0;
    if(n>0)
    {
      x++;
      return fun(n-1)+x;
    }
    return 0;
}
int main()
{
  int a=5;
  cout << fun(a);
}
```

* Types of Recursions

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

(1) Tail Recursion

→ If a recursive function is calling itself, that calle is called as Recurssive call. and aft if the recursive call is the last statement of function, then that is called "Tail Recursion."

e.g:

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

* so in this recursion, everything is performed on calling time only., nothing is performed on returning time.

* In comparison of Tail Recurssion with loop.

→ Tail recurssion requires space in order of 'n' as n+1 activation record will be created in stack. O(n)

→ Loop requires only one activation record. i.e. space occupied is order of 1. O(1).

→ As, Tail Recurssion consumes more space hense it, better to write "Loop" rather than Tail Recurssion.

② Head Recursion :

→ If the first statement of Recursive function is a Recursive calle then it is called 'Head Recursion".

e.g:
```
void fun(int n)
{ if(n>0)
  {
    fun(n-1);
  }
}
```

* In head Recursion function doesn't have to process or preform at the time of calling. It has to do everything only at the returning time.

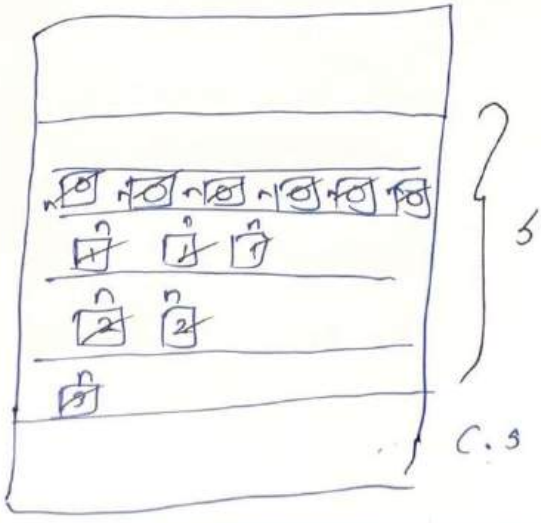
→ In case of head Recursion, it can not be ~~conve~~ easily converted into loop looking as it is.

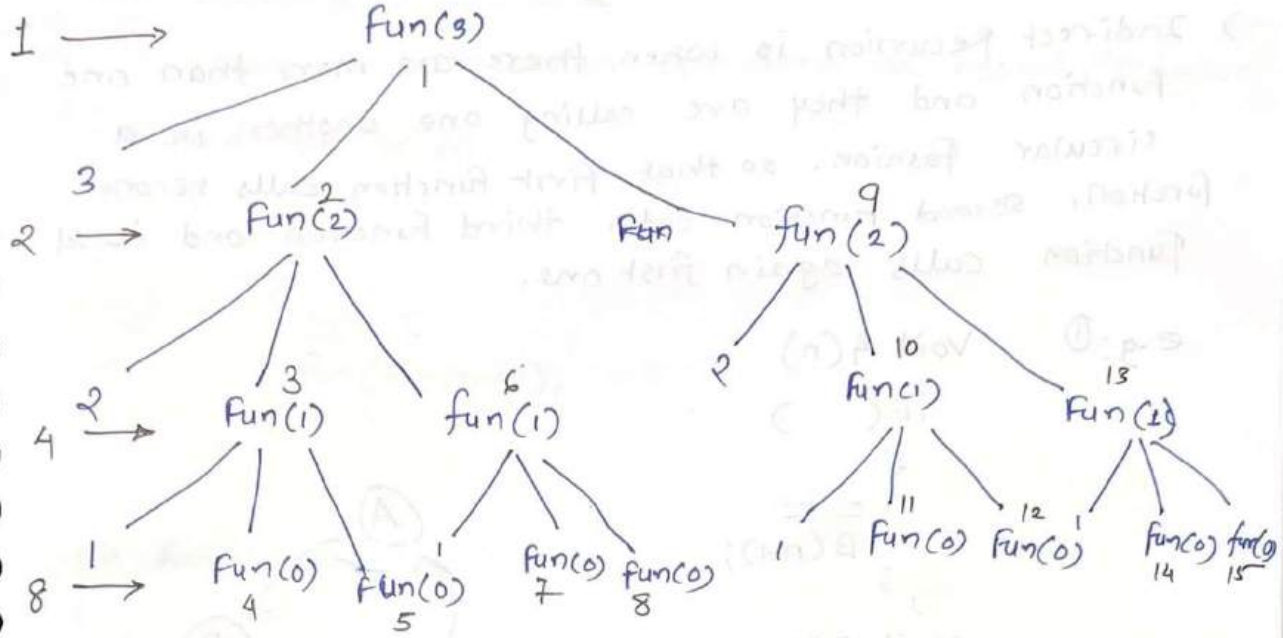
③ Tree Recursion :→

Linear Recursion →
```
fun(n)
{ if(n>0)
  {
    fun(n-1);
  }
}
```

Tree Recursion →
```
fun(n)
{ if(n>0)
  {
    fun(n-1)
    fun(n-1);
  }
}
```

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

fun(3);
```





O/p: 3 2 1 1 2 1 1

→ Total calls = 4 Total Activation records : 15

→ Time complexity = $1 + 2 + 4 + 8 = 15$

$= 2^0 + 2^1 + 2^2 + 2^3$

$= 2^{3+1} - 1$

→ Time complexity = $2^{n+1} - 1 = O(2^n)$

→ Space complexity = Maximum height of stack.

→ As same space gets reused inside the stack.

→ one activation record goes and in the same place, another activation record gets created.

space = $3 + 1$

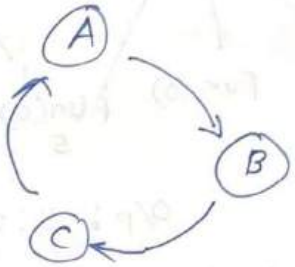
$= n + 1$ $O(n)$

④ Indirect Recursion:

→ Indirect Recursion is when there are more than one function and they are calling one another in a circular fashion. so that first function calls second function, second function calls third function and third function calls again first one.

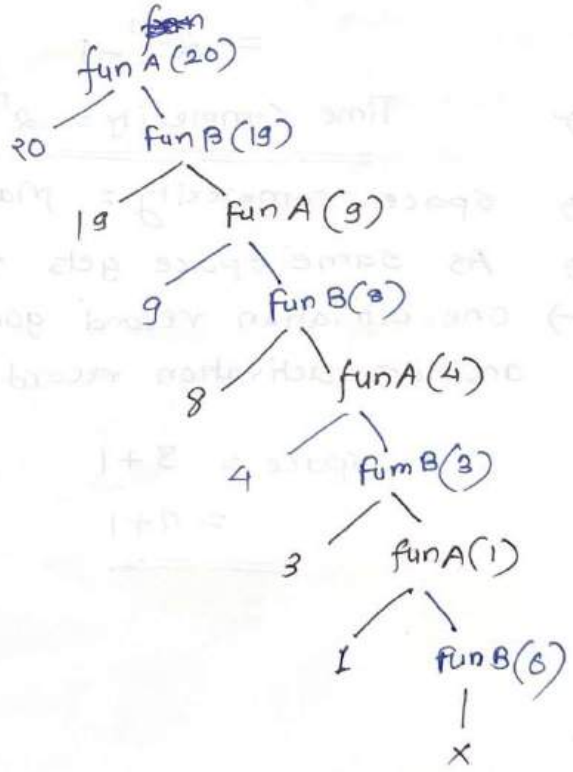
e.g: ①

```
Void A(n)
  if( )
  {
    ==
    B(n-1);
  }
Void B(n)
  if( )
  {
    ==
    A(n-3);
  }
```



②

```
void funA(int n)
{ if(n>0)
  {
    printf("%d",n);
    funB(n-1);
  }
}
void funB(int n)
{
  if(n>0)
  {
    printf("%d",n);
    funA(n/2);
  }
}
int main( )
{ int x=20;
  fun(20);
}
```



⑤ Nested Recursion

→ Recursion inside recursion is called as "nested Recursion"

e.g:

```
void fun(int n)
{
  if(  )
  {
    ____
    fun(fun(n-1));
  }
}
```

②

```
int fun(int n)
{
  if (n > 100)
    return n-10;
  else
    return fun(fun(n+11));
}
int main()
{ int x=96
  Fun(fun(n+11));
  fun(96);
}

int main()
{
  int x=
```

fun(a

Fun(96) = 91

↓

fun(fun(96+11)) 97 = fun(108)

107

fun(97)

↓

fun(fun(108)) 98 = Fun(108)

fun(98)

↓

fun(fun(109)) 99 = fun(109)

fun(99)

↓

fun(fun(110)) 100 = fun(110)

Fun(100)

↓

fun(fun(111)) 101 = fun(111)

fun(101)

↓

91

* Sum of Natural Number using Recursion

$Sum(n) = 1 + 2 + 3 + \cdots + (n-1) + n$

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

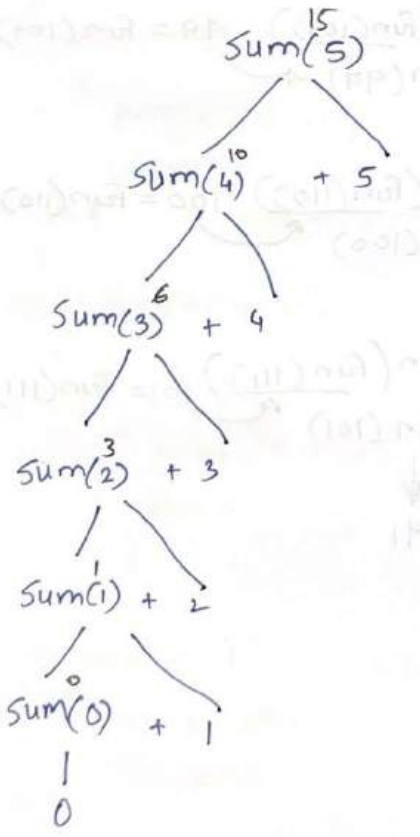
$$Sum(n) = \begin{cases} 0 & n = 0 \\ Sum(n-1) + n & n > 0 \end{cases}$$

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

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

③ Using loop
```
int sum(int n)   Time = O(n)
{ int i, s = 0;
  for(i=1; i<=n; i++)
     s=s+i ;
  return s;
}
```

1) return sum(n-1) + n



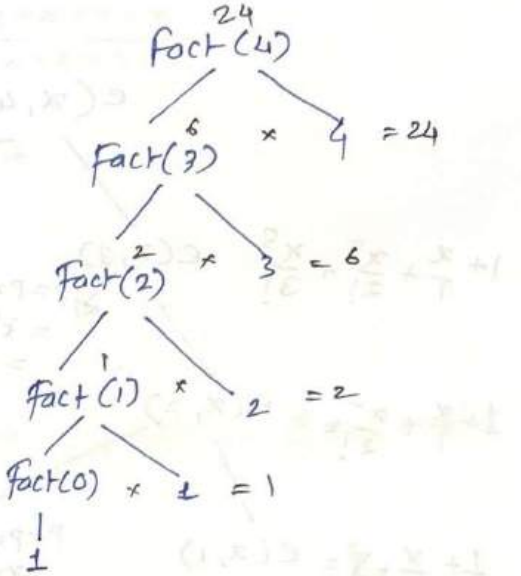
* Factorial using Recursion :

$fact(n) = 1 \times 2 \times 3 \times \ldots \times (n-1) \times n$

$fact(n) = fact(n-1) \times n$

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

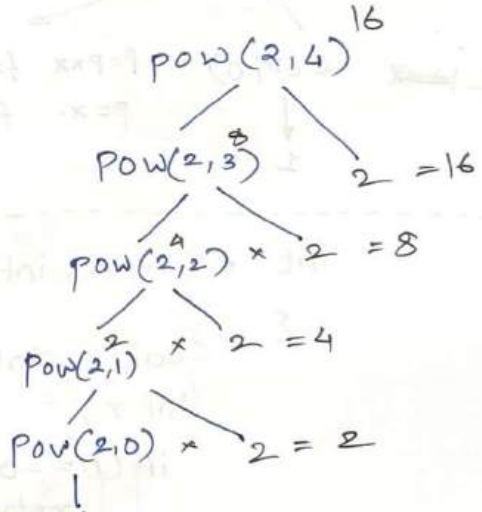
```
int fact (int n)
  { if (n==0)
      return 1;
    else
      return fact(n-1)*n;
  }
int main()
  {
    cout << fact(4);
  }
```



* Power using Recursion

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

f(2,4) $2^4 = 16$

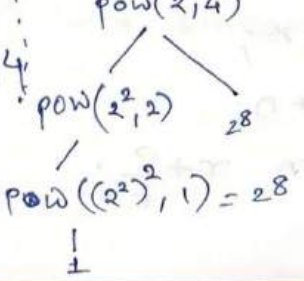


⇒ we can reduce number of Multiplication

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

$= (2 \times 2)^4$

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

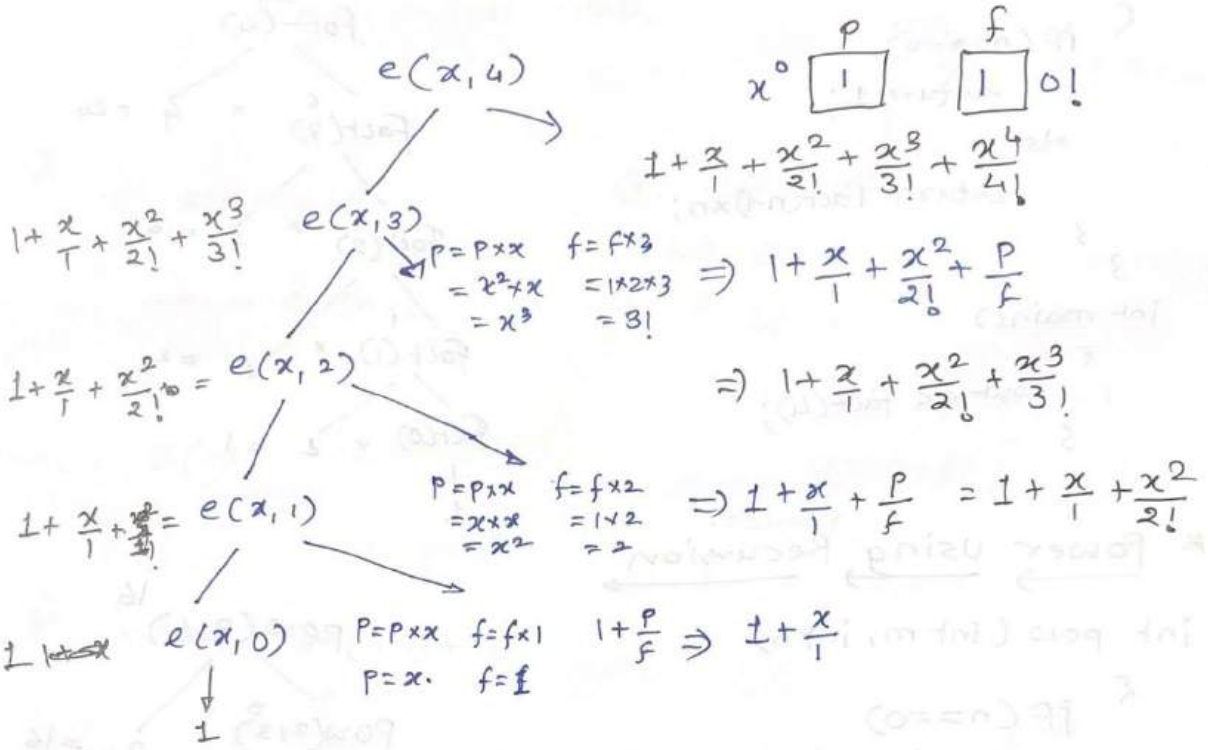



```
int pow (int m, int n)
  { if (n==0)
      return 1;
    if (n%2==0)
      return pow(m*m, n/2)
    else
      return m* pow(m*m, (n-1)/2)
```

* Taylor series using Recursion

$$e^x = 1 + \frac{x}{1} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \text{----} + n \text{ times.}$$

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



```
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:

① $e^x = 1 + \frac{x}{1} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$ n times.

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

$= 0 + 0 + \frac{x \times x}{2 \times 1} + \frac{x \times x \times x}{2 \times 2 \times 1} + \frac{x \times x \times x \times x}{4 \times 4 \times 4 \times 4}$

$= 0 + 0 + 2 + 4 + 6 + 8 + \cdots$

$= 2[1 + 2 + 3 + 4 + \cdots]$

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

$= n(n+1)$ Time $\rightarrow O(n^2)$ – Quadratic

②

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

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

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

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

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

$O(n) \rightarrow$ Linear

Iterative

Recursive.

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

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

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

---

* Fibonacci Series using Recursion

0, 1, 1, 2, 3, 5, 8, 13

F(n)	0	1	1	2	3	5	8
n	0	1	2	3	4	5	6

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

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

* Iterative way

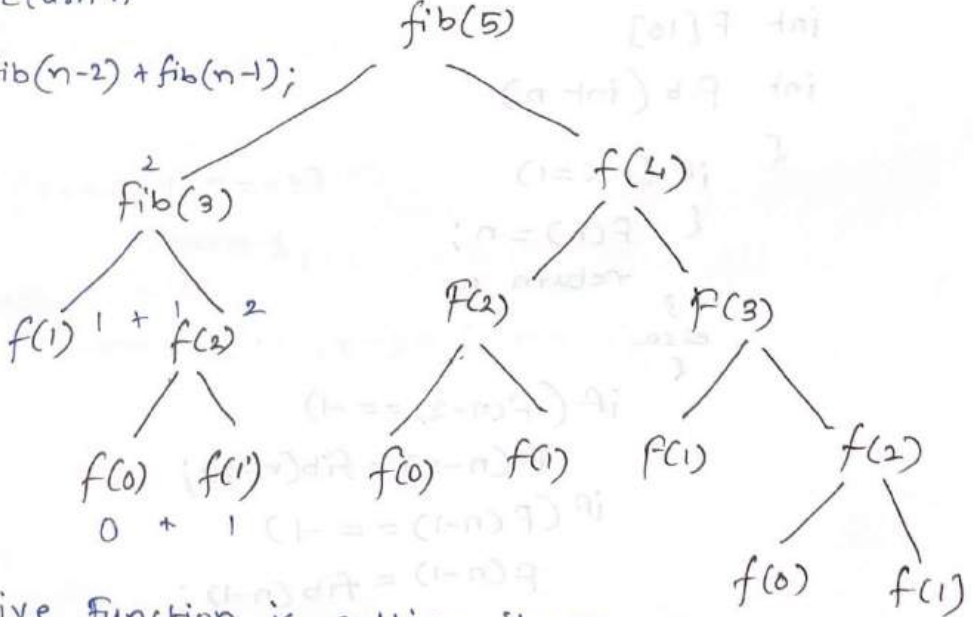
```
int fib(int n)
{
    int t0 = 0, t1 = 1, s, i;
    if (n <= 1)
        return n;
    for (i = 2; i <= n; i++)
    {
        s = t0 + t1;
        t0 = t1;
        t1 = s;
    }
    return s;
}
```

Time → O(n)

* Recursive way,

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

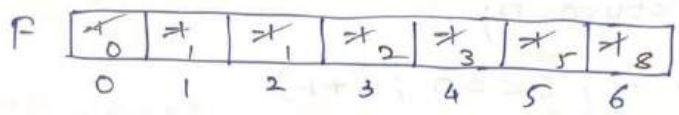
Time → O(2^n)

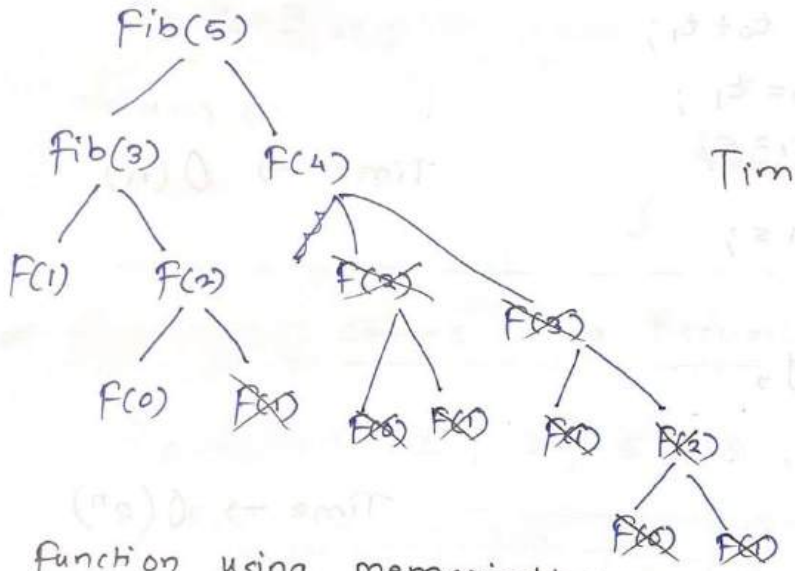


⇒ If recursive function is calling itself multiple times for same values, then such a recursive function is called "Excessive Recursion."

→ Now to Reduce Excessive calls. We can store the values in static or Global variable or Array.

Holding the result to avoid excessive calls., this approach is called as " Memorization".





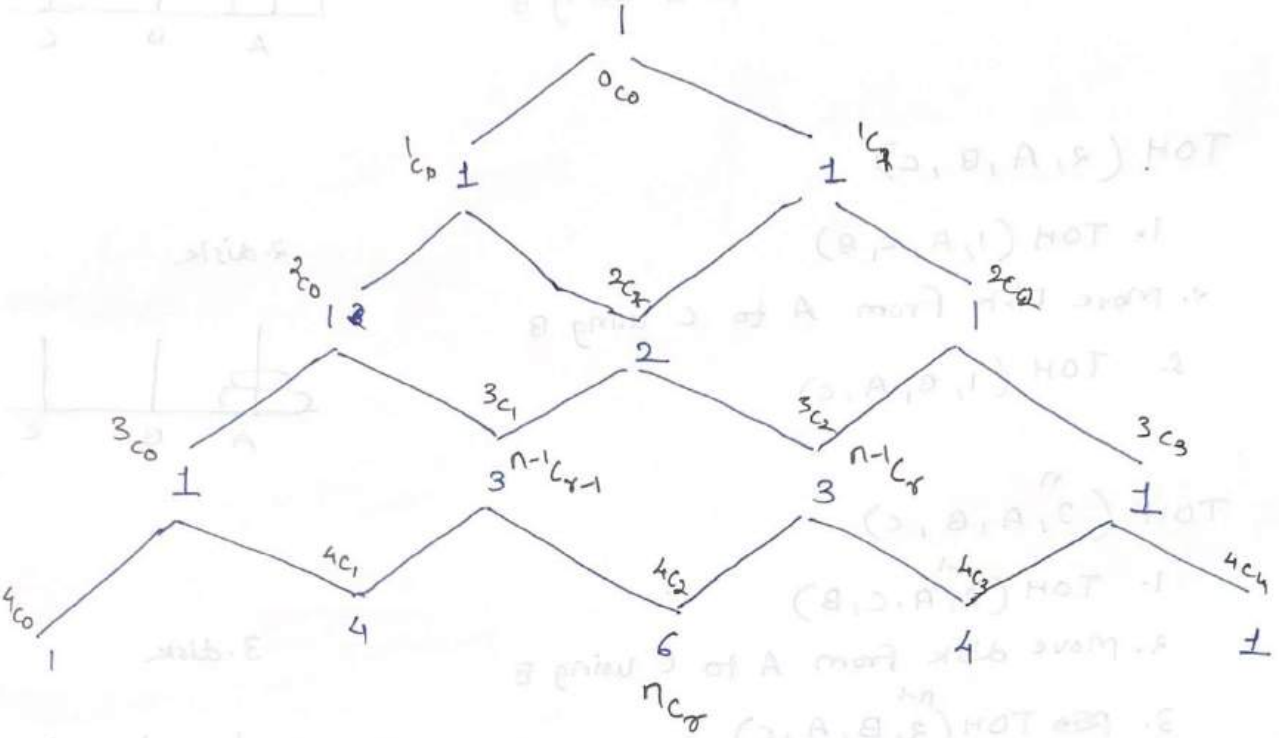
Time: O(n)

function using memorization.

```
int F[10]
int fib (int n)
 {  if (n<=1)
    {  F(n) = n;
       return n;
    }
   else
    {
       if ( F(n-2) == -1)
          F(n-2) = fib(n-2);
       if (F(n-1) == -1)
          F(n-1) = fib(n-1);
       return F(n-2) + F(n-1);
    }
 }
```

* Combination formula. $\longrightarrow\longrightarrow$ $^nC_r = \frac{n!}{r!(n-r)!}$

* Pascal's Triangle



$$\left[{}^nC_r = {}^{n-1}C_{r-1} + {}^{n-1}C_r \right]$$

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

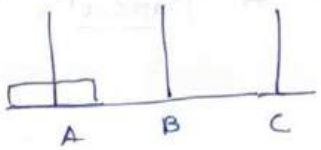
* Tower of Hanoi

No. of disk → 1, source → A, using → B, Destination → C

TOH (1, A, B, C)

Move Disk from A to C using B

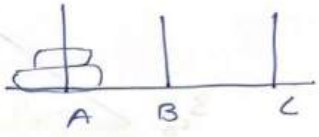
1. disk.



TOH (2, A, B, C)

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

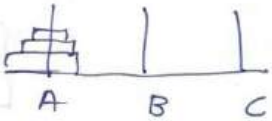
2. disk



TOH ($\overset{n}{3}$, A, B, C)

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

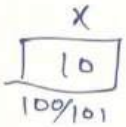
3. disk



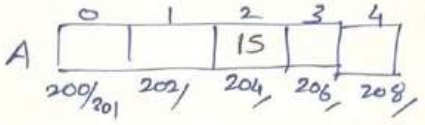
* Array

* Array ⇒ Array is a collection of similar data elements grouped under one name.
→ Arrays are vector variable.

Scalar → int x = 10



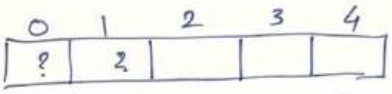
Vector → int A[5];



A[2] = 15;

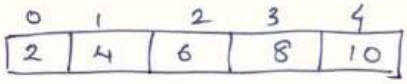
* Declaration of Array

① int A[5];

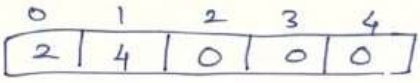


garbage values.

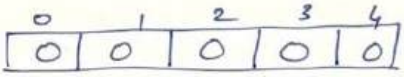
② int A[5] = {2, 4, 6, 8, 10}



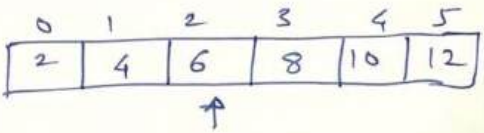
③ int A[5] = {2, 4};



④ int A[5] = {0};



⑤ int A[ ] = {2, 4, 6, 8, 10, 12}




```
int A[5] = {2, 5, 4, 9, 8}

for(i=0; i<5; i++)
    cout << i;
}
```

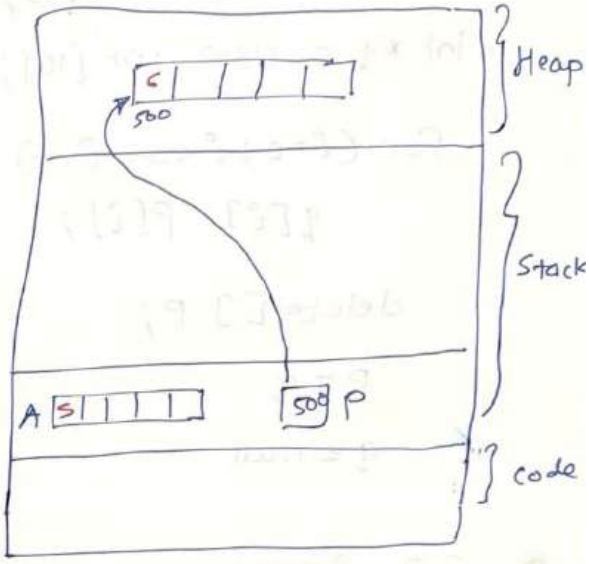
```
cout << A[2];
cout << 2[A];
cout << *(A+2);
```

* Static Vs Dynamic Array →

→ for static array, vector variable, memory for this array will be created inside stack.

→ In c++ size of array can be decided at the time of Run time.

```
Void main()
{
    int A[5];
    int n;
    cin >> n;
    int B[n];
    ---
    int *P;
    P = new int[5]          c++
    P = (int*)malloc(5*sizeof(int));   c
    delete []P;             c++
    free(P);                c
```




```
A[0] = 5;
P[0] = 6;
```

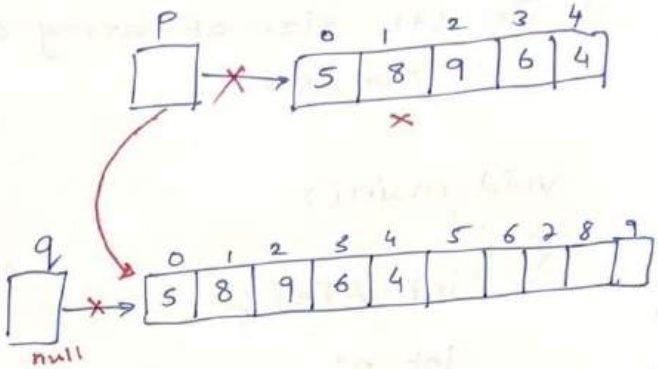
⇒ Whenever, an array is created, once the array of some size is created, then its size cannot be resize. (change).

⇒ In heap, it is possible to change by some alternative ways. but it is not possible in stack array at all.

* How to increase size of Array

→ We can not increase the size of existing array but this array can be shifted in bigger size array.

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



* 2D Array :

→ For 2D Array memory will be allocated as single dimension array only.

① `int A[3][4];`

A

	0	1	2	3
0	100	13	15	17
1	19	11	15	15
2	17	19	21	23

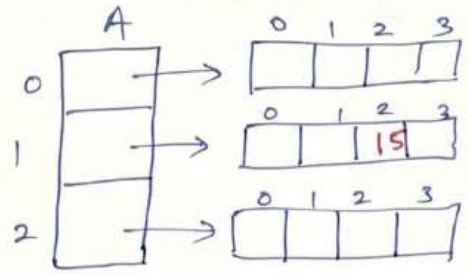
`A[1][2] = 15;`

int A[

`int A[3][4] = {{1,2,3}, {4,5,6}, {7,8,9}}`

② `int *A[3];`

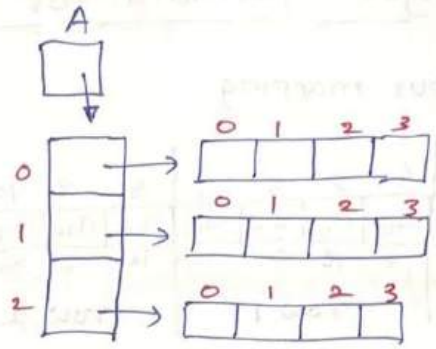
```
A[0] = new int[4];
A[1] = new int[4];
A[2] = new int[4];
```



`A[1][2] = 15;`

③

```
int **A;
A = new int*[3];
A[0] = new int[4];
A[1] = new int[4];
A[2] = new int[4];
```

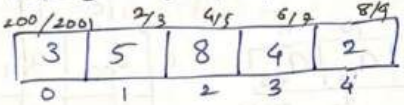


* To access 2D array elements:

```
for (i=0; i<3; i++)
{
    for(j=0; j<4; j++)
    {
        cout << A[i][j]
    }
}
```

* Arrays in Compilers

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



⇒ Address will be given at the run time. there is no address at the Compile time.

A[3] = 10;

Add(A[3]) = 200 + 3*2 = 206

Add(A[3]) = L_0 + 3*2.

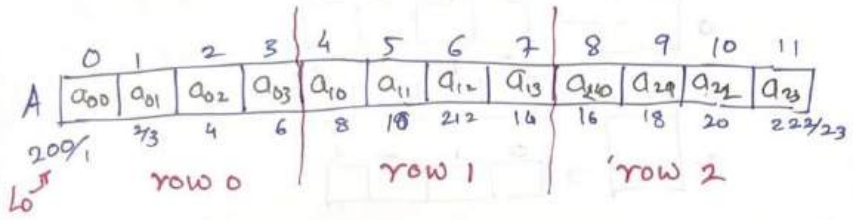
$$Add(A[i]) = L_0 + i \times w$$

L_0 : Base Address, i : index, w : size of Data type.

⇒ Compiler writes the formula for getting address at the compile time.
Actual address is available at the time of run time.

* Row Majour formula for 2D Arrays :

Row majour mapping



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

A[i][j]
m × n.

Add (A[1][2]) = 200 + [1×4 + 2] × 2

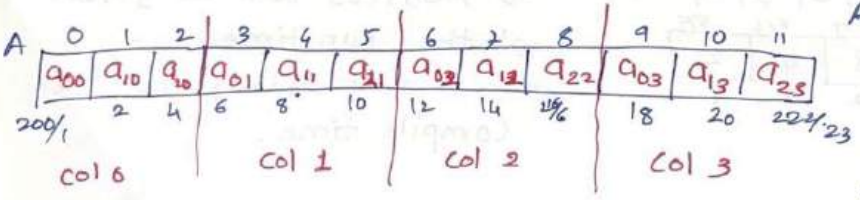
= 200 + 6×2 = 212

Add (A[2][3]) = 200 + [2×4 + 3] * 2 = 222

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

n → No of column

* Column Majour formula for 2D Array



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

int A[3][4]
m × n

Add (A[1][2]) = 200 + [2×3 + 1] * 2 = 214

Add (A[1][3]) = 200 + [3×3 + 1] × 2 = 220

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

m → No of row

w = size of data type.

C++ follows Row majour formula.

* Formulas for nD Array

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

Row major:

$$Add(A[i_1][i_2][i_3][i_4]) = L_0 + [i_1 \times d_2 \times d_3 \times d_4 + i_2 \times d_3 \times d_4 + i_3 \times d_4 + i_4] \times w$$

Column major:

$$Add(A[i_1][i_2][i_3][i_4]) = L_0 + [i_4 \times d_1 \times d_2 \times d_3 + i_3 \times d_1 \times d_2 + i_2 \times d_1 + i_1] \times w$$

* Formula for 3D Array

int A[i][m][n]

Row major:

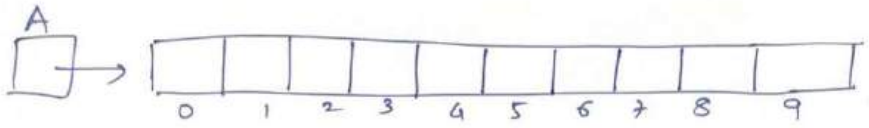
$$Addr(A[i][j][k]) = L_0 + [i \times m \times n + j \times n + k] \times w$$

Column major:

$$Addr(A[i][j][k]) = L_0 + [k \times l \times m + j \times l + i] \times w$$

Array ADT (Abstract Datatype)

⇒ Abstract data type means representation of data and the set of operations on the data.



size = 10
length = 0
(No elements)

① int A[10] ✓

②
```
int size;
Cin >> size
int A[size] ✗
```
It will give error. We can't use variable for this type of array declaration. Hense we need to use pointer to create variable size array.

```
int *A;
A = new int[size] ✓
```

* Operations on Array :

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()

```
#include <iostream>
using namespace std;

Struct Array
{
    int *A;
    int size;
    int length;
};

Void Display (struct Array arr)
{
    cout << endl << "The array:" <<endl;
    for (int i=0 ; i<< arr.length; i++)
        cout << arr.A[i] << " ";
}

int main()
{
    struct Array arr;
    cout << " Enter the size of Array= ";
    cin >> arr.size;

    arr.A = new int [arr.size];
    arr.length = 0;

    int n;
    cout << " Enter no. of elements = ";
    cin >> n;
    arr.length = n;

    cout << "Enter " << n << "  all elements: " <<endl;
    for (int i=0 ; i<n ; i++)
        cin >> arr.A[i];

    Display(arr);
}
```

* Adding / Insertion of element. (Append) (Insert)

Append is adding at the last

Insert is inserting at given index.

e.g.

```
struct Array
{
  int A[20];
  int size;
  int length;
};
void Append (struct Array *arr, int x)
{
    if(arr->length < arr->size)
      {
        arr->A[arr->length]=x;
        arr->length++;
      }
}
Void Insert (struct Array *arr, int index, int x)
{
  if (arr->length < arr->size)
    {
      arr->length++;
      for(int i = arr->length; i > index ; i--)
          arr->A[i] = arr->A[i-1];
    }
    arr->A[index] = x;
Void Display (struct Array arr)
  {
    cout << endl << " The array: " <<endl;
    for (int i=0 ; i<arr.length ; i++)
        cout << arr.A[i] << " " ;
  }
int main()
  {
    struct Array arr = {{2,4,6,7,9}, 20 ,5} ;
     Append (&arr, 12);
     Display (arr);
     Insert (&arr, 4, 16);
     Display (arr);
  }
```

```
void Delete (struct Array *arr, int index)
{
    for (int i = index; i < arr->length; i++)
    {
        arr->A[i] = arr->A[i+1];
    }
    arr->length--;
}
```

* Linear Search →

→ for performing search, the elements must be unique.

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

Succesful → key = 5

Time complexity → Best → O(1)

Time complexity → worst → O(n)

Unsuccesful → key = 12 → O(n)

Avg → O(n).

```
int search (int arr[], int size, int k)
{
    for (int i=0; i < size; i++)
    {
        if (arr[i] == k)
            return i;
    }
    return -1;
}
```

* There are two methods to improve linear search

① Transposition

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

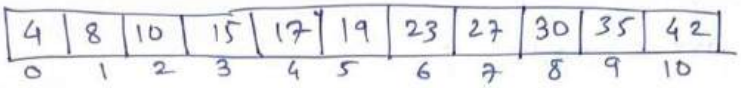
② Move to front/Head.

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

* Binary Search

Pre requisite: To perform binary search, elements must be sorted.

Size = 11
length = 11




```
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;
}
```

```
Recu.BinSearch (l, h, key)
{
    if (l <= h)
    {
        mid = ⌈(l+h)/2⌉;
        if (key == A[mid])
            return mid;
        else if (key < A[mid])
            return Rec.Binsearch(l, mid-1, key);
        else
            return Rec.Binsearch(mid+1, h, key);
    }
    return -1;
}
```

* Strings

① ASCII Codes: character set is the set of characters that are supported by programming language like C/C++ or any other language.

0-127

→ We define some set of numbers as character.

→ 0-127 : total 128 } 1 byte memory is used.
7 bits $2^7 = 128$

A-65	a=97	0-48
B-66	b=98	1-49
C-67	c=99	2-50
⋮	⋮	⋮
Z=90	z=122	9=57

ASCII codes are used to represent "English"

other language are represented by "Unicode"

→ Unicode takes 2 byte of memory.

⇒ Character Variable:

→ char takes 1 byte of memory.

```
char temp;
temp = 'A';   ✓
temp = 'AB';  x
temp = A;     x
temp = "A";   x
```

temp: A (65)

```
printf("%c", temp)  — A
printf("%d", temp)  — 65
cout << temp;       — A
```

⇒ character Array:

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

x

A	B	C	D	E
0	1	2	3	4

```
char x[] = {'A','B','C','D','E'};
char x[5] = {65,66,67,68,69};
char x[5] = {'A','B'};
```

x

A	B	0	0	0
0	1	2	3	4

string

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

name	J	o	h	n	\0	o	o	o	o	o

→ To know the length of string or where string is ending, slash zero (\0) is used, this is null symbol.
it is also called as "string delimeter" or "string terminator"

to make it string

name	J	o	h	n	\0	o	o	o

Declaration of strings

① `char name[10] = {'J', 'o', 'h', 'n', '\0'};`

② `char name[] = {'J', 'o', 'h', 'n', '\0'};`

③ `char name[] = "John";`

To print

```
prinf ("%s", name);
sconf ("%s", name);
```

* This can read only one word.
* Not multiple words or statement.

* Finding length of a string:

```
Void stringLength (char str[])
{
  int i;
  for (i=0; str[i] != '\0'; i++)
    {
    }
  cout<< "The length is " << i << endl;
}
```

* Changing Case of string:

```
int main()
{
  char A
```

```
Void ChangeCase (char str[])
{
  for (int i=0; str[i] != '\0'; i++).
  {
    if (str[i] > 65 && str[i] <= 90)
      str[i] = str[i] + 32

    else if (str[i] >= 'a' && str[i] <= 'z')
      str[i] = str[i] - 32;

  }
  cout << str;
}
```

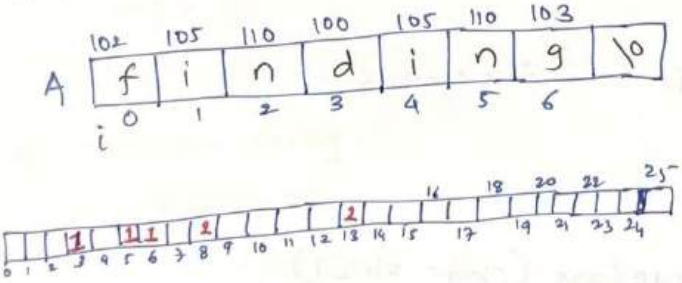
* Comparing strings & Palindrome:

```
void compareStrings (char str1[], char str2[])
{
  int i;
  for (i=0 ; str1[i] != '\0' && str2[i] != '\0'; i++)
  {
      if (str1[i] != str2[i])
      {
        cout << "strings are not equal";
        break;
      }
      else if (str1[i] == '\0' && str2[i] == '\0')
      {
        cout << "strings are equal;
      }
  }
```

⇒ Palindrome is if strings is same in reverse order as well.

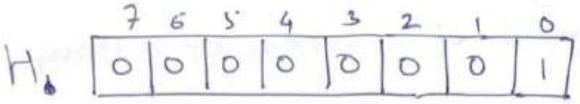
e.g: madam, A22O.

* finding Duplicates in a string:

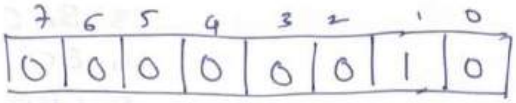



```
int main()
{ char A[] = "finding"
  int H[26], i;
for(i=0 ; A[i] != '\0' ; i++)
{
  H[A[i] - 97]+ = 1;
}
for (i=0; i<26 ; i++)
{
  if (H[i]>1)
  {
    printf ("%c", i+97);
    printf ("%d", H[i]);
  }
}
```

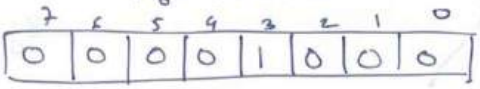
* Left shift



H = H<< 1 Left chif by 1.



H_0 = H<< 3



* bitwise 'and' & 'or' operation:

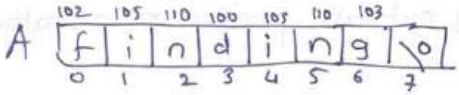
a=10 → 1 0 1 0
b=6 0 1 1 0

a&b 0 0 1 0 - 2

1 0 1 0
0 1 1 0

a|b 1 1 1 0 - 14

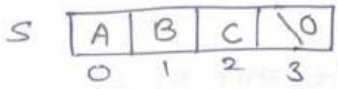
* Finding Duplicates in a string using Bitwise operations.




```
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;
  }
}
```

* Permutation of a string:

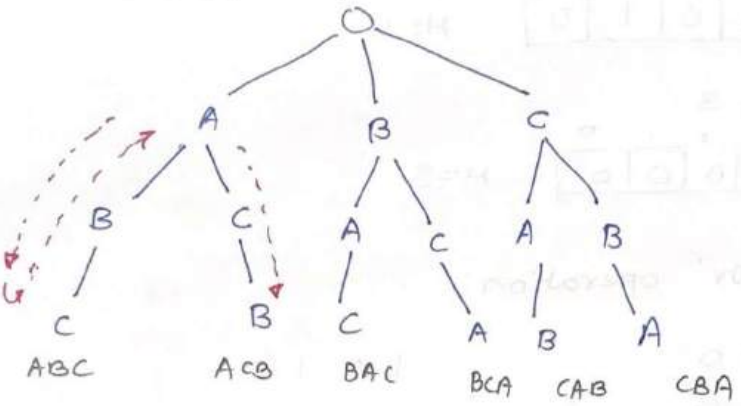
Finding all possible arrangement of a letters of a string.



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

$3!$ $\cap!$

State Space Tree



⇒ State space tree: The tree representing all various possible solutions..
and while finding solutions, go back and come again this approach is called "Back tracking

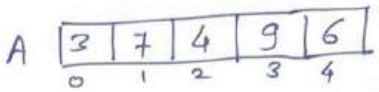
⇒ Brute force: Brute force means finding out all possible permutations.

Matrices

* Diagonal Matrix

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

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




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

⇒ To save space and time we can store the elements in 1D array instead of 2D array as the other elements are zero only.

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

* Diagonal Matrix

```
class Diagonal
{
  private:
    int *A; int n;
  public:
    Diagonal()
    { this->n = 2;
      A = new int[2];
    }
    Diagonal(int n)
    { this->n = n;
      A = new int[n];
    }
    ~Diagonal()
    { delete []A;
    }
    void Set(int i, int j, int x)
    int Get(int i, int j);
    void Display();
};
```

```
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];
   return 0;
}
Void Diagonal::Display()
{
  for(int i=1; i<=n; i++)
  {  for(int j=1; j<=n; j++)
     {  if(i==j)
           cout<<A[i-1]<<" ";
        else
           cout<<" 0 ";
     }
     cout<<endl;
  }
}
```

```
int main()
{  int d;
   cout<<"Enter Dimensions";
   cin>>d;
   Diagonal dm(d);
   int x;
   cout<<"Enter All Elements";
   for(int i=1; i<=d; i++)
   {  for(int j=1; j<=d; j++)
      {  cin>>x;
         dm.Set(i,j,x);
      }
   }
   dm.Display();
   return 0;
}
```

* Lower Triangular Matrix

$j \longrightarrow$

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

($i \downarrow$)

$M[i,j] = 0$ if $i < j$

$M[i,j] =$ non zero if $i >= j$

Non zero $= 1+2+3+4+5 = 15$

$= 1+2+3+4+\ldots+n = \frac{n(n+1)}{2}$

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

$\Rightarrow$ So we need to store $\frac{n(n+1)}{2}$ elements for $n \times n$ triangular matrix.

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
row 1	row 2		row 3			row 4				row 5				

Index $(A[4][3]) = [1+2+3]+2 = 8$

Index $(A[5][4]) = [1+2+3+4]+3 = 13$

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

Column-Major formula:

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

Index $(A[4][4]) = [5+4+3]+0 = 12$

Index $(A[5][4]) = [5+4+3]+1 = 13$

Index $(A[5][3]) = [5+4]+2 = 11$

Index $(A[i][j]) = [n + n-1 + n-2 + \ldots + n-(j-2)] + (i-j)$

$= [n(j-1) - [1+2+3+\ldots+j-2]] + (i-j)$

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

* Symmetric Matrix

$j \rightarrow$

$i \downarrow$	1	2	3	4	5
1	2	2	2	2	2
2	2	3	3	3	3
3	2	3	4	4	4
4	2	3	4	5	5
5	2	3	4	5	6

$M =$

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

It can be stored using

① lower Triangular matrix

or ② Upper Triangular Matrix.

* Tridiagonal Matrix

$j \rightarrow$

$i \downarrow$	1	2	3	4	5
1	a_{11}	a_{12}	0	0	0
2	a_{21}	a_{22}	a_{23}	0	0
3	0	a_{32}	a_{33}	a_{34}	0
4	0	0	a_{43}	a_{44}	a_{45}
5	0	0	0	a_{54}	a_{55}

$M =$

Main Diagonal $i-j=0$

lower $i-j=1$

upper $i-j=-1$

$|i-j| \leq 1$

Defn $\begin{cases} M[i,j] = \text{non-zero} & \text{if } |i-j| \leq 1 \\ M[i,j] = 0 & \text{if } |i-j| > 1. \end{cases}$

A	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

Lower Diag. | Main Diagonal | Upper Diagonal.

No. of Non zero elements

$= 5+4+4$

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

$= 3n-2$

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

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

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

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

* 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

5×5

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

Elements needs to store $= n + n - 1$

$= 2n - 1$

$= 2(5) - 1 = 9$

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

row | column

Index (A[i][j])

Case 1: if $i \leq j$

Index $= j - i$

$A[2][4] = 4 - 2 = 2$

$A[3][4] = 4 - 3 = 1$

Case 2: if $i > j$

Index $= n + i - j - 1$

Section 10 : Sparse Matrix and Polynomial Representation

* Sparse Matrix Representation :

Sparse matrix is a matrix in which there are more number of zero elements.

→ To store stor matrix to save space & time there are two approaches.

1. Co-ordinate List / 3 Column Representation
2. Compressed sparse row.

* 3 Column Representation :

row	column	element
8	9	8 (Non zero ele)
1	8	3
2	3	8
2	6	10
4	1	4
6	3	2
7	4	6
8	2	9
8	5	5

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

8 × 9

* Compressed sparse rows

element matrix A [3, 8, 10, 4, 2, 6, 9, 5]

Cummulitive row matrix IA [0, 1, 3, 3, 4, 4, 5, 6, 8]
0 1 2 3 4 5 6 7 8

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

* Addition of sparse Matrices

→ for addition of two matrices there dimensions must be same.

$A =$

	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

5×6

A (i →)

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

$B =$

	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

5×6

B (j →)

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

$A+B = C$

0	0	0	6	0	0
0	10	0	0	5	0
0	2	2	5	0	7
0	0	0	9	0	0
12	0	0	0	0	0

5×6

C

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
9	6	10	5	2	2	5	7	9	12

```
# include <iostreame>.
Using namespace std;

class Element
{
  public:
      int i;
      int j;
      int x;
  };

class Sparse
{
  private:
     int m;  int n;  int num ;  Element *ele;
  public:
    Sparse (int m ,int n, int num)
    {    this -> m =m;
         this-> n = n;
         this -> num= num;
         ele= new Element [this -> num];
    }
   ~ Sparse ()
    {
      delete [] ele;
    }
  void read ()
   {
     cout << " Enter non-zero elements";
     for (int i=0; i<num ; i++)
           cin >> ele[i].i >> ele[i].j >> ele[i].x ;
   }
   void display()
    {    int K=0
         for (int i=0 ; i<m; i++)
         {    for(int j=0; j<n ;j++)
              {   if (ele[K].i=i && ele[K].j=j)
                       cout << ele [K++].x << "  ".
                  else cout <<" 0 ";              cout<<endl.
};        }        }
```

```
Struct Sparse*  add(struct Sparse *s1, struct Sparse *s2)
{
   struct Sparse *sum;
   int i, j, k;
   i = j = k = 0;
   if (s1->n != s2->n && s1->m != s2->m)
        return NULL;
   Sparse *sum = new Sparse(s1->m, s1->n, s1->num + s2->num)
   i = j = k = 0;
   while (i < s1->num && j < s2->num)
    {
       if(s1->ele[i].i < s2->ele[j].i)
            sum->ele[k++] = s1->ele[i++];
       else if (s1->ele[i].i > s2->ele[j].i)
            sum->ele[k++] = s2->ele[j++];
       else
        {
           if (s1->ele[i].j < s2->ele[j].j)
                sum->ele[k++] = s1->ele[i++];
           else if (s1->ele[i].j > s2->ele[j].j)
                sum->ele[k++] = s2->ele[j++];
           else
            {
                sum->ele[k] = s1->ele[i];
                sum->ele[k++].x = s1->ele[i++].x + s2->ele[j++
                                                          .x
            }
    }
int main()
  {
     struct Sparse s1(5,5,5);
            Sparse s2(5,5,5);
        Sparse s3 = add(&s1, &s2);
  }
```

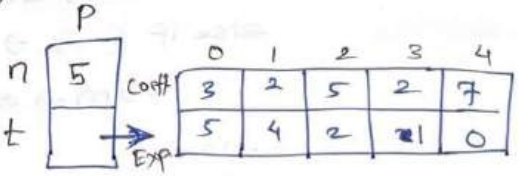
* Polynomial Representation

Polynomial is a collection of terms, Each term is having coefficient, exponent and variable.

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

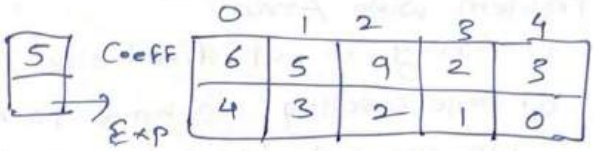
(Exp, coeff, variable)

```
struct Term
{
  int coeffi;
  int Exp;
}
  struct Poly
  {
    int n;
    struct Term *t;
  }
int main()
{
    struct Poly P;
    printf("No. of non-zero terms")
    scanf("%d", &P.n);
    P.t = new Term[P.n];
    printf("Enter Polynomial Terms")
    for (i=0; i<P.n; i++)
      {
        printf("Term no. %d", i++)
        scanf("%d %d", &P.t[i].coeffi, &P.t[i].Exp);
    }
```

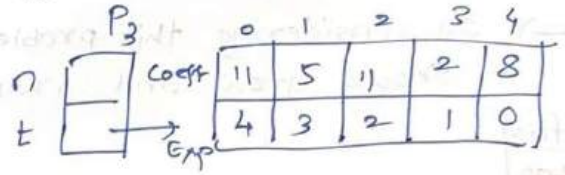


$P_1(x) = 5x^4 + 2x^2 + 5$

$P_2(x) = 6x^4 + 5x^3 + 9x^2 + 2x + 3$




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

while (i < P1.n && j < P2.n)
    if(P1.t[i].Exp > P2.t[j].Exp)
    { P3.t[k] = P1.t[i];
        k++ ;
        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;
```

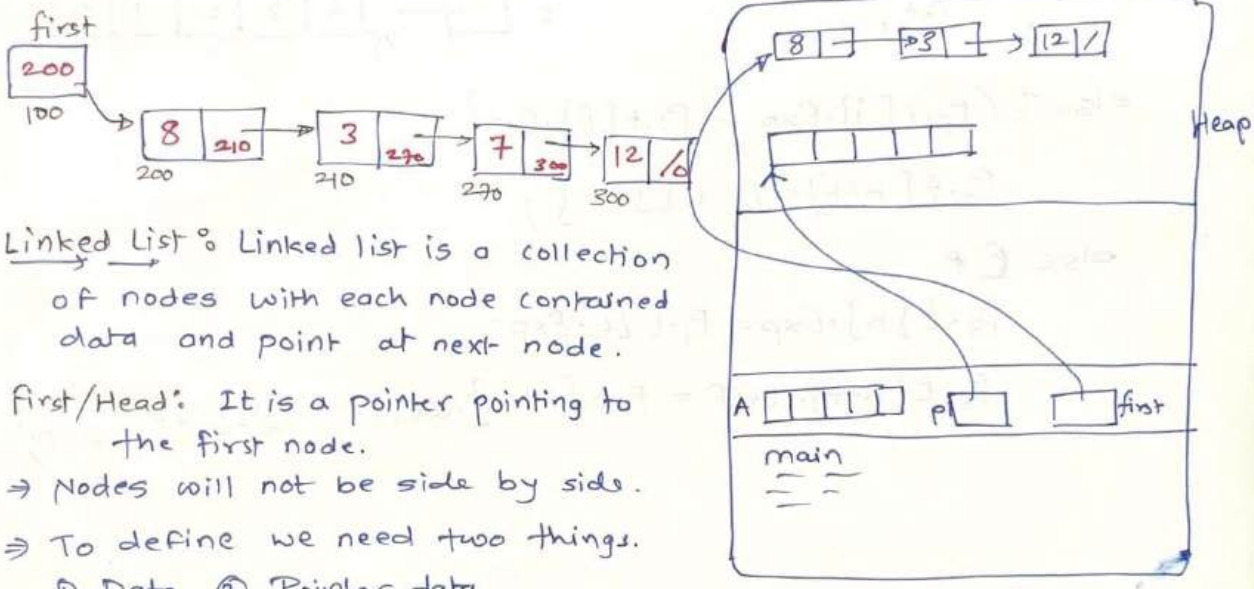


Section 11: Linked List

⇒ Problem with Array
① Array is of fixed size
② While creating we have to mention the size and later we can not increase or decrease the size of array.

⇒ We don't know, during runtime how many spaces we require. we may run out of space or can wast space if array initialy decleared of bigger size thon required.
→ This is like a bench.

⇒ So considering this proble we need data structure which should grow and reduce in size.



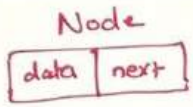
Linked List: Linked list is a collection of nodes with each node contained data and point at next node.

first/Head: It is a pointer pointing to the first node.

⇒ Nodes will not be side by sides.

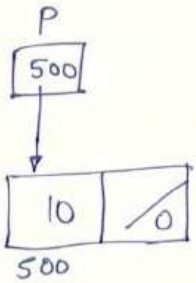
⇒ To define we need two things.
① Data ② Pointer data.

⇒ This stucture is called "Self Referential Structure".



structure of Node

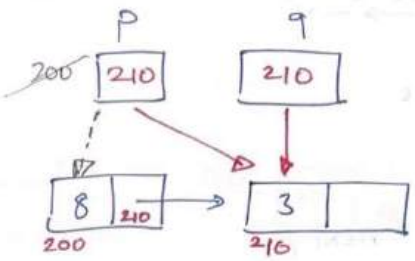
```
struct Node *P;
P= new Node;
P→data = 10;
P→ next = 0;
```




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

① struct Node *p, *q

1. q = p;
2. q = p → next
3. p = p → next;



② struct Node *p = NULL;

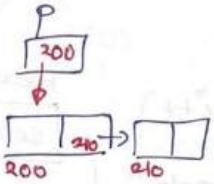
1. if (p == NULL)
2. if (p == 0)
3. if (!p)

→ True if NULL

So this are three conditions to check ~~if this p is pointing to any Node or not~~. if this pointer p is not pointing to any node.

③ 1. if (p != NULL)
2. if (p != 0)
3. if (p)

→



If p is pointing to any node then this conditions will be true.

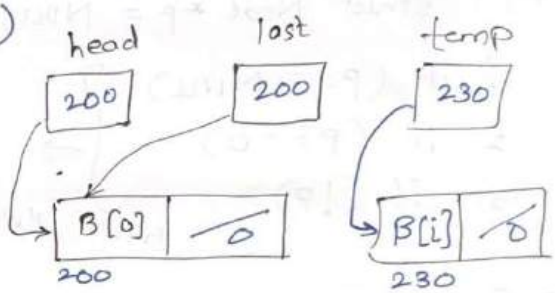
④ if (p → next == NULL) → so this p is pointing to last Node.
and vise versa for if (p → next != NULL)

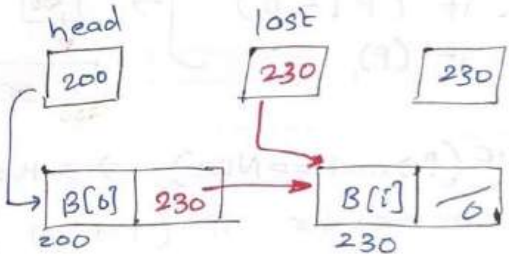
* Display Linked List:

* Display linked list :

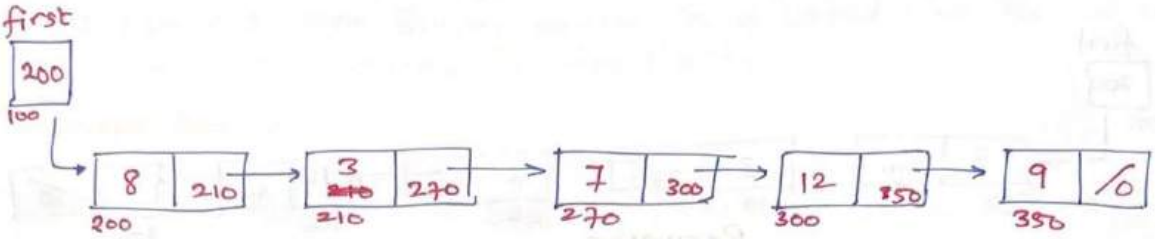
Own

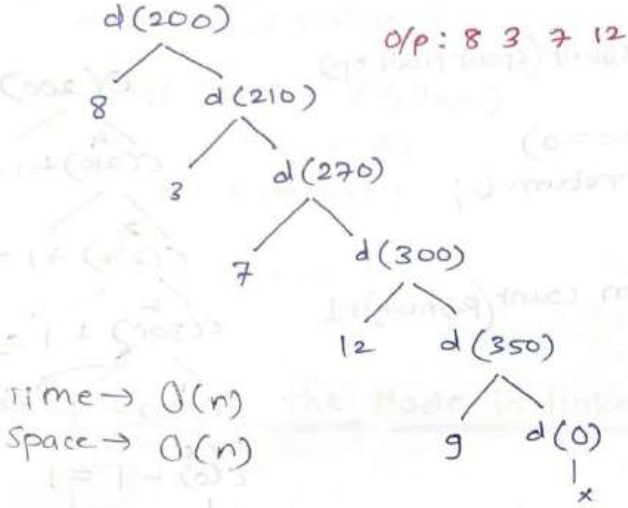
```
struct Node
{
  int data;
  struct Node * next;
};

struct Node * create (int B[], int size)
{
  Node * head = new head;
  head → data = B[0];
  head → next = nullptr;

  Node * last;
  last = head;
  for (int i = 1; i <= size; i++)
  {
    Node * temp = new Node;
    temp → data = B[i];
    temp → next = nullptr;

    last → next = temp;
    last = temp;
  }
  return head;
}

void Display (struct Node *A)
{
  Node * temp = A;
  while (temp → next != nullptr)
  {
    cout << " " << temp → data.
    temp = temp → next;
  }
}

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





* Recursive Display of Linked List





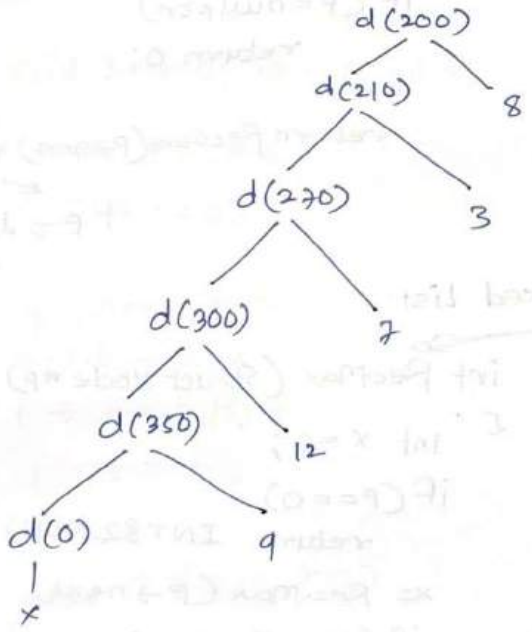
O/p: 8 3 7 12 9

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

Display (first)

Time → O(n)
Space → O(n)

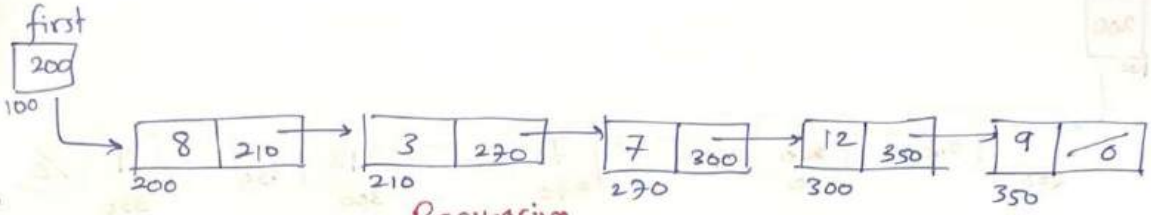
⇒ If any procedure or recursive function is written for traversing a linked list just once, then
It will make (n+1) call for entire linked list and also the stack size will be (n+1). So i.e. it is order of 'n'.




```
if (p != NULL)
{
    Display (p->next);
    Printf ("%d", p->data.
```

O/p: 9 12 7 3 8

* Counting Nodes in a Linked List

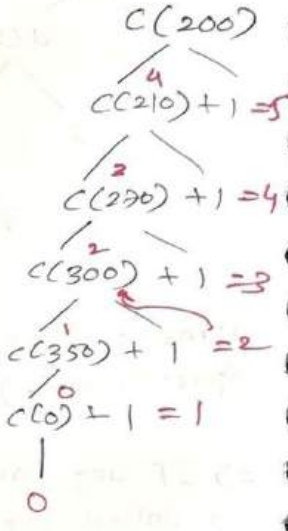


loop

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

Recursive

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



* Counting Sum of Nodes in Linked List

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

```
int Recsum (struct Node *p)
{
  if (p=nullptr)
      return 0;
  else
  return Recsum(p->next) + p->data
                          ← data
  + p->data
```

* Maximum Element in a Linked List

```
int Max (struct Node *p)
{
  int max = INT32_MIN;
  while (p)
  {
     if (p->data > max)
         max = p->data;
     p= p->next;
  }
  return max;
}
```

```
int Rec.Max (struct Node *p)
{ int x=0;
  if (p==0)
     return INT32_MIN;
  x= Rec.Max (p->next)
  if (x > p->data)
     return x;
  else
     return p->data;
}
```

* Searching in a linked list:

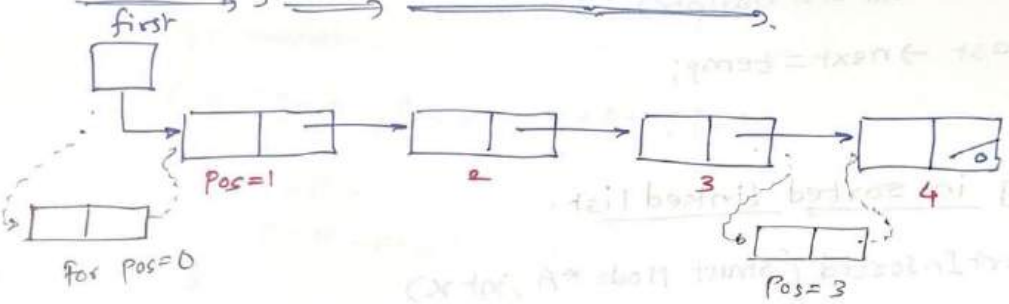
→ We can not used Binary search in a linked list as we can not reach the middle of the list.

Linear Search

```
Node * Search(Node *A, int key)
{
  While (A != nullptr)
  {
    if (key == A->data)
      return A;
    A = A->next;
  }
  return nullptr;
}
```

```
Node* RecSearch(Node *A, int key)
{
  if (A == nullptr)
    return nullptr;
  if (key == A->data)
    return (A);
  return RecSearch(A->next, key);
}
```

* Inserting the Node in linked List:

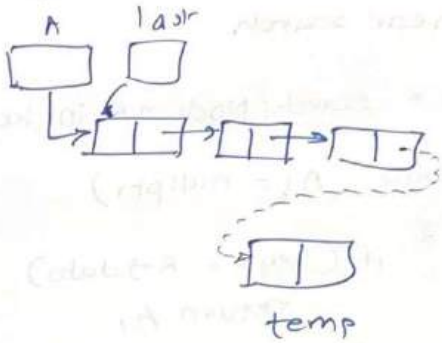



```
Void Insert(int pos, int x)
{
  Node *t, *q;
  if (pos == 0)
  {
    t = new Node;
    t->data = x.
    t->next = first
    first = t;
  }
  else if (pos > 0)
  {
    p = first;
    for (i=0; i < pos-1 && p; i++)
      p = p->next;
  }
```

```
  if (p)
  {
    t = new Node;
    t->data = x;
    t->next = p->next;
    p->next = t;
  }
}
```

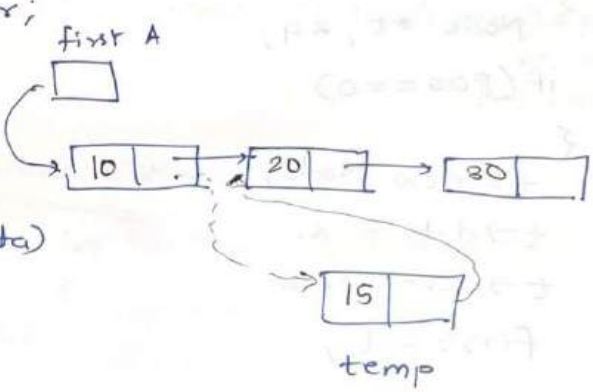
* Inserting at Last in Linked List.

```
Void Insert Last (int x, struct Node *A)
{
    Node *t = new Node;
    struct Node * last = nullptr;
    t -> data = x;
    t -> next = nullptr;
    last = A;
    if (A == nullptr)
        A = last = temp;
    While (last -> next != nullptr)
    {
        last = last -> next;
    }
    if (last -> next == nullptr)
        last -> next = temp;
}
```

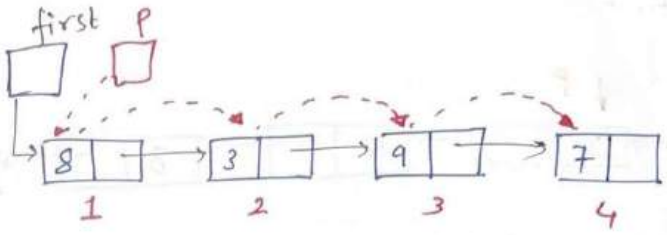


* Inserting in sorted Linked list.

```
Void InsertInsorted (Struct Node *A, int x)
{
    Node *temp = new Node;
    Struct Node *p = nullptr;
    P = A;
    temp -> data = x;
    temp -> next = nullptr;
    While (P -> data < temp -> data)
    {
        Q = P;
        P = P -> next;
    }
    Q -> next = temp;
    temp -> next = P;
}
```

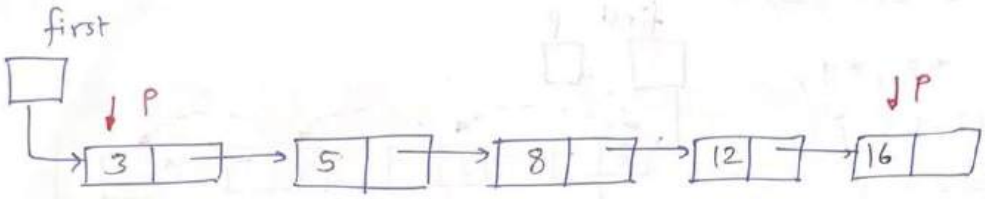


* Deleting from Linked List:




```
int Delete (int pos)
{
   Node *p, *q;
   int x= -1; i;
 if (pos==1)
{
  x= first->data;
   P= first;
  first = first->next;
   delete P;
 }
  else
   {
      P= first;
       q= nullptr;
      for (i=0; i<pos-1 && p; i++)
       {
           q=P;
           P= P->next;
        }
        if (P)
        {
           q->next = P->next;
           x= p->next;
           delete P;
         }
     }
     return x
 }
```

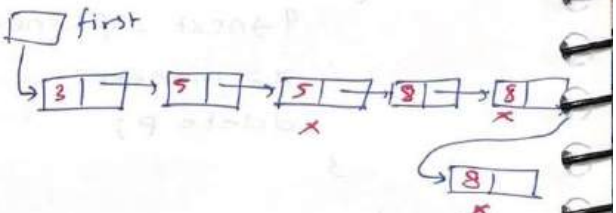
* Check if a linked list is Sorted or Not :




```
int isSorted (struct Node *p)
{
    int x = -65536;
    while (p != nullptr)
    {
        if (p->data < x)
            return 0;
        x = p->data;
        p = p->next;
    }
    return 1;
}
```

* Remove Duplicates from Sorted Linked list :

```
Void RemoveDuplicate (struct Node *A)
{
    Node *p = A;
    Node *q = A->next;
    while (q != nullptr)
    {
        if (p->data != q->data)
        {
            p = q;
            q = q->next;
        }
        else
        {
            p->next = q->next;
            delete q;
            q = p->next;
        }
    }
```



Reversing in a linked list:

① Reversing by elements.

```
Void Reverse1 (Struct Node *A)
{
   Node *first = A;
   Node *p = first;
  int arr[10]; int i=0;
 while (p != nullptr)
 {  arr [i] = p->data;
    p= p->next;
     i++;
  }
    p= first;
    i--;
   while (p != nullptr)
  {
     p->data = arr[i];
     P = P->next
     i--;
  }
}
```

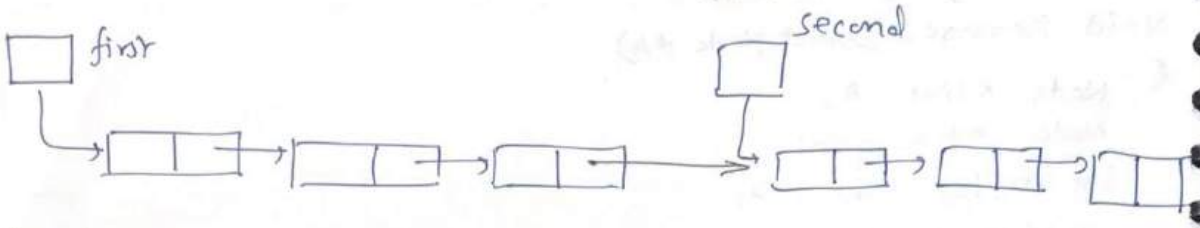
② Reversing by links

```
Node * Reverse2 (Struct Node*A)
 {
    Node * first= A;
    Node * p = first;
    int arr[10];
    int i=0;
   Node* q , r = nullptr;
  while (p != nullptr)
  {
     r=q;
     q=p;
     p= p->next;
     q=>next = r;
   }
    first = q;
    return first;
  }
```

③ Reversing by recurssive functn.

```
Void{Reverse3(Struct Node *q, Node *p)
 {  if (p)
    {
      Reverse3 (p, p->next)
      p->next = q;
    }
   else
       first = q;
 }
```

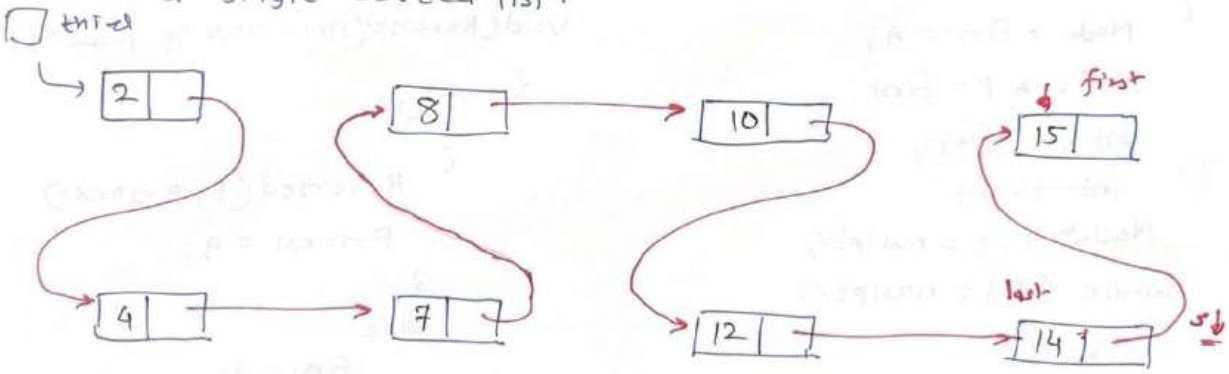
* Concatenating 2 Linked lists:




```
Struct Node * Concatenate (Struct Node *A , struct Node *B)
{
    Node *  head = A;
    while (A->next != nullptr)
    {
        A = A->next;
    }
    A->next = B;
    B = nullptr;
    return head;
}
```

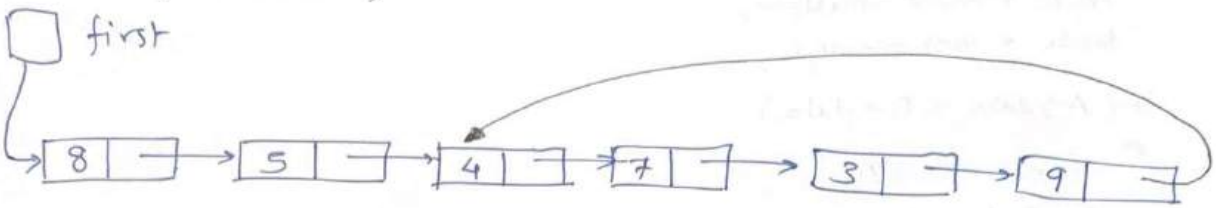
* Merging two Linked list;

Merging is a process of combining two sorted list into a single sorted list.

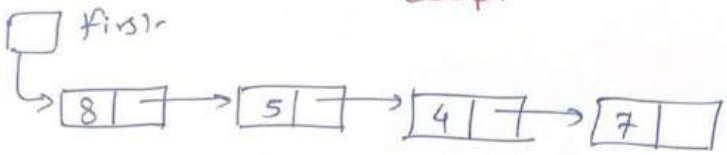


```
struct Node* SortedMerge(struct Node* A, struct Node* B)
{
    Node *third = nullptr;
    Node * last = nullptr;
  if (A->data < B->data)
   {
     third = last = A;
     A = A->next;
     last->next = nullptr;
   }
  else
    {
      third = last = B;
      B = B->next;
      last->next = nullptr;
    }

  while (A && B)
  {
        if (A->data < B->data)
       {
          last->next = A;
           last = A;
           A = A->next;
           last->next = nullptr;
        }
        else
        {
           last->next = B;
           last = B;
           B = B->next;
           last->next = nullptr;
        }
   }

    if (A)
     { last->next = A;
      }
     else
      { last->next = B;
       }

    return head;
}
```

* Check for Loop in a linked list:



Loop.



Linear

```
int isloop (struct Node *f)
{
    struct Node *p, *q;
    p=q=f;
    do
      {
        p=q->next;
        q= q-> next;
        q= q ? q->next : q      — if q is not null then
                                   q= q->next or q=q=null
      }
      while (p && q && p!=q)
    if (p==q)
        return 1;
    else
        return 0;
}
```

* C++ class for a Linked list ***

```
#include <iostream>
using namespace std;

class Node
{
  public:
    int data;
    Node *next;
};

class LinkedList
{
  private:
    Node *first;
  public:
    LinkedList() {first=nullptr;}
    LinkedList(int A[], int n);
    ~LinkedList();

    void Display();
    void Insert(int index, int x);
    int Delete(int index);
    int Length();
};

LinkedList::LinkedList(int A[], int n)
{
  Node *last, *t;
  int i=0;
  first=new Node;
  first->data=A[0];
  first->next=nullptr;
  last=first;
  for(i=1;i<n;i++)
  {
    t=new Node;
    t->data=A[i];
    t->next=nullptr;
    last->next=t;
    last=t;
  }
}

LinkedList::~LinkedList()
{
  Node *p=first;
  while(first)
  {
    first=first->next;
    delete p;
    p=first;
  }
}
```

```
Void LinkedList :: Display()
{
  Node *p = first;
  while(p)
  {
    cout << p->data << " ";
    p = p->next;
  }
  cout << endl;
}
```

```
Void LinkedList :: Insert(int index, int x)
{
  Node *t, *p = first;
  if(index < 0 || index > length())
      return;
  t = new Node;
  t->data = x;
  t->next = nullptr;
  if(index == 0)
  {
    t->next = first;
    first = t;
  }
  else
  {
    for(int i=0; i<index-1; i++)
      p = p->next;
    t->next = p->next;
    p->next = t;
  }
}
int main()
{
  int A[] = {1,2,3,4,5};
  LinkedList l(A,5);
  l.Insert(3,10);
  l.Display();

  return 0;
}
```

```
int LinkedList :: Length()
{
  Node *p = first;
  int len = 0;
  while(p)
  {
    len++;
    p = p->next;
  }
  return len;
}
```

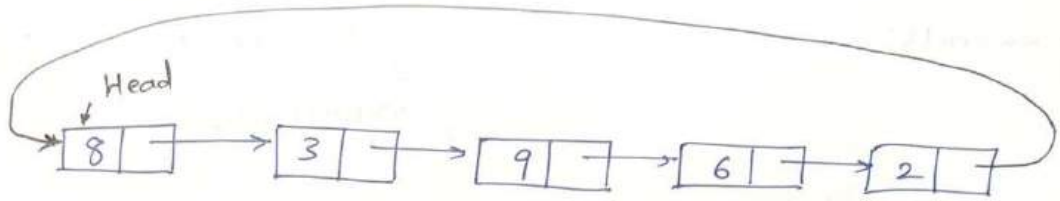
```
int LinkedList :: Delete(int index)
{
  Node *p, *q = nullptr;
  int x = -1;
  if(index < 1 || index > Length())
      return -1;
  if(index == 1)
  {
    p = first;
    first = first->next;
    x = p->data;
    delete p;
  }
  else
  {
    p = first;
    for(int i=0; i<index-1; i++)
    {
      q = p
      p = p->next;
    }
    q->next = p->next;
    x = p->data
    delete p;
  }
  return x;
}
```

* Circular Linked List:

Circular Linked List: A linked list in which the last node will point on first node, then it is circular linked list.

or collection of nodes circularly connected.

⇒ Circular list can not be null.




```
Void Display (Node *p)
{
    do
    {
        cout<< p->data;
        p = p->next;
    } while (p != Head)
}
Display (Head);
```

```
Void Display (Node *p)
{
    static int flag = 0;
    if (p != Head || flag == 0)
    {
        flag = 1;
        cout << p->data;
        Display (p->next);
    }
    flag = 0;
}
```

* To create circular LL (constructor)

```
Circular Linked List (int *A, int n)
{
    Node *t;
    Node *tail;
    head = new Node;
    head->data = A[0];
    head->next = head;
    tail = head;
    for (int i=1; i<n; i++)
    {
        t = new Node;
        t->data = A[i];
        t->next = tail->next;
        tail->next = t;
        tail = t;
    }
}
```

* Inserting in Circular Linked List:

```
Void InsertCLL (Node * head, int index, int x).
{
    Node * p = head;
    Node * temp = new Node;
    temp->data = x;
    if (index == 0)
    {
        if (head = nullptr)
        {
            head = temp;
            head->next = head;
        }
        else
        {
            temp->next = head;
            while (p->next != head)
                p = p->next;
            p->next = temp;
            head = temp;
        }
    else
    {
        for (int i=0; i < index-1; i++)
            p = p->next;
        temp->next = p->next
        p->next = temp;
    }
}
```

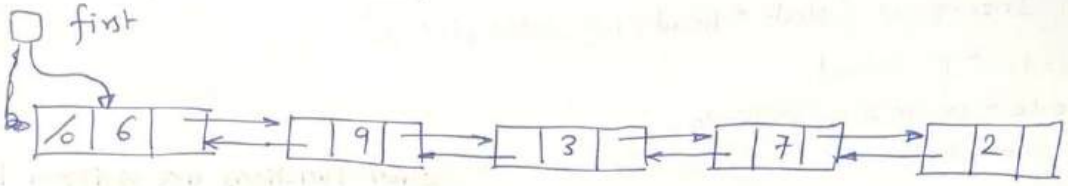
when functions are written in c++ class, then we don't have to pass head as we call on function on object.
e.g.: l.InsertCLL( )
l.Delete( )

* Deleting in Circular Linked List;

```
int DeleteCL (Node *head, int index)
{
    Node *q; int i, x; Node *p = head;
    if (index < 0 || index > Length(head))
        return -1;
    if (index == 1)
    {
        while (p->next != head)
            p = p->next;
        x = head->data;
        if (head == p)
            delete head; head = nullptr;
        else
        {
            p->next = head->next;
            delete 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; delete (q);
    }
    return x;
}
```

Length is the function written in the class which returns the length of linked list.

Doubly Linked List




```
#include <iostream>
using namespace std;
Class Node {
 public:
   Node * prev ; int data ; Node * next;
 }
class Doubly Linkedlist
{ private:
     Node * head;
 public:
   Doubly Linkedlist ();  Doubly Linkedlist (int A[], int n)
~ Doubly Linkedlist();  Void Display() ; int length();
void Insert (int index, int x); int Delete (int index) ; Void Reverse ();
}
Doubly Linkedlist :: Doubly Linkedlist()
{
   head = new Node;
   head → prev = nullptr;  head → data = 0;  head → next = nullptr;
}
Doubly Linkedlist :: Doubly Linkedlist (int *A, int n)
{ head = new Node ;  head → prev = nullptr;  head → data = A[0]
   head → next = nullptr ;  Node * tail = head;
  for (int i=1 ; i<n ; i++)
  {  Node *t = new Node;
     t → prev = tail;
     t → data = A[i];
     t → next = tail → next     or  t → next = nullptr;
     tail = t;
  }
}
  Void DoublyLinkedList:: Display()
{  Node *p = head;
   while (P != nullptr)
   {  cout << p → data ;
      p = p → next;
      if (P != nullptr)
          cout << " ↔ "
   }
   cout << endl;
}
```

Node

prev	data	next

```
class Node {
{ public:
    int data;
    Node* prev;
    Node * next;
}
```

* If the space is not a problem the Doubly circular linked list is best.

```
int DoublyLL::length()
{ int length=0;
  Node *p = head;
  while(p!=nullptr)
   { length++;
     p=p->next;
   }
  return length;
}
```

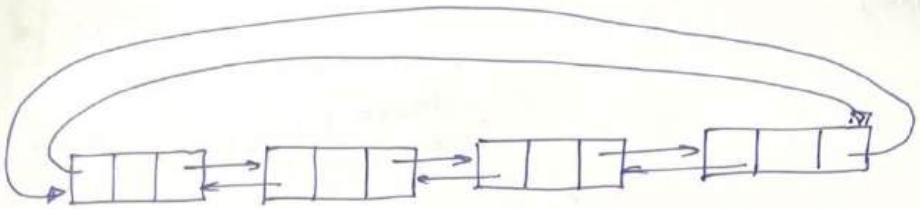
```
Void D.L.L.:: Insert(int index, int x)
if (index <0 || index >length())
     return;
Node * p=head;
Node *t= new Node;
t->data = x;
if (index == 0)
  { t->prev=nullptr;
    t->next = head;
    head->prev=t;
    head = t;
  }
else
   for(int i=0; i<index-1; i++)
        p=p->next;
   t->prev=p;
   t->next= p->next;
   if (p->next)
      p->next->prev=t;
   p->next=t;
  }
}
int main()
{ int A[]={1,3,5,7,9};
DoublyLinkedList dll(A,5);
  dll.Insert(0,11);
  dll.Delete(1);
  dll.Display();
  dll.Reverse;
  return 0;
}
```

```
DoublyLinkedList :: ~DoublyLinkedList()
{  Node *p=head;
   while(head)
     {
       head = head->next;
       delete p;
       p=head;
     }
}
```

```
int D.L.L.:: Delete(int index)
{   int x=-1;
    Node * p= head;
 if(index<0 || index >length())
      return x;
 if(index == 1)
 {  head = head->next;
    if(head)
       head->prev = nullptr;
    x= p->data;
    delete p;
 }
 else
 {  for(int i=0; i<index-1; i++)
       p = p->next;
    p->prev->next = p->next;
    if (p->next)
       p->next->prev = p->prev;
    x= p->data;
    delete p;
 }
 return x;
}
```

```
Void D.L.L.:: Reverse()
{ Node *p=head;
  Node * temp;
  while(p!=nullptr)
  {
    temp=p->next; p->next=p->prev;
    p->prev=temp; p=p->prev;
    if(p->next==nullptr)
    {
      p->next=p->prev;
      p->prev=nullptr; head=p;
      break;
    }
  }
}
```

* Circular Doubly Linked List :

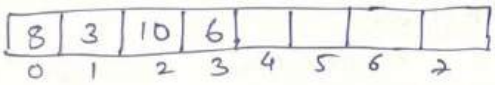


⇒ If memory space is not a constrained then Circular-Doubly linked list is the best option.

⇒ We can perform all the operations (Insert, Delete, Display

* Comparison between Array and Linked List

Array



① Array can be created inside stack as well as heap.

② Once you have created an array then size is fixed.

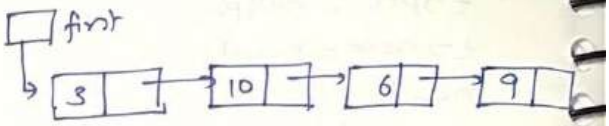
③ As this is fixed in length chances of getting perfectly utilized is less. It should be used when we are sure about the number of elements needs to store.

④ Will occupy space only equal to the size of data. So takes less space.

⑤ Elements can be accessed randomly by just a call e.g A[2], A[5].

⑥ As it can be accessed randomly and it is inside stack so accessing is faster in Array.

Linked List.



① Linked list is always created in heap.

② Linked list is a variable size. You can grow as much as want.

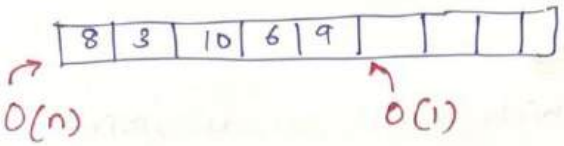
③ Linked list can be used when we don't know the length. We can create nodes if required and also can reduce when not in use.

④ Need space for data and also for the pointer pointing to next node. Hence takes more space.

⑤ Elements can be accessed sequentially. If need to access something we should start from first and go till we reach.

⑥ As it can be accessed sequentially and present in heap (Indirect contact) so accessing is comparatively slower.

⑦ Insert:



→ If we insert from left hand side then we have to shift all previous elements to right. hense time required is order of n O(n).

→ If we insert from RHS. then we can directly insert at last position and hense time is constant O(1).

→ In array, Data shifting is reqd It is comparatively complex and require more space.

→ So if we are inserting data from RHS then array will be faster.

⑧ Delete

Order of time for delete is same as Insert.

⑨ Search.

In Array, we can perform Linear Search as well as binary search.

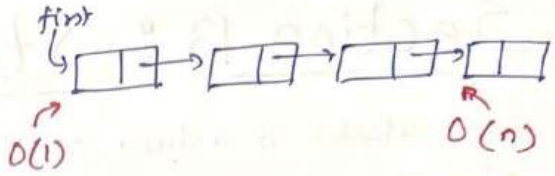
Linear → O(n) Time

Binary → O(log n)

→ so search is faster in Array

⑩ Most of the sorting techniques are designed for Array.

⑦ Insert:



→ If we insert from LHS, then simply create one node, link it and move first. So time required is constant O(1).

→ If we insert from RHS then we should move one pointer p from first to last node and then create a node and link it. so time required is order of n O(n).

→ In Linked List pointer shifting is required. it is simple and less space consuming.

→ If we are inserting data from LHS then Linked List will be faster.

⑧ Delete:

Order of time for delete is same as Insert.

⑨ search:

We can perform only Linear Search. We cannot perform binary search. If we perform binary search then order of time will be N.logN which is inefficient.

⑩ Some of the sort techniques suitable for Linked list.

① Insertion sort.

② Merge sort.

Section 13 : Stack

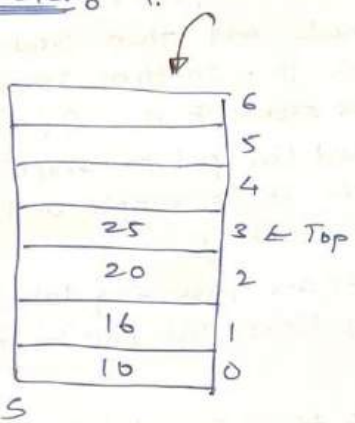
→ ADT stacks is a data structure which works on discipline of LIFO: Last in first out.

* Abstract Data Type (ADT) of stack.

It is the collection of elements which follow principle LIFO.

Data :

1. Space for storing elements
2. Top pointers.



* Operations :

1. Push(x) → To insert the value
2. Pop() → To remove topmost value.
3. Peek(Index) → To see what is the value at given index from top.
4. StackTop() → value at the top.
5. isEmply().
6. isfull().

⇒ We can implement stack using following know data structure.

1. Array 2. Linked List

① Stack Using Array :

```
class Stack
{
  private:
    int size;
    int top;
    int *s;
  public:
    Stack(int size);
    ~Stack();
    void push(int x);
    int pop();
    int peek(int index);
    int isFull();
    int isEmpty();
    void display();
    int stackTop();
};
```

```
Stack::Stack(int size)
{
  this->size = size;
  top = -1;
  s = new int(size);
}

Stack::~Stack()
{
  delete s;
  s = nullptr;
}

void Stack::push(int x)
{
  if(isFull())
    cout << "Stack Overflow!" << endl
  else
  {
    top++
    s[top] = x;
  }
}
```

```
int stack::pop()
{ int x=1;
  if (isEmpty())
    cout << stack Underflow!;
  else
  {
    x = s[top]
    top--;
  }
  return x;
}
```

```
int Stack::peek(int index)
{  int x=-1;
 if(top-index+1 < 0 || top-index+1==size)
   cout<<" Invalid Position" <<endl;
 else
    x = s[top-index+1];
  return x;
}
```

"top-index+1" is just a conversion formula for index & position. and it is form by observation.

```
int stack::isfull()
{ if (top==size-1)
    return 1;
  return 0;
}
```

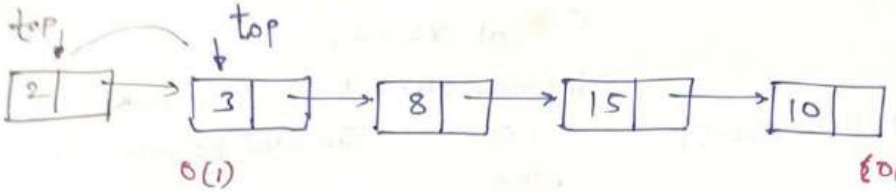
```
int stack::isEmpty()
{ if (top == -1)
    return -1;
  return 0;
}
```

```
void stack::display()
{ for(int i=top; i>=0; i--)
    cout<< s[i] <<"|" <<flush;
  cout<<endl;
}
```

```
int Stack::stackTop()
{  if (isEmpty())
     return -1;
   return s[top];
}
```

```
int main()
{ int A[] = {1, 3, 5, 7, 9}
  stack stk(sizeof(A)/sizeof(A[0]));
// populate stack with array elements;
  for (int i=0; i < sizeof(A)/sizeof(A[0]); i++)
  {   stk.push(A[i]);
   stk.push(11);
   stk.display();
   stk.peek(3);
   stk.stackTop();
   stk.pop;

    return 0;
  }
```

* Stack Using Linked List:




```
#include <iostream>
using namespace std;
class Node
{
 public:
    int data;
    Node * next;
};
class Stack
{ private:
      Node * top;
 public:
   Stack();
 ~Stack();
 void push(int x);
 int pop();
 int peek(int index);
 int isEmpty();
 int isFull();
 int stackTop();
};
```

```
void stack::push(int x)
{
  Node *t= new Node;
  if (t==nullptr)
     cout<<"stack overflow";
  else
    {
       t->data = x;
       t->next = top;
       top = t;
    }
}
```

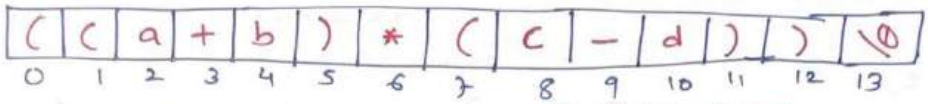
```
Stack:: stack()
  {   top=nullptr;
  }
Stack :: ~stack()
  {
    Node * p = top;
    while(top)
     {
       top = top->next;
        delete p;
        p=top;
     }
}
```

```
int stack::pop()
{
   Node *p;
   int x = -1;
  if (top==nullptr)
      cout << "stack Underflow"<<endl.
  else
    {
      p=top;
      x=p->data;
      top= top->next;
       delete p;
    }
  return x;
}
```

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

int Stack:: isEmpty()
{
  return top ? 0:1;
}

int Stack:: stackTop()
{
  if(top)
    return top->data;
  return -1;
}

int Stack:: peek(int index)
{
  if(isEmpty())
    return -1;
  else
  {
    Node *p=top;
    for(int i=0; p!=nullptr && index-1; i++)
    {
      p=p->next;
    }
    if(p!=nullptr)
      return p->data;
    else
      return -1;
  }
}

int main()
{
  int A[]={1,3,5,7,9};
  Stack stk;

  // populate stack
  for(int i=0; i<sizeof(A)/sizeof(A[0]); i++)
    stk.push(A[i]);

  stk.peek(3);
  stk.pop();
}
```

* Use of stack ① Paranthesis Matching:

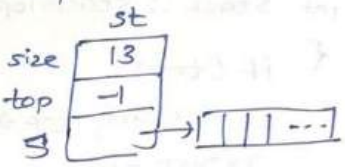
exp : ((a+b) * (c-d))



① Using Array

```
int isBalance (char * exp)
{
    struct Stack st;
    st.size = strlen(exp);
    st.top = -1;
    st.s = new char [st.size];
    for(i=0; exp[i] != '\0'; i++)
    {
        if (exp[i] == '(')
            push (&st, exp[i]);
        else if (exp[i] == ')')
        {
            if (isEmpty (st))
                return false;
            pop (&st);
        }
    }
    return isEmpty (st) ? true : false;
}
```

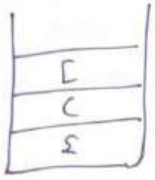
```
struct Stack
{
    int size;
    int top;
    char * s;
};
```



② Using Linked List.

```
int isBalanced (Char * exp)
{
    stack stk;
    for (int i=0; i<strlen(exp); i++)
    {
        if (exp[i] == '(')
            stk.push (exp(i));
        else if (exp[i] == ')')
        {
            if (stk.isEmpty())
                return false;
            else
                stk.pop
        }
    }
    return stk.isEmpty() ? true : false;
}
```

{ ([a+b] * [c-d])/e }




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

② Infix to Postfix Conversion

1. Infix: Operand Operator Operand

   e.g: a+b → a add to b.

2. Prefix: Operator Operand Operand

   e.g: +ab → add a and b

3. Postfix: Operand operand operator

   e.g: ab+ → a and b what to do add.

→ for complex expression $8+3*(9-6)/2^2+6/2$

→ To perform and calculate the result of this expression we have to do action according to "Precedence" of operator given by BODMAS Rule.

⇒ In the Infix, it is impossible to perform all action in one scan. We need to scan for each operation separately and that is time consuming and inefficient.

⇒ Using Postfix we can perform all operator action in a single scan.

⇒ First we need to convert regular infix expression to postfix form.

⇒ We generally used Postfix form not a Prefix form.

Infix : $8+3*(9-6)/2^2+6/2$

Postfix : 8 3 9 6 - 2 ^ / * + 6 2 / +

Symbol	Precedence
+, −	1
*, /	2
( )	3 ← Highest Precedence.

⇒ If the two symbols are same in a expression, then we give Precedence from left to right.

* Conversion from Infix to prefix and postfix

⇒ Rule: Whenever we write ~~and~~ an expression, it should be fully parenthesized.
No operator should be left open.

⇒ Actually, compiler needs fully parenthesized expression.

① a+b*c

(a+(b*c)) ← fully parenthesized.

[] represent already converted.

Prefix

(a+[*bc])

+a*bc

Postfix

(a+[bc*])

abc*+

② a+b+c*d

a+b+c*d ← Now, imagine parenthesis

Prefix

a+b+c*d

a+b+[*cd]

[+ab]+[*cd]

++ab*cd

Postfix

a+b+c*d

a+b+[cd*]

[ab+]+[cd*]

ab+cd*+

③ (a+b)*(c-d)

Prefix

(a+b)*(c-d)

[+ab]*(c-d)

[+ab]*[-cd]

*+ab-cd

Postfix

(a+b)*(c-d)

[ab+]*(c-d)

[ab+]*[cd-]

ab+cd-*

* Infix, to Postfix Conversion

Associativity

⇒ If we don't provide paranthesis then compilor logically creats the paranthesis depending on Prescedence and Associativity.

Sym	Pre	Asso
+−	1	L-R
*, /	2	L-R
^ (power operator)	3	R-L
− (negating binary oprt.)	4	R-L
( )	5	L-R
=		R-L

L-R ⇒ Left to right.

① $a+b+c-d$

Left to Right

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

$ab+c+d-$

② $a = b = c = 5$ (3 2 1)

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

$(a=(b=[c5=]))$

Right to Left.

$(a=[bc5==])$

$abc5===$

③ $a \wedge b \wedge c$

$(a \wedge (b \wedge c)) = (a)^{(bc)}$

$a \wedge b \wedge c$

Right to Left.

$a \wedge [bc\wedge]$

$abc\wedge\wedge$

④ $-a$ — negation unary operator

Pre: $-a$ Post: $a-$

Right to left.

⑤ $--a$

$(-(-a))$

Pre: $--a$ Post: $a--$

In unary operation, there is only one operand.

Dereferencing.

Right to Left.

⑥ *p

Prep: *p Post: p*.

**p

(*(*p))

***p** p**

⑦ n!

Pre: !n Post: n!

⑧ log x

Pre: logx Post: xlog

⇒ All Urinary operator such as Negation (-), dereferencing(*), factorial (!), logarithmic (log) have same prescedence and that is Right to left.

** e.g:

$$-a + b * \log n!$$

$$3 \quad 5 \quad 4 \quad 2 \quad 1$$

$$-a + b * \log [n!]$$

$$-a + b * [n! \log]$$

$$[a-] + b * [n! \log]$$

$$[a-] + [b\, n! \log *]$$

$$a - b\, n! \log * +$$

Urinary → multiplication → addition (R-L)

* Infix to Postfix using

① stack Method ①.

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
		abc*+de/-

Sym	Pre.	Asso
+ -	1	L-R
* /	2	L-R
a,b,c	3	L-R

② Stack Method ②

follow same rules, as method ①; only here variables (operands) also passes through stack and they got highest prescedence i.e. 3.

Sym	Pre.	Asso.
+ -	1	L-R
* /	2	L-R
a,b	3	L-R.

① Push in stack if it is previously empty or having element of ~~high~~ lower prescedence.

② If it contains the element of higher prescedence then before pushing it in the stack, pop out ~~the~~ to all the higher prescedence elements from the stack and add them in the Postfix. and once it is done then push in the stack.

* code:

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

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

Using Array.

```
char * InToPost (char * infix)
{  int i=0 ; j=0
 char * postfix ;
int len =strlen(infix)
postfix = new char [len+1]
while (infix [i] != '\0')
{ if (isOperand (infix[i]))
      postfix [j++]=infix[i++]
          i.e. P.f[j]=if[i]
                j++ , i++.
  else
   {if (pre (infix [i]) > pre (StackTop(st))
      {   push (&st, infix[i++])
          i++ ;
       }
     else
        { postfix[j]= pop(&st);
          j++ i;
        }
   }
  while ( ! isEmpty (st))
   {postfix[j]= pop (&st)
          j++ ;
   }
  postfix[j]='\0';
  return postfix;
}
```

// creating postfix of string length + 1 to store '\0' at the end;

// if it is operand then directly adding it to postfix

// If prescedence of element (operator) to be push in stack is higher than the element at the top in the stack. then push the element in stack.

→ // If it is not then first pop-out the higher prrscedence element

// Once all the elements in the string are passed then, pop out all the elements which are remaining in the stack and add '\0' at the end of postfix

```
int main()
 {   char * infix = " a+b*c-d/e";
     push ('#')
     char *postfix = InToPost(infix);
      cout << postfix ;
      return 0;
 }
```

~~Using LinkedList~~

* Evaluation of Postfix Expression :

→ In Evaluation, we parsee throughf the postfix expression

- ① If it is operand then we push it in the stack
- ② If it is a operator then we pop out two values from the stack and perform the operation on this two values.

③ Important, first value pop out will be on RHS and seconed value is ~~on~~ 1st popped out is on LHS and operator will be in middle.

⇒ e.g: Infix: 3 * 5 + 6/2 - 4

Postfix: 3 5 * 6 2 / + 4 -

Symbol	Stack	Operation
3	3	
5	⇒ 5, 3	
*	⇒ 15	3 * 5 = 15
6	⇒ 6, 15	
2	⇒ 2, 6, 15	
/	⇒ 3, 15	6/2 = 3
+	⇒ 18	15+3=18
4	⇒ 4, 18	
-	⇒ 14	18-4 = 14
	Answer = 14	

⇒ Precedents and Associativity are meant for parenthesization not for execution.

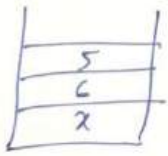
Imp:
- They decide how the parenthesis will be given, not which operation will perform first.

e.g: x = 6 + 5 + 3 * 4

(2) (1)

x = 65 + 34 * + =

((6+5) + (3*4))



```
int operation (char op, int x, int y)
{ int (op == '+')
        return x+y;
  elseif (op == '-')
        return x-y;
   elseif( op == '*')
         return x* y;
    elseif (op == '/')
          return x/y;
  }
int Evaluate (char *postfix)
{  Stack stk;
   int x, y, result;
  for(int i=0; postfix[i] != '\0'; i++)
{
   if (isOperand (postfix[i])
            stk.push(postfix[i] - '0');
    else
    {
       y = stk.pop();
       x = stk.pop();
       result = operation( postfix[i], x, y);
       stk.push(result)
    }
  }
   result = stk.pop()
  return result;
}

int main()
  {
     char postfix[] = " 35*62/+4- ";
     cout << Evaluate (postfix) << endl;
     return 0;
  }
```

need to substract the ASCII code of '0' from all as It is a character array postfix and we need to perform the operation on integers

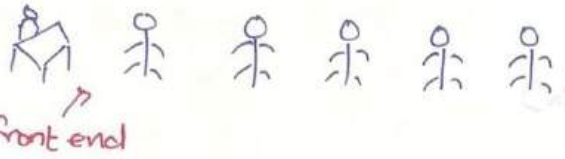
Section 14 : Queus

⇒ Queue is a logical data structure.

⇒ It works on dispiene of first In first Out (FIFO)

⇒ Simple example of Queue is people standing in a queue. for their turn.

⇒ Queus has two ends. Insertion is done at rear end and deletion is done at front end.



⇒ Queue ADT

Data:
1. Space for storing elements.
2. Front ~~poter~~ pointer for deletion
3. Rear pointer : For Insertion.

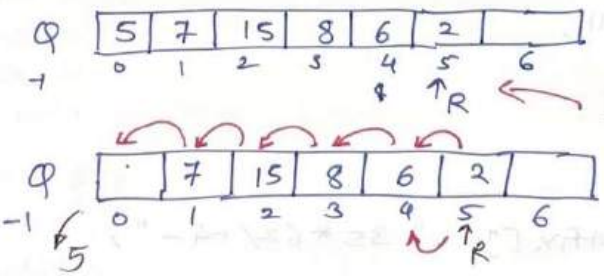
Operations:
1. enqueue(x)
2. dequeue()
3. isEmpty()
4. isfull()
5. First()
6. Last()

⇒ Queue can be implemented using
1) Array
2) Linked list

* ① Implementation of Queue with Array.

① Method ①: Queue using Single Pointer.

size = 7



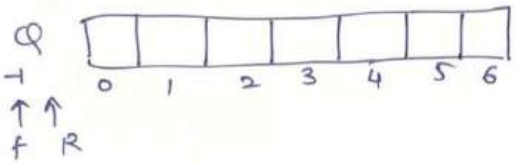
Time: Insertion : O(1) As we are inserting at last and before inserting shifting last pointer to next position.

** Delete: O(n). As we delete from first position so that position is empty. We can't keep first position empty in array. So we need to shift all elements to Right one by one.

Method 2: Queue using two Pointers

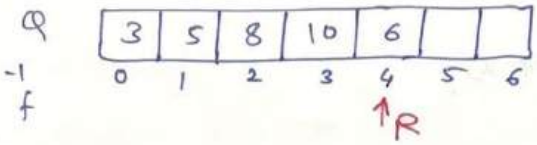
⇒ In method 1, drawback was. It take time order of 'n' to delete a element from a Queue.

⇒ So to overcome this drawback, Method 2. uses two pointer first and Rear.



Insertion : O(1)
Deletion : O(1)

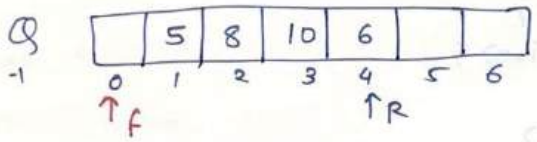
1. Insertion:



→ While insertion Rear pointers moves to next position and then element is inserted at that position.

→ Rear will always be pointing on the Last elements

2. Deletion:



⇒ for deletion, first pointer moves to next location and delete the element at that location.

⇒ first pointer will always point at the position just before the first element (It act as a counter kept in a Queue).

* Conditions:

Initially: front = Rear = -1

① isEmpty () ⇒ front = Rear

② isfull () ⇒ if (Rear = size - 1)

6 = size(7) - 1

* Queues Using Two pointer:

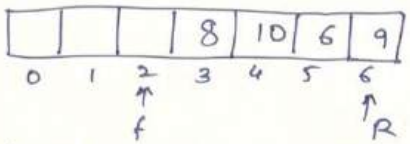
```
#include <iostream>
#Using namespace std;

class Queue
{
  private:
      int size;
      int front;
      int rear;
      int *Q;
  public:
    Queue (int size);
   ~Queue ();
    void enqueue (int x);
    int dequeue ();
    void display ();
};

Queue :: Queue (int size)
{
   this -> size = size;
   front = rear = -1;
   Q = new int (size);
}

Queue :: ~Queue()
{ delete Q;
  Q=nullptr;
}

void Queue :: enqueue (int x)
{ if (rear == size-1)
      cout << " Queue is full ";

    else
      {
        rear ++;
        Q[rear] = x;

  }
```

```
int Queue::dequeue()
{
    int x = -1;
    if (front == rear)
        cout << "Queue is empty";
    else
    {
        front++;
        x = Q[front];
    }
    return x;
}

void Queue::display()
{
    for (int i = front+1; i <= rear; i++)
        cout << Q[i] << endl;
}

int main()
{
    Queue q(7);
    q.enqueue(3);
    q.enqueue(5);
    q.enqueue(8);
    q.enqueue(10);
    q.enqueue(13);
    q.dequeue();
    q.display();
}
```

* Drawbacks of using Array for a Queue.

Drawbacks

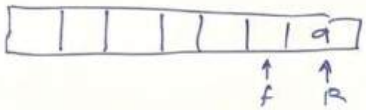
①



Now, in this if we try to fill a new element, it will show array is full as we insert from rear end.

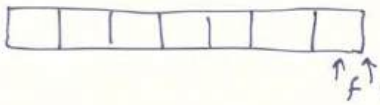
But here, we have allready deleted some elements so some spaces are vaccant but we can not reuse them.

②



Every location can be use only once we can not use location again.

③



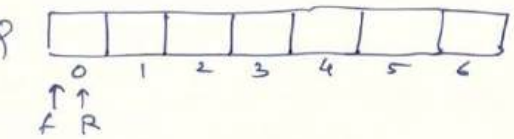
The situation where queue is empty also and full also. as front and rear are at same position.

Solutions

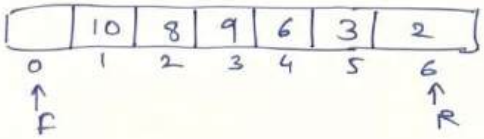
1. Resetting Pointers: If at any places rear and front are becoming same at any time then bring first and rear at beginning. i.e. re-initialize them to minus one. so that they can again start from beginning and we can reuse those places.

2. Circular Queue: In circular Queue, array is not circular but ~~Re~~ Front and Rear moves in a circular way.

→ Initially both Rear and Fromt are at position zero (0).

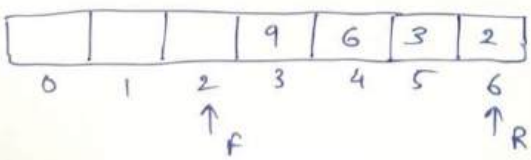


→ while filling elements rear will move forward.



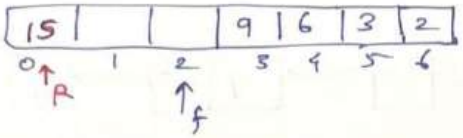
** ⇒ Don't fill element in first space. The space (position) where * front is pointing will be always empty.

⇒ Let's remove some elements. then front will move forward.

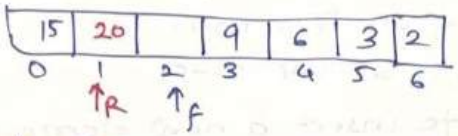


⇒ Now, if we insert any element rear will start from beggining and place elements over there.

Add → 15



Add → 20

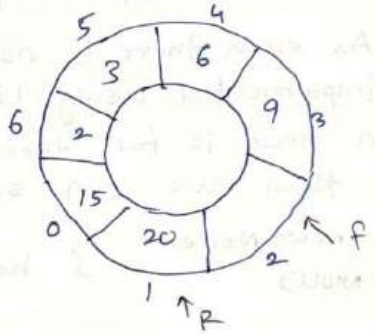


⇒ We can not insert at position where First is pointing.
If we insert there then First and rear will be pointing at same position and that is condition for empty Queue.

⇒ We can use MOD (calculate Reminder) operation to get the circular position of elements. using as follows.

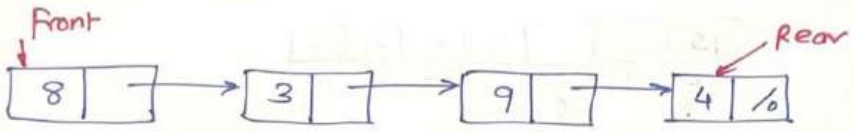
Rear = (Rear+1) % size

Pos			
0	=	(0+1) % 7	= 1
1	=	(1+1) % 7	= 2
2	=	(2+1) % 7	= 3
3	=	(3+1) % 7	= 4
4	=	(4+1) % 7	= 5
5	=	(5+1) % 7	= 6
6	=	(6+1) % 7	= 0




```
void CircularQueue::enqueue(int x)
{
   if ((rear+1) % size == front)
       cout << "Queue is full
                overflow" <<endl;

   else
    { rear = (rear+1) % size;
      Q[rear] = x;
    }
}
int CircularQueue::dequeue()
{   int x = -1;
    if (front == rear)
        cout << "Queue Underflow" << endl;
    else
      {
        front = (front+1) % size;
        x = Q[front];
      }
    return x;
}
```

② Implementation of Queue Using Linked List.



→ Front is a pointer pointing on First node.
→ Rear is a pointer pointing on last node.
→ Rear pointer helps to insert a new element at a constant time O(1).
→ If there is no node. i.e. Queue is empty then Fron = Rear = NULL.

Empty if (front = NULL)

→ When the first node is created then, both front and rear will be pointing on first node.

→ isfull () → As such there is no condition on maximum size of Queue when implemented using Linked list. So there is no fix size. But when heap is full then we will be unable to create any node then we can say Queue is full.

```
Node *t = new Node
  if (t = NULL)   } heap is full.
```

* Code:

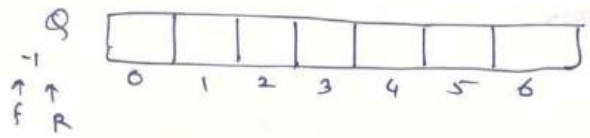
```
#include <iostream>
using namespace std;

class Node
{
  public:
    int data;
    Node * next;
};

class Queue
{
  private:
    Node* front;
    Node * Rear;
  public:
    Queue();
   ~Queue();
    void enqueue(int x);
    int dequeue();
    bool isEmpty();
    void display();
};
```

```
Queue :: Queue()
{
    front = nullptr;
    rear = nullptr;
}

Queue :: ~Queue()
{
   Node *p = front;
   while(front)
   {
     front = front->next;
     delete p;
     p = front;
   }
}
```

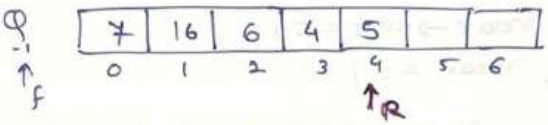
```
void Queue :: enqueue (int x)
{
    Node *t = new Node;
    If (t == nullptr)
        cout << "Queue Overflow" << endl;
    else
    {
        t -> data = x
        t -> next = nullptr;
        if (front == nullptr)
        {
            front = t;
            rear = t;
        }
        else
        {
            rear -> next = t;
            rear = t;
        }
    }
}
int Queue :: dequeue()
{
    int x = -1;
    Node *p;
    if (front == nullptr)
        cout << "Queue Underflow" << endl.
    else
    {
        p = front;
        front = front -> next;
        x = p -> data;
        delete p;
    }
    return x;
}

int main()
{
    int A[] = {1, 3, 5, 7, 9};
    Queue q;
    for (int i=0; i < 5; i++)
        q.enqueue A[i];
    q.display();
    q.dequeue();
    return 0;
}

void Queue :: display()
{
    Node *p = front;
    while (p)
    {
        cout << p -> data << flush;
        p = p -> next;
        If (p != nullptr)
            cout << " <- " << flush;
    }
    cout << endl;
}
```

* Double Ended Queue DEQueue

→ DEQueue doesn't strictly follow FIFO principle.
→ It can be implemented by both array and linked list.
→ In DEQueue, front pointer can be use for both insert as well as Delete. Some way Rear pointer can be use for insert as well as Delete.

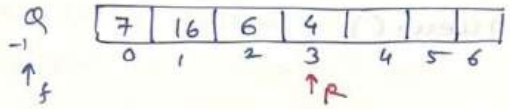


1. Insert using Rear:



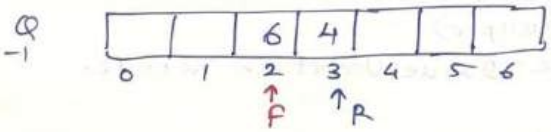
Rear incremented and inserted.

2. Delete using Rear:



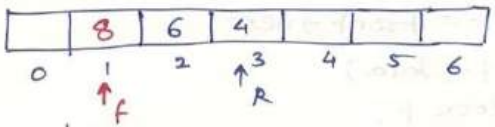
Deleted and Rear decremented.

3. Delete using front:



Deleted and front incremented.

4. Insert using front:



Front decremented and Inserted.

⇒ JAVA supports DEQueue data structure.

* Types of Restricted DEQueue.

① i/p Restricted DEQueue

	Insert	Delete
Front	✗	✓
Rear	✓	✓

② o/p Restricted DEQueue

	Insert	Delete
Front	✓	✓
Rear	✓	✗

```
void DEQueue :: enqueueFront (int x)
{   if (front == -1)
        cout << " DEQueue Overflow" <<endl;
    else
      {   Q [front] = x;
          front --;
      }
}
void DEQueue :: enqueueRear (int x)
{
    if (rear == size-1)
        cout << " DEQue Overflow" <<endl;
    else
      {  rear ++ ;
         Q [rear] = x;
      }
}
void DEQueue :: dequeuefront ()
{   int x = -1 ;
    if ( front == rear)
        cout << " DEQueue Underflow" <<endl;
    else
      {   x = Q [front] ;
          front++;
      }
}
void DEQueue :: dequeueRear()
{
    int x = -1 ;
    if (rear == -1)
        cout << " DEQueue Underflow" <<endl;
    else
      {   x = Q [rear];
          rear -- ;
      }
    return x ;
}
```

* Priority Queues.

→ There are two methods of implementing Priority Queue.

Method ① Limited set of Priorities: ⇒ In this priorities for the elements are already set.

→ This method is useful in mostly in operating system.
→ we can set priorities for threads in Multi-threading.

e.g: Priorities = 3

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

Priority Queues →

→ Here 1 is highest priority

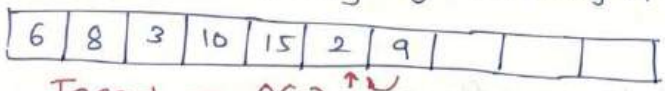
⇒ Operations such as Delete, should be done always from highest priority queue first, If that is empty then go to next priority Queue and so on.

Method ② Element Priorities: Here number itself represents its priority.

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

⇒ Smaller Number → Higher Priority.

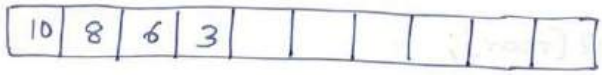
Case 1: Insert in some order
Delete Max. Priority by searching it.



Time: Insert — O(1)

Delete → search — O(n) = O(2n) = O(n)
shift — O(n)

Case 2: Insert in Increasing order of Priority
Delete last element of Array.

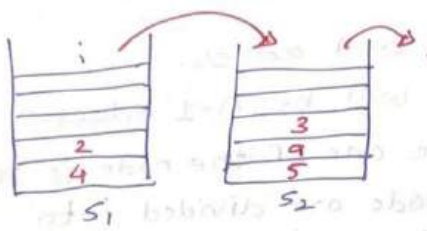


Time: Insert = → shift - O(n) = O(n)

Delete = O(1)

* Queue Using 2 Stacks:

elements: 6, 3, 9, 5, 4, 2, 8, 12, 10, ----



→ In enqueue, push elements in stack S_1.

→ In dequeue, pop elements from stack S_2. If S_2 is empty then transfer elements from S_1 to S_2 and then pop from S_2.

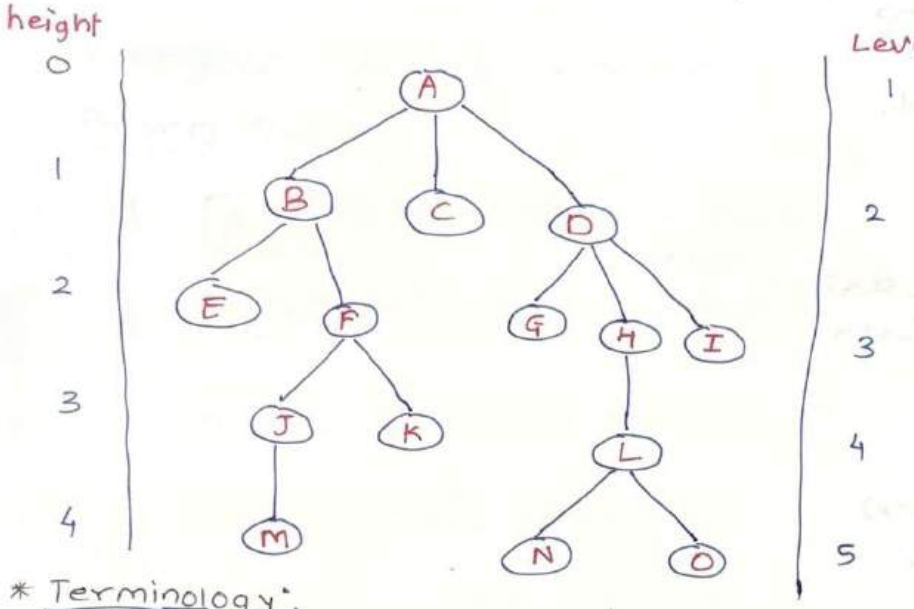
```
#include <iostream>
#include <stack>
using namespace std;
Class Queue
{
private:
  stack <int> e_stk;
  stack <int> d_stk;
public:
  Queue() {};
~Queue() {};
  void enqueue(int x)
  int dequeue();
};
void Queue:: enaueu(int x)
{  e-stk.push(x);
}
int Queue :: dequeue()
{   int x=-1;
   if (d_stk.empty())
     {  if (e_stk.empty())
          {  cout << "Queue Underflow"
             return x;
          }
        else
          {  while(!e_stk.empty()
               {  d_stk.push(e_stk.top());
                  e_stk.pop();
               }
          }
     x = d_stk.top();
     d_stk.pop();
     return x;
   }
```

Section 15 : Trees

Trees: It is a collection of Nodes and edges.

→ If there are 'n' nodes then there will be 'n-1' edges.

→ Tree is a collection of node where one of the node is taken as a root node and the rest of node are divided into disjoint subset and each subset is a tree or subtree.



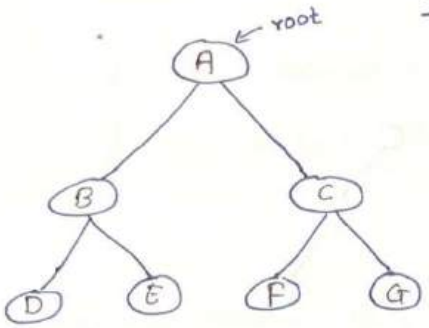
* Terminology:→

1. Root : Top most node
2. Parent :
3. Child :
4. siblings :
5. Descendents
6. Ancestors : All the nodes from root to that node along the path, are the Ancestors to that node.
7. Degree of Node: Degree of Node is the number of direct children.

   Degree of B = 2 , Degree of D = 3
8. Internal/External : Non-terminal/Terminal Nodes:

   ~~Non~~ Internal → Degree ≠ 0 External → Degree = 0.
9. Levels :
10. Heights:
11. Forest: Collection of trees.

* Binary Tree



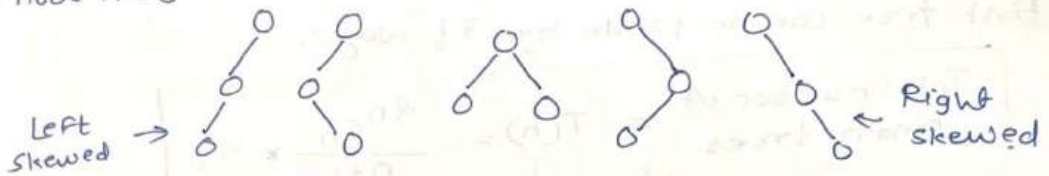
If max. degree is two, then it is called as Binary tree. (or less than two)

→ It can have degree less than two but can't have more than two.

→ children = {0, 1, 2}

⇒ (1) Number of binary trees: (for Unlabelled Nodes)

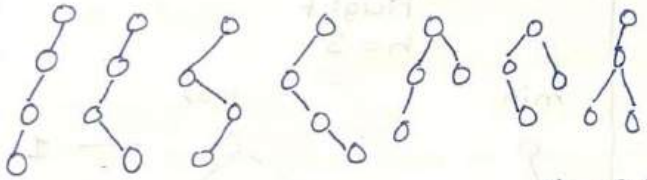
If node n = 3



Total number of binary trees = 5 $T(3) = 5$

Number of trees having maximum height = 4 $= 2^2$

If node n = 4



$T(5) = 14$

replica

No. of trees with maximum height = 8

If node ⇒ n = n

Total number of binary trees $T(n) = \frac{2nC_n}{n+1}$ ← Catalan Number

$$T(5) = \frac{2\times5C_5}{5+1} = \frac{10C_5}{6} = \frac{10\times9\times8\times7\times6}{5\times4\times3\times2\times1\times6} \quad T(5) = 42$$

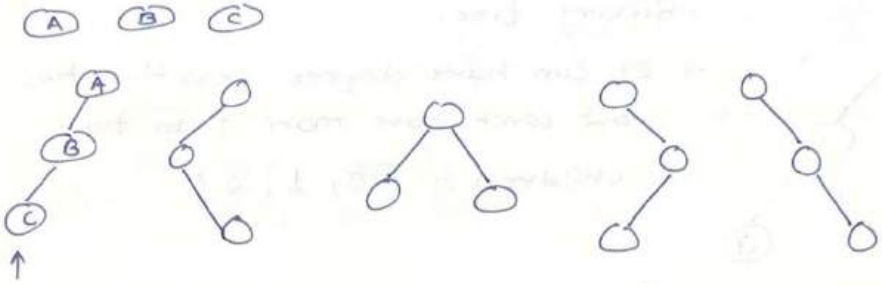
Number of trees having maximum height $= 2^{n-1}$

⇒ Other formula for Catalan Number

$$T(n) = \sum_{i=1}^{n} T(i-1) * T(n-i) \qquad T(0) = 1.$$

Q Number of binary trees (for labelled Node)

$n = 3$



But each node can be filled by different arrangements of labelled node.

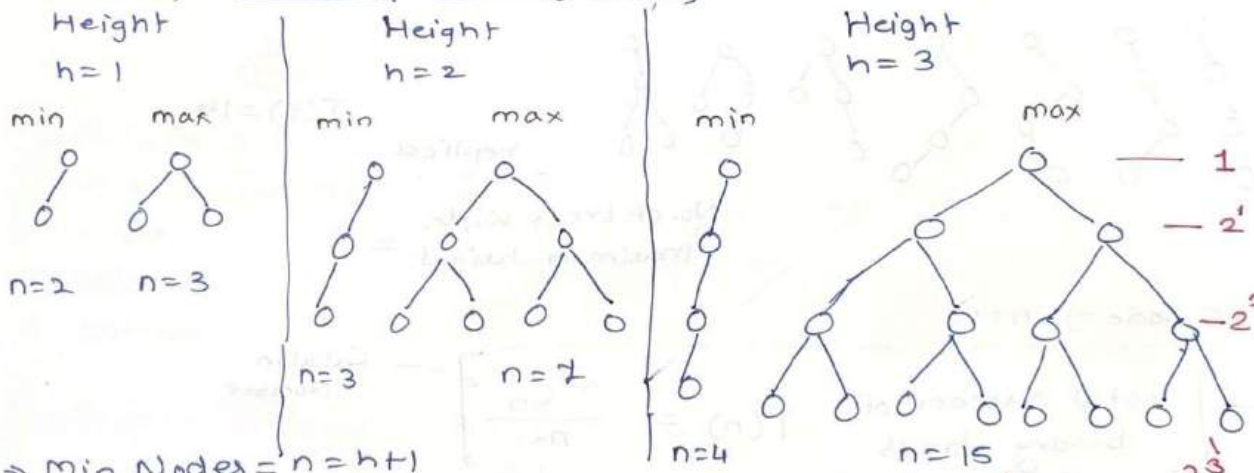
Here first tree can be made by $3!$ ways.

Hence,

$$\text{Total number of binary trees} = T(n) = \frac{^{2n}C_n}{n+1} * n!$$

(shapes) (filling)

* Height vs Nodes in Binary Tree:



$\Rightarrow$ Min Nodes $= n = h+1$

$= 1 + 2 + 2^2 + 2^3 = 15$ GP.

G.P. series $= a + ar + ar^2 + ar^3 + \ldots ar^k = \frac{a(r^{k+1} - 1)}{r-1}$

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

Min. Nodes	$n = h+1$
Max Nodes	$n = 2^{h+1} - 1$

⇒ Now if we know number of nodes then we can calculate height.

Max. height	$h_{max} = n-1$	$O(n)$
Min. height	$h_{min} = \log_2(n+1) - 1$	$O(\log n)$

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

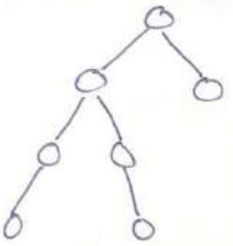
$n+1 = 2^{h+1}$

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

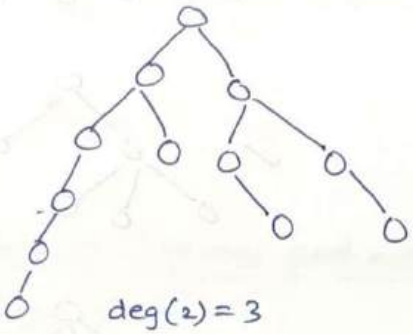
$h+1 \cdot \log 2 = \log(n+1)$

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

* Internal Nodes vs External Nodes in Binary Tree



deg (2) = 2
deg (1) = 2
deg (0) = 3



deg (2) = 3
deg (1) = 5
deg (0) = 4

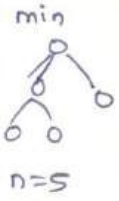
deg (2) = 1
deg (1) = 4
deg (0) = 2

$$[\deg(0) = \deg(2) + 1]$$

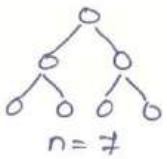
* strict Binary Tree : Proper/Complete Binary tree.

⇒ strict Binary Tree, node can have either zero or two children. It can not have one children

h = 2

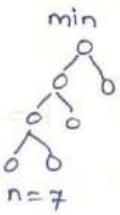


n=5

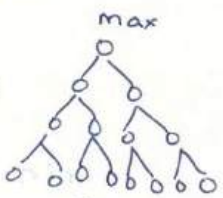


n = 7

h = 3

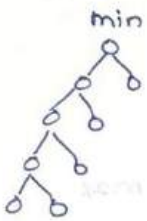


n=7



n=15

h=4



n=9

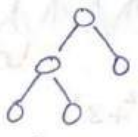
n = 31

if height 'h' is given

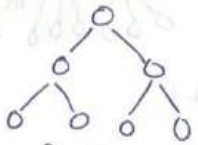
Min Nodes	$n_{min} = 2h+1$
Max Nodes	$n_{max} = 2^{h+1} - 1$

if 'n' Nodes are given

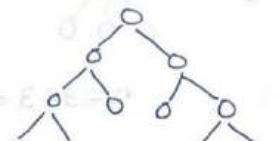
Min Height	$h_{mix} = \log_2(n+1) - 1$
Max Height	$h_{max} = \frac{n-1}{2}$



i = 2
e = 3



i = 3
e = 4



i = 5
e = 6

i = internal nodes

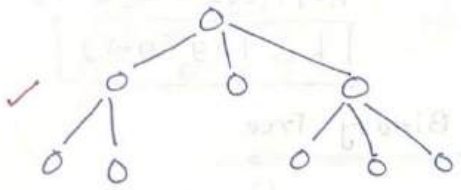
e = external nodes

$$e = i + 1$$

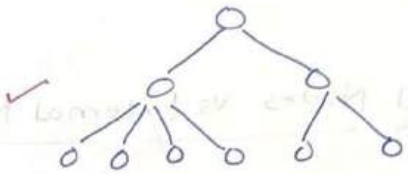
* n-array Trees

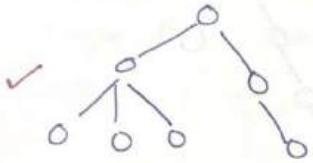
→ n-array tree, every node have the maximum capacity to have 'n' children, if they are less then its fine.. but can not have children more than 'n'.

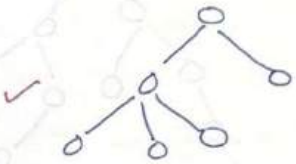
3-ary Tree {0, 1, 2, 3}

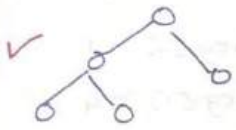


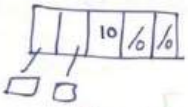
4-ary Tree {0, 1, 2, 3, 4}

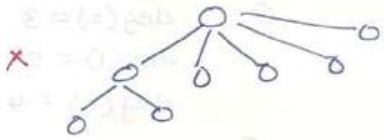




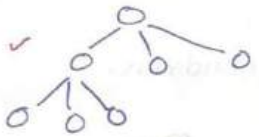


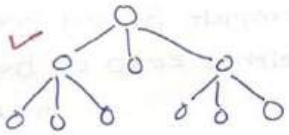


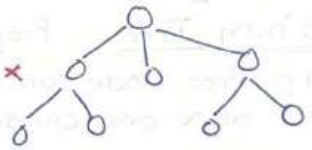




* Strict → 3-ary tree → {0, 3}



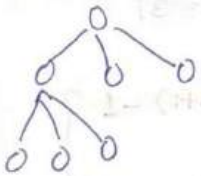




Analysis

h = 2

min



3 + 3 + 1

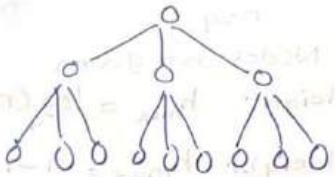
n = 2 × 3 + 1

i = 2

e = 5

5 = 2 × 2 + 1

max



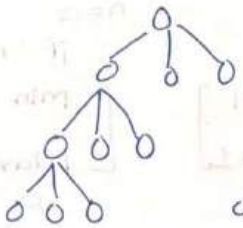
i = 4

e = 9

9 = 2 × 4 + 1

h = 3

min



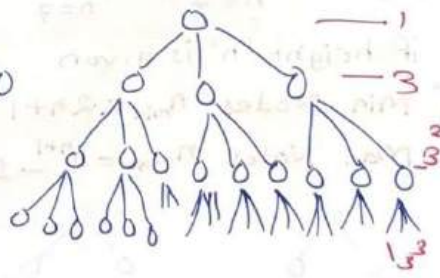
n = 3 + 3 + 3 + 1

= 3 × 3 + 1

i = 3

e = 7

7 = 2 × 3 + 1



1 + 3 + 3² + 3³

i = 13

e = 27

27 = 2 × 13 + 1

⇒ if Height is given

Let m → degree of m-array

$$= 1+3+3^2+3^3$$

$$= 1+m+m^2+m^3+\dots m^h$$

$$= \frac{(m^{h+1}-1)}{m-1}$$

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

Min Nodes $n_{min} = m \cdot h + 1$

Max. Nodes $n_{max} = \frac{m^{h+1}-1}{m-1}$

⇒ if 'n' Nodes are given

Min Height $h_{min} = \log_m [n(m-1)+1] - 1$

Max Height $h_{max} = \frac{n-1}{m}$

⇒ Relation between internal and external node

if degree $m=3$

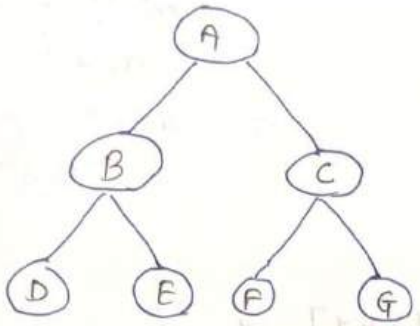
$$e = 2i+1$$

if degree m

$$[e = (m-1)i + 1]$$

* Representation of Binary tree

① Array representation of Binary tree:



if element → i^{th}

Left child → $2 * i$

Right child → $2 * i + 1$

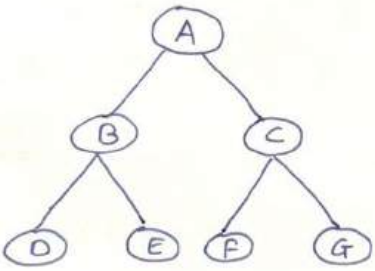
Parent → $\left[\frac{i}{2}\right]$ ⇒ Decimal value.

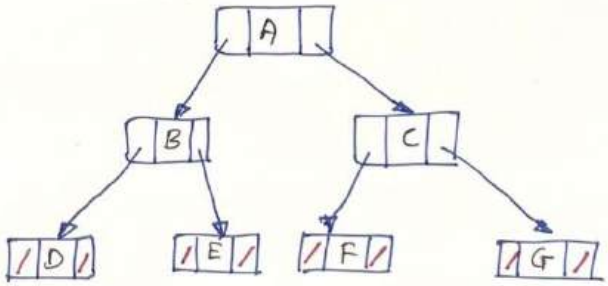
T	A	B	C	D	E	F	G
	1	2	3	4	5	6	7

$\frac{7}{2} = [3.5] = 3$

Element	index (i)	L. child	R. child
A	1	2	3
B	2	4	5
C	3	6	7
	i	$2 * i$	$2 * i + 1$

② Linked representation of Binary tree:





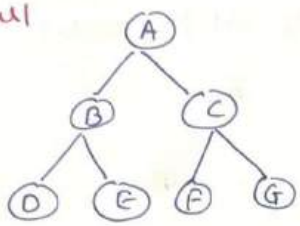
Node

[for 'n' nodes.
Number of NULL spaces = n+1]

```
struct Node
{
    struct Node *lchild;
    int data;
    struct Node *rchild;
}
```

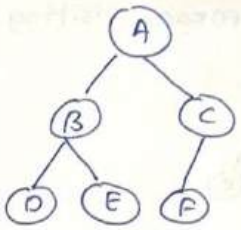
* Full vs Complete Binary Tree

Full



	1	2	3	4	5	6	7
T	A	B	C	D	E	F	G

Complete ✓

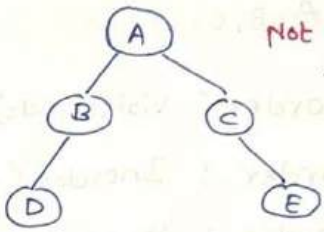


	1	2	3	4	5	6	7
T	A	B	C	D	E	F	

⇒ Full binary tree, is a tree having maximum number of nodes.

⇒ If binary tree is represented in array and filled left to right, when there is no blank space from first to lost element for a tree. then it is called as "Complete Binary tree".

Not comple ✗

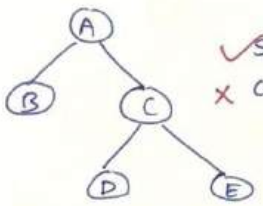


	1	2	3	4	5	6	7
	A	B	C	D	–	–	E

⇒ Every full binary tree is always complete binary tree.

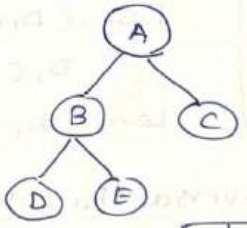
⇒ Every complete tree is need not to be always full binary tree.

* Strict vs Complete Binary Tree



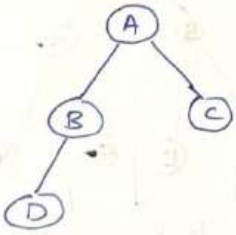
✓ Strict
✗ Complete

	1	2	3	4	5	6	7
T	A	B	C	–	–	D	E



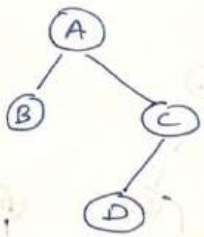
✓ Strict
✓ Complete

	1	2	3	4	5	6	7
T	A	B	C	D	E		



✗ Strict
✓ Complete

	1	2	3	4
T	A	B	C	D

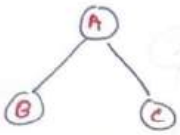


✗ strict
✗ complete

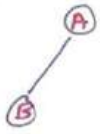
	1	2	3	4	5	6
	A	B	C	–	–	D

Tree Traversals

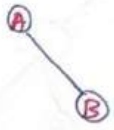
Traversing means visiting all the elements or visiting all the nodes.



Pre: A, B, C
In: B, A, C
Post: B, C, A
Level: A, B, C



Pre: A, B
In: B, A
post: B, A
Level: A, B



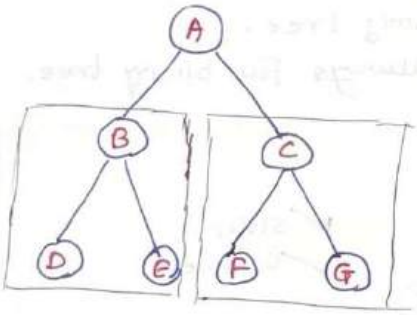
Pre: A, B
In: A, B
post: B, A
Level: A, B

Preorder: visit(node), Preorder(Left subtree), Preorder(right subtree)

Inorder: Inorder(Left), visit(node), Inorder(right)

Post order: Postorder(Left), Postorder(right), visit(node)

Level order: Level by level



Pre: A, (B, D, E), (C, F, G)
A, B, D, E, C, F, G

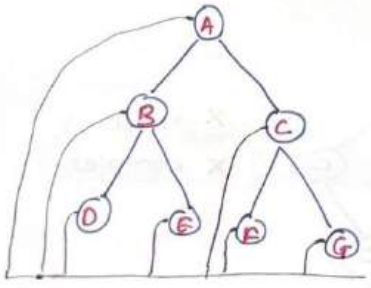
In: (D, B, E), A, (F, C, G)
D, B, E, A, F, C, G

Post: (D, E, B), (F, G, C), A
D, E, B, F, G, C, A

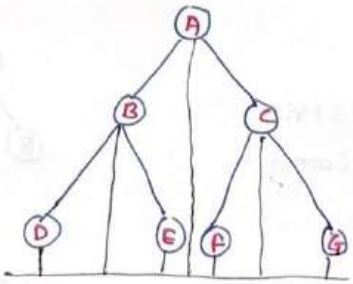
Level: A, B, C, D, E, F, G.

* Binary Tree Transversal Easy Methods

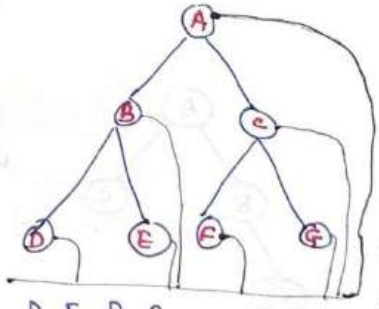
Method ①: Lines should not intersects.



A, B, D, E, C, F, G
Pre-order ⇒



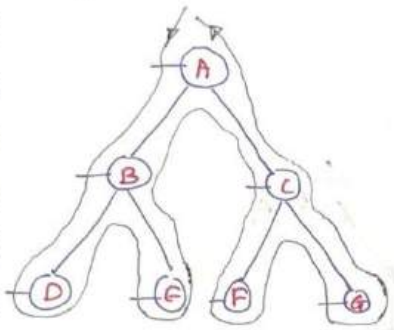
D, B, E, A, F, C, G
In-order ⇒



D, E, B, F, G, C, A
Post-order ⇒

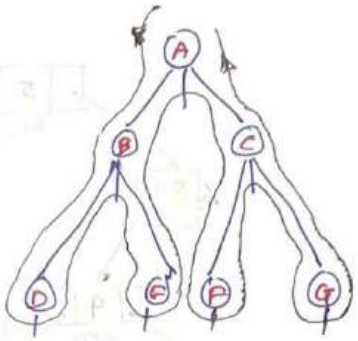
Method 2 →

Preorder



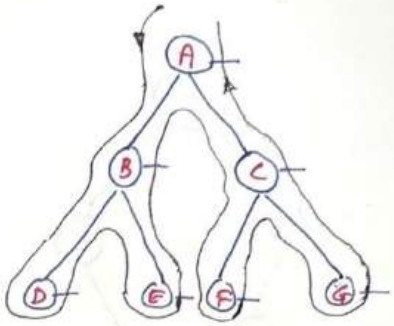
Pre: A, B, D, E, C, F, G

Inorder



In: D, B, E, A, F, C, G

Post order



Post: D, E, B, F, G, C, A.

⇒ Move along the boundary

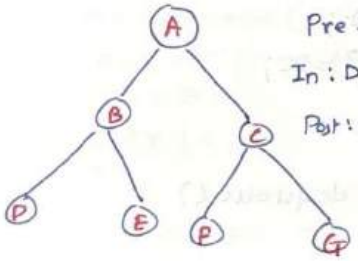
Method 3 →

Point the fingure and, just by observration give the order: move along the boundary.

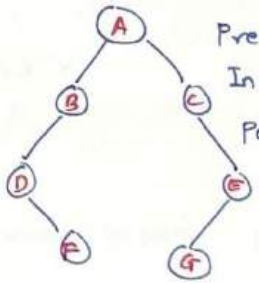
Pre-order

~~Post~~ In order

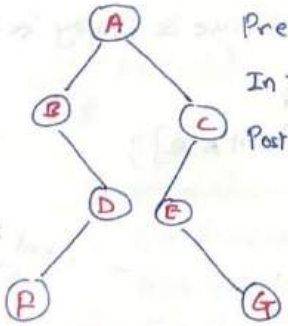
Post-order



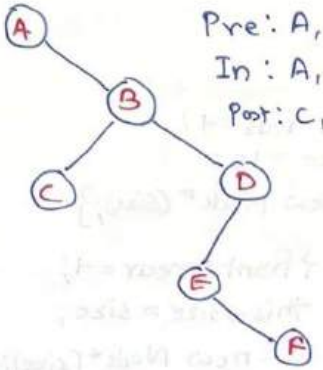
Pre: A, B, D, E, C, F, G
In: D, B, E, A, F, C, G
Post: D, E, B, F, G, C, A



Pre: A, B, D, F, C, E, G
In: D, F, B, A, C, G, E
Post: F, D, B, G, E, C, A.

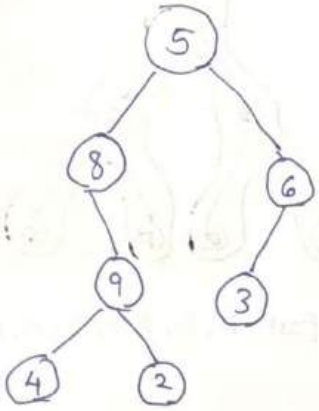


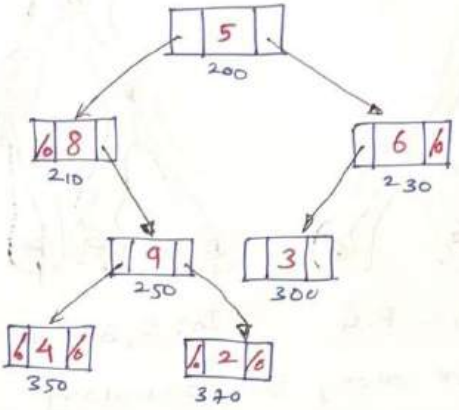
Pre: A, B, D, F, C, E, G
In: B, F, D, A, E, G, C
Post: F, D, B, G, E, C, A.



Pre: A, B, C, D, E, F
In: A, C, B, E, F, D
Post: C, F, E, D, B, A

* Creating Binary Tree using Linked Method.





Q | 200 | 210 | 230 | 250 | 300 | 350 | 370 |

* Code

```
#include <iostream>
#include <stdio.h>
using namespace std;

Class Node
{public:
   Node *lchild;
   int data;
   Node * rchild;
};

Class Queue
{
private:
 int front;
 int rear;
 int size;
 Node ** Q;
public:
 Queue(){front=rear=-1;
         size =10;
         Q=new Node* (size);}
Queue(int size){front=rear=-1;
         this->size = size;
         Q= new Node*[size];}
void enqueue (Node * x)
Node* dequeue ();
int isEmpty (){ return front==rear;}
};
```

```
Void Queue::enqueue (Node *x)
{
 if (rear==size-1)
      cout << " Queue is full "<<endl;
 else
    {  rear ++;
       Q[rear] = x;
    }
}

Node * Queue :: dequeue ()
 {
    Node * x = NULL;
  if (front==rear)
     cout<< " Queue is Empty "<<endl;
  else
    {  front ++
       x= Q[front];
       ~~front~~
    }
    return;          <- need to check
 }
```

```
Class Tree
{public:
  Node * root;
 void  Create Tree();
};

  void  Tree :: Create Tree ()
 {
    Node *p, *t = NULL;
    int x;
    Queue  q(100);
cout printf " Enter root value "
     cin >> x;
     root = new Node;
      root ->data = x
     root -> lchild = nullptr;
     root-> &child = nullptr;
      q.dequeue(root):
 while (!q.isEmpty())
 {
    p=dequeue(&q)
    cout << " Enter the value of leftchild ",
    cin >> x;
    if (x != -1)
    {
      t = new Node;
      t ->data = x;     t-> lchild = t-> rchild = NULL; p
      p-> lchild = t;  q. enqueue (t);
    }
    cout << " Enter the value of right child of;
    cin >> x;
    if (x != -1)
     {
       t = new Node;
       t->data = x;
       t-> lchild = t-> rchild = null;
       p-> rchild = t;
       q.enqueue (t);
     }
   }
}
```

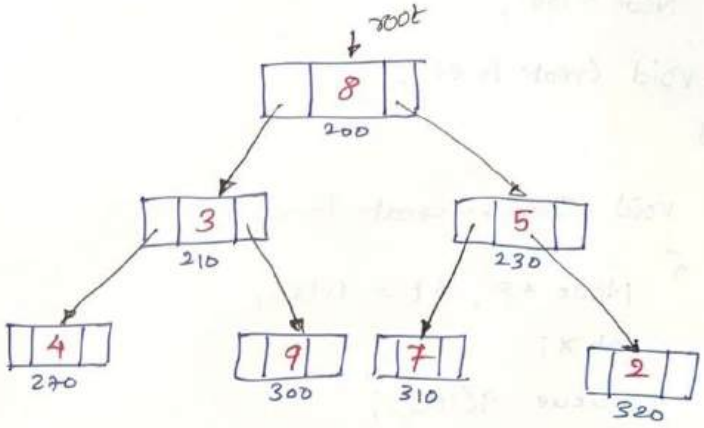
```
int main ()
{
  Tree t;
  t. Create Tree ();
}
```

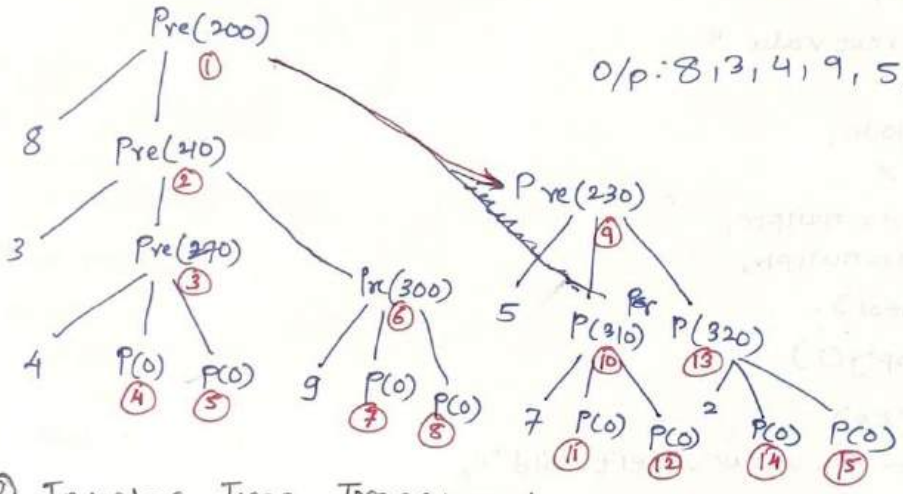
* Tree Transversal (code)

① Preorder Tree Transversal → (Recurssive)

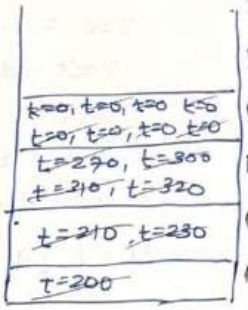
Node, Left Node, Right node.

```
void PreOrder(Node *t)
{
  if (t != NULL)
  {
1.  printf("%d", t->data);
2.  preorder(t->lchild);
3.  Preorder(t->rchild);
  }
```





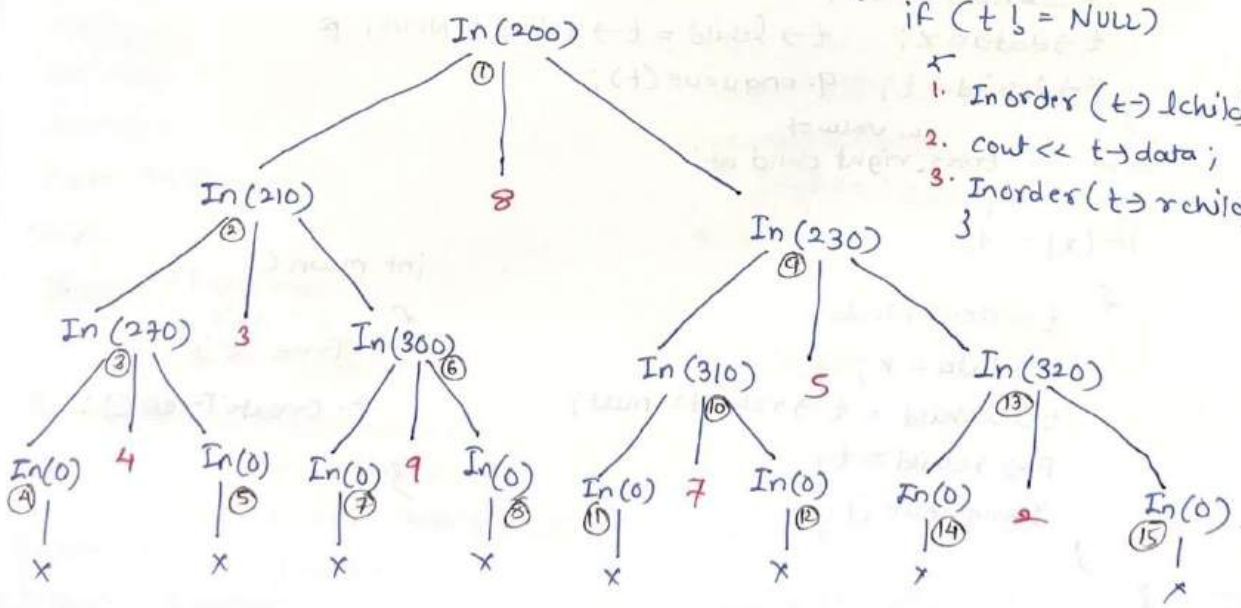
O/p: 8, 3, 4, 9, 5, 7, 2



Stack record

② Inorder Tree Transversal. Left, Node, right

```
void Inorder(Node *t)
{
  if (t != NULL)
  {
1.  Inorder(t->lchild);
2.  cout << t->data;
3.  Inorder(t->rchild)
  }
```



O/P: 4, 3, 9, 8, 7, 3, 2

Iterative

① Pre-order

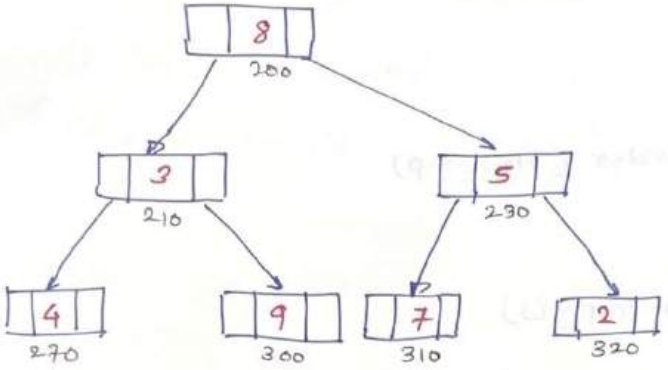
```
~~Void Pre-order~~
void Tree :: iterativePreorder (Node *p)
{
    stack < Node* > stk;
    while (p != nullptr || stk.empty())
    {
        if (p != nullptr)
        {
            cout << p->data << ", " << flush;
            stk.emplace(p);
            p = p->lchild;
        }
        else
        {
            p = stk.pop();
            p = p->rchild;
        }
    }
    cout << endl;
}
```

emplace ⇒ push.

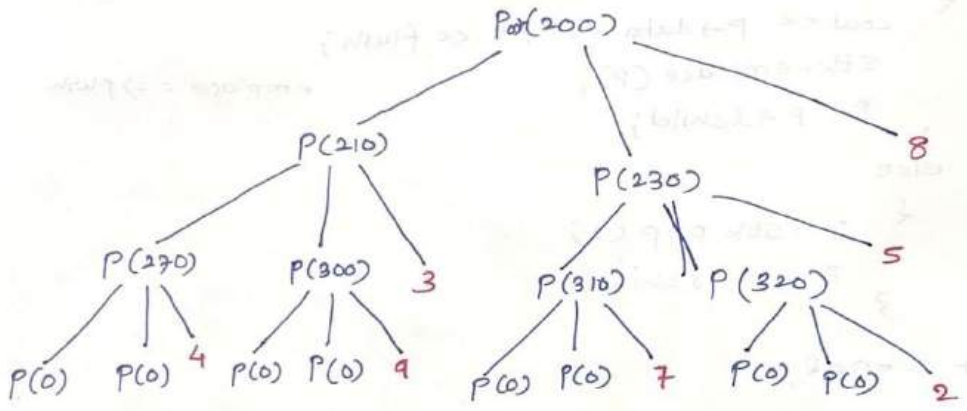
② In-Order

```
void Tree :: iterativeInorder (Node *p)
{
    stack < Node * > stk;
    while (p != nullptr || stk.empty())
    {
        if (p != nullptr)
        {
            stk.emplace(p);
            p = p->lchild;
        }
        else
        {
            p = stk.pop();
            cout << p->data << ", " << flush;
            p = p->rchild;
        }
    }
    cout << endl;
}
```

② Post-order Tree Traversal : Left, right, Node.

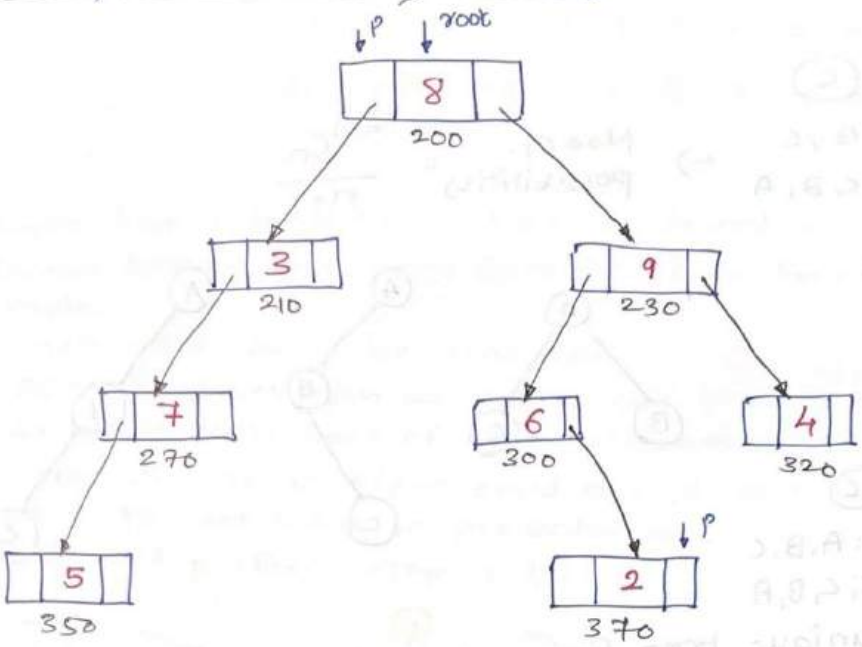



```
Void Postorder (Node *p)
{   if (p)
    {
      Postorder (p→lchild);
      Postorder (p→rchild);
      cout << p→data << ", " << flush;
    }
}
```




```
Void Tree :: iterative Post order ( Node *p)
{
    stack < long int > stk;
    long int temp;
    while ( P != nullptr || stk.empty ())
    {
        if (P != nullptr)
        {
            stk.emplace (long int) p);
            P = P→lchild;
        }
        else
        {
            temp = stk.top ();
            stk.pop ()
            if (temp > 0)
            { stk.emplace (-temp);
              P = ((Node*) temp) → rchild;
            }
            else
            { cout << ((Node*) (-1*temp) → data << ", ";
              P = nullptr;
            }
        }
    }
```

③ Level-order Tree Transversal:



O/P: 8, 3, 9, 7, 6, 4, 5, 2

Q

200	210	230	270	300	320	350	370

```
Void Tree:: Levelorder (Node *p)
{
    queue <Node*> q;
    Cout << p->data << " , " << flush;
    q.emplace (p);
    while (! q.empty ())
    {
        p = q.front ();
        q.pop ();
        if (p->lchild)
        {
            cout << (p->lchild)->data << " , " << flush;
            q.emplace (p->lchild);
        }
        if (p->rchild)
        {
            cout << (p->rchild) -> data << " , " << flush;
            q.emplace (p->rchild);
        }
    }
}
```

⇒ for all transversal, height of stack is depedent on height of a tree

size = height + 2

⇒ Total calls = 2n + 1

n = no. of nodes

⇒ Time complexity = order of n

T = O(n)

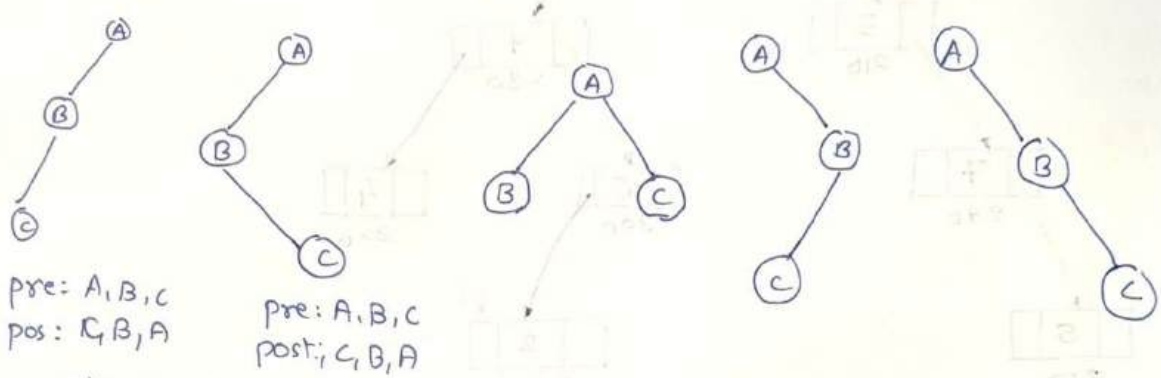
* Can we generate Tree from Transversal :

$n = 3$

(A) (B) (C)

pre-order – A, B, C
Post-order = C, B, A

$\Rightarrow$ No of possibilities $= \frac{2nC_n}{n+1}$



pre: A, B, C
pos: C, B, A

pre: A, B, C
post: C, B, A

$\Rightarrow$ We want a unique tree for a given transversal.

Conclusion:

① If you just give pre-order or postorder or inorder, then you can not generate unique tree.
There are $\frac{2nC_n}{n+1}$ trees are possible.

② If lets say two transversal pre-order and post-order are given then also more than one combination of tree is possible. hence you can not generate unique tree.

③ To generate a unique tree, we need combination of either preorder and In-order or postorder and Inorder.
→ In-order can help you to find root and what should go on left side and what should go on right side.
→ Thats why In-order is mandatory.

preorder or postorder or In-order } $\frac{2nC_n}{n+1}$ preorder & postorder] 1+

① preorder + Inorder
② Postorder + Inorder] 1

unique.

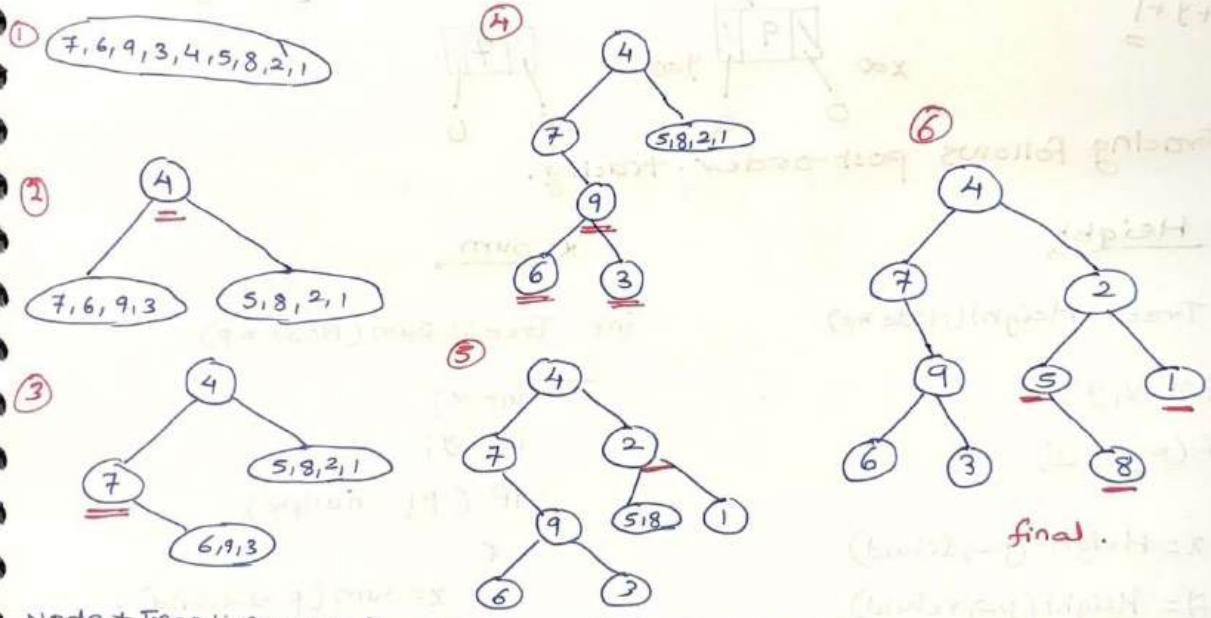
* Generating Tree from Transversal.

Pre-order → 4, 7, 9, 6, 3, 2, 5, 8, 1

In-order → 7, 6, 9, 3, 4, 5, 8, 2, 1

(*)

1) Create first node with all values is In-order.
2) i) Search each value starting from first from Pre-order in In-order node.
   ii) then that value be first root.
   iii) split the values such as all values in Left to root will go on Left child node of root and all the values right to root will go in right child nod of root.
3) then go to next value in pre-order and search that value in node and perform step 1 & 2.




```
Node * Tree :: generate from Transversal ( int * inorder, int * preorder, int instart, int inEnd)
{
    static int preIndex = 0;
    if (instart > inEnd)
        return nullptr.
    Node * node = new Node (preorder[preIndex++]);
    if (instart == inEnd)
        return Node;
    int splitIndex = searchInorder(inorder, instart, inEnd, node->data);
    node->lchild = generate from Transversal(inorder, pre-order, instart, splitIndex - 1);
    node->rchild = generate from Transversal(inorder, pre-order, splitIndex + 1, inEnd);
    return node;
}
```

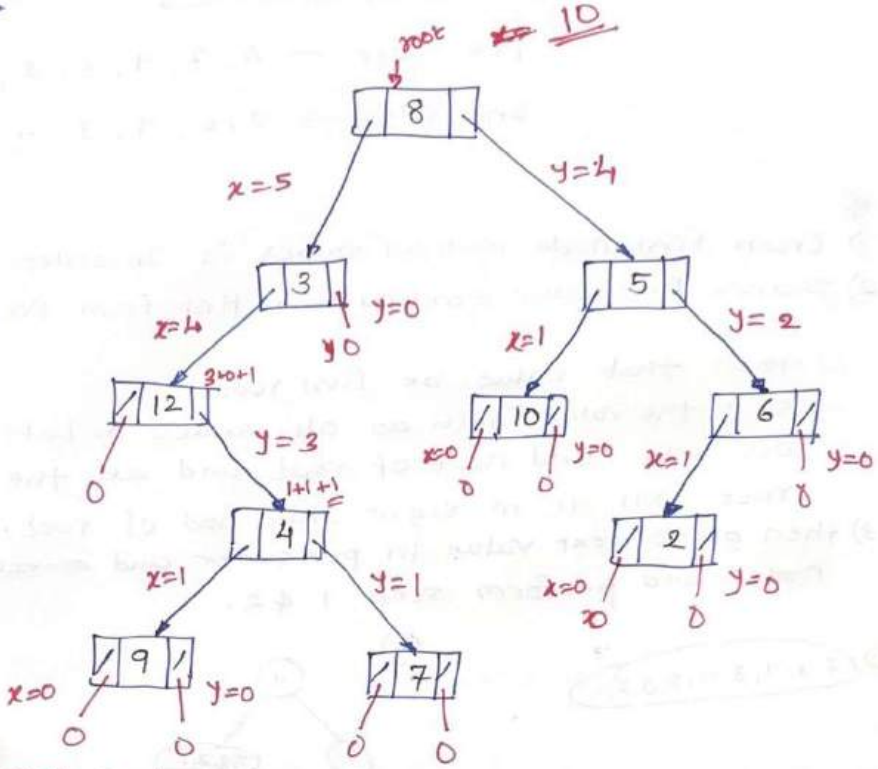
see .txt documentation for more.

* Counting Nodes:

```
int count(Node *p)
{
 int x, y;
 if (p != NULL)
 {
 1. x = count(p->lchild)
 2. y = count(p->rchild)
 3. return x+y+1;
 }
 return 0;
}
```

$x+y+1$

=



⇒ Tracing follows post-order tracing.

* Height

```
int Tree:: Height(Node *p)
{
 int x, y;
 if (p != NULL)
 {
  x = Height(p->lchild)
  y = Height(p->rchild)
  if (x > y)
      return x+1;
   else
      return y+1;
 }
 return 0;
}
```

* Sum

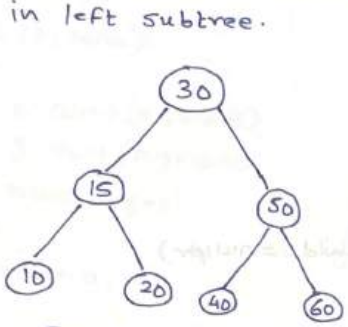
```
int Tree:: Sum(Node *p)
{
  int x;
  int y;
  if (p != nullptr)
  {
    x = Sum(p->lchild);
    y = Sum(p->rchild);
    return x+y + p->data;
  }
  return 0;
}
```

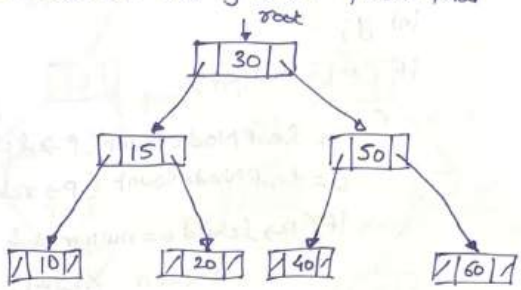
* leaf Node Count →

```
int Tree:: leafNodeCount (Node *p)
{
    int x;
    int y;
    if (p != nullptr)
    {
        x = leafNodeCount(p->lchild)
        y = leafNodeCount (p->rchild)
        if (p->lchild == nullptr && p->rchild == nullptr)
                return x+y+1;
        else
                return x+y;
    }
    return 0;
}

Void Tree:: DestroyTree (Node *p)
{
    if (p != nullptr)
    {
        DestroyTree (p->lchild);
        DestroyTree ( p->rchild);
        delete p;
    }
}
```

Section 16 : Binary Search Tree

*Binary Search Tree : It's a binary tree in which for every node, all the elements in the right subtree are greater than that in left subtree.





Inorder: 10, 15, 20, 30, 40, 50, 60

* Properties:

① No duplicate elements

② Inorder gives sorted order.

③ No. of B.S.T. for 'n' Nodes = $T(n) = \frac{^{2n}C_n}{n+1}$

* Searching in Binary Search Tree:

```
Node* Search(Node *t, int key)
{
   while (t != NULL)
   {
      if(key == t->data)
          return t;
      elseif (key < t->data)
          t= t->lchild;
      else
          t = t->rchild;
   }
   return NULL;
}
```

```
Node * Rsearch(Node *t, int key)
{
   if (t== NULL)
       return NULL;
   if (key == t->data)
       return t;
   else if (key < t->data)
       return Rsearch(t->lchild, key)
   else {
       return Rsearch(t->rchild, key)
   }
}
```

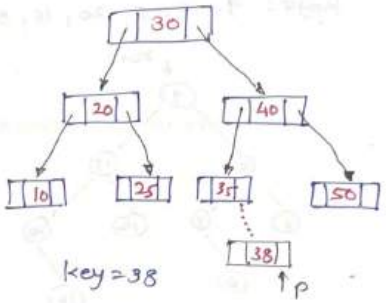
* Time taken for searching the element depends upon height of Tree

$T \rightarrow O(h)$.

$\log n \leq h \leq n$

* Inserting in a Binary Search Tree

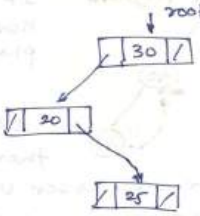
```
Void Insert (Node *t, int key)
{
  Node *r=NULL, *p;
while (t!=NULL)
{
   r=t;
  if (key == t->data)
        return;
  elseif (key < t->data)
        t= t->lchild;
  else
       t = t->rchild;
}
   p= new Node;
   p->data = key ; p->lchild = p->rchild =NULL;
   if(p->data < r->data)
        r->lchild = p;
   else
        r-> rchild=p;
}
```




```
// root (t) is empty)
if (t==nullptr)
 { p=new Node;
  p->data = key;
 p->lchild = nullptr;
 p->rchild=nullptr;
   root=p;
   return;
}
```

* Recursive

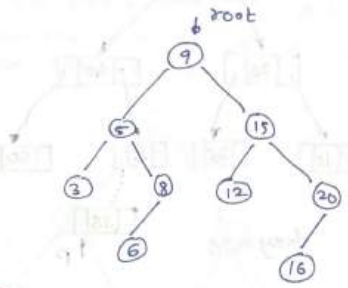
```
Node *insert(Node *p, int key)
{
  Node *t;
  if (p==NULL)
  {
    t= new Node;
   t->data = key;
   t->lchild = t->rchild=NULL;
   return t;
  }
  if(key < p->data)
      p->lchild= insert(p->lchild,key);
  else if( key > p->data)
     p->rchild=insert(p->rchild, key);

  return p;
}
```




```
int main()
{
  Node * root =NULL;
  root = insert (root, 30);
        insert (root, 20);
        insert (root, 25);
}
```

* Creating Binary Search Tree:

keys: 9, 15, 5, 20, 16, 8, 12, 3, 6.



We need to insert n element each time i.e. n times.

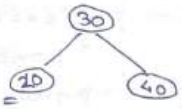
Search → T(→ O(log n)

T → O(n.log n)

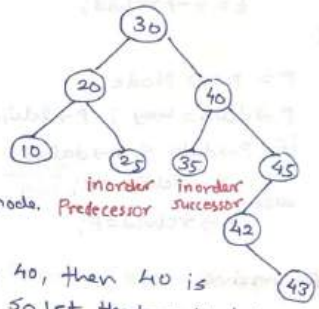
* Delete Binary Search Tree:

(1)

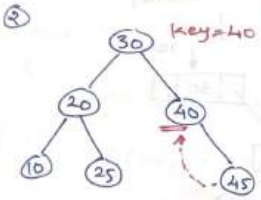
key=10



→ So if we are deleting 20, then there is nothing below it. It is a leaf node. then simply delete it.

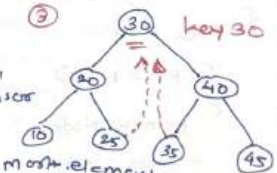


(2)



If we are deleting 40, then 40 is having one child. So let that child take place and delete 40.

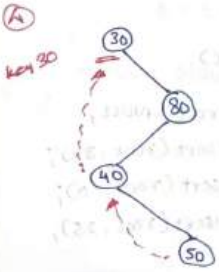
(3)



If we are deleting 30, then its inorder predecessor or inorder successor will go to its place.

Inorder predecessor → Left then Right-Right most element.
Inorder presuccessor → Right then Left-Left most element.

(4)



Now, if we are deleting 30 and we need inorder successor, then 40 will go to its place. but 40 has one child 50.

That child will go to place of 40.

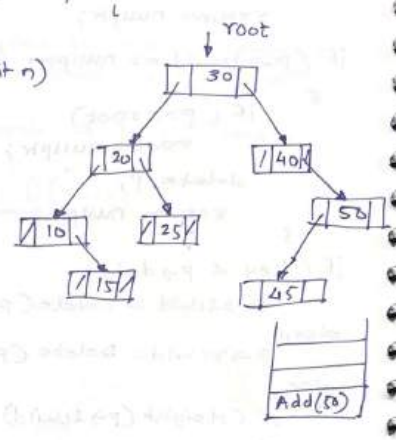
We need more than one modification.

```
Node * BST :: Delete (Node *p, int key)
{
  Node *q;
  if (p==nullptr)
      return nullptr;
  if (p->lchild == nullptr && p->rchild == nullptr)
  {
      if (p==root)
          root = nullptr;
      delete p;
      return nullptr;
  }
  if (key < p->data)
      p->lchild = Delete(p->lchild, key);
  elseif
      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;
}

int BST:: Height(Node *p)
{ int x; int y;
  if (p== nullptr)
      return 0;
  x= Height (p->lchild);
  y= Height (p->rchild);
  return x>y ? x+1 : y+1;
}

Node *BST :: InPre (Node *p)
{ while(p&& p->rchild != nullptr)
  {
      p=p->rchild;
  }
  return p;
}

Node *BST :: Insucc (Node *p)
{
  while (p&& p->lchild != nullptr)
      p=p->lchild;
  }
  return p;
}
```

* Generating BST from Preorder:

Pre	30	20	10	15	25	40	50	45
	0	1	2	3	4	5	6	7

i

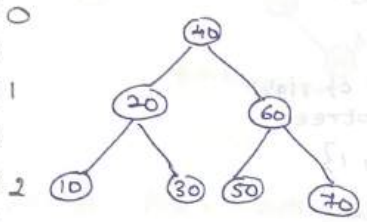



```
void BST::. CreateFromPreorder(int *pre, int n)
{
  // create root node
  int i=0;
  root = new Node;
  root -> data = pre [i++];
  root -> lchild = nullptr;
  root -> rchild = nullptr;

  // Iterative steps
  Node *t;
  Node * p =root;
  stack <Node * > stk;

  while ( i <n)
  {
    // Left child case
   iF ( pre [i] < p -> data)
  {
     t=new Node;
     t-> data = pre [i++];
     t-> lchild = nullptr;
     t-> rchild = nullptr;
     p-> lchild = t;
     stk.push (P);
      p=t;
  }
  else
  {     // Right child cases
    if (pre[i] > p->data && pre[i] < stk.empty() ? 32767 : stk.top()->data)
    {
        t= new Node;
        t-> data = pre [i++];
        t-> lchild = nullptr;
        t-> rchild = nullptr;
        p->rchild = t;
         p=t;
    }
   else {  p= stk.pop();
           stk.pop ();
   }
  }
}
```

```
//BST from Preorder transversal
cout << " BST from Preorder: " << flush;
int pre[]= {30, 20, 10, 15, 25, 40, 50, 45};
int n = sizeof(pre) / sizeof (pre[0]);
  BST b;
b. CreateFromPreorder (pre, n);
b. Inorder (b.getRoot()); cout << endl;
```

* Drawback of Binary search Tree

Keys: 40, 20, 30, 60, 50, 10, 70

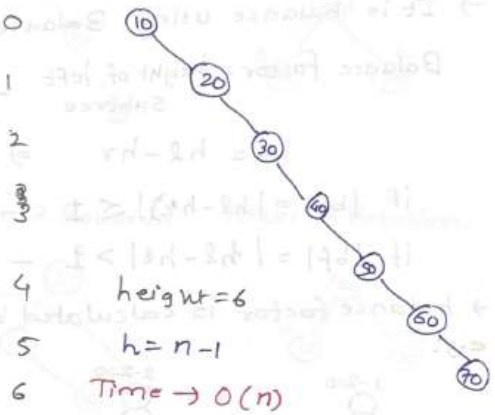


height = 2

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

Time → O(log n)

Keys: 10, 20, 30, 40, 50, 60, 70



height = 6

$h = n - 1$

Time → O(n)

⇒ Time is depedent on the height.

⇒ We can not control the height of Binary search tree. As it is depedent on order of insertion of key

⇒ We as a developer don't have any control on order of insertion as it is depedent on the end user which uses it.

⇒ So drawback of Binary search tree is we don't have any control over the height.

* Section 17: AVL Trees

→ This are height balanced search trees.
→ It is balance using Balance factor.

Balance factor = height of left subtree − height of right subtree.

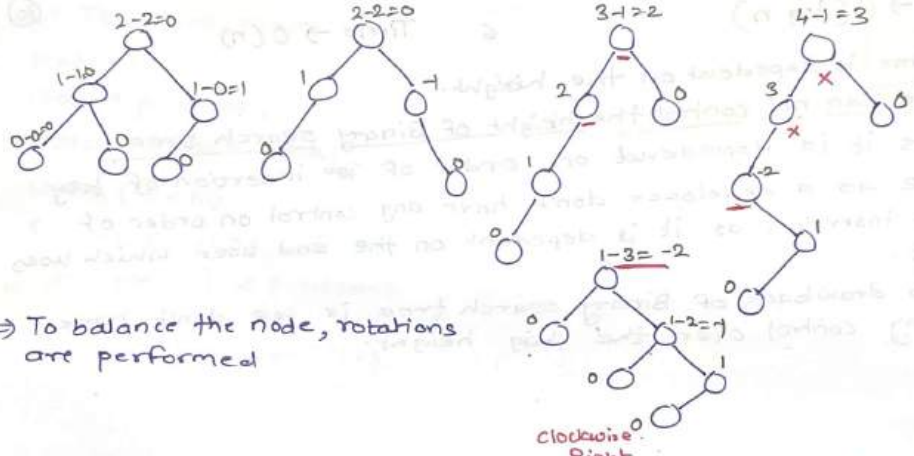
$$bf = hl - hr \Rightarrow \{-1, 0, 1\}$$

if $|bf| = |hl - hr| \leq 1$ — balanced

if $|bf| = |hl - hr| > 1$ — imbalanced

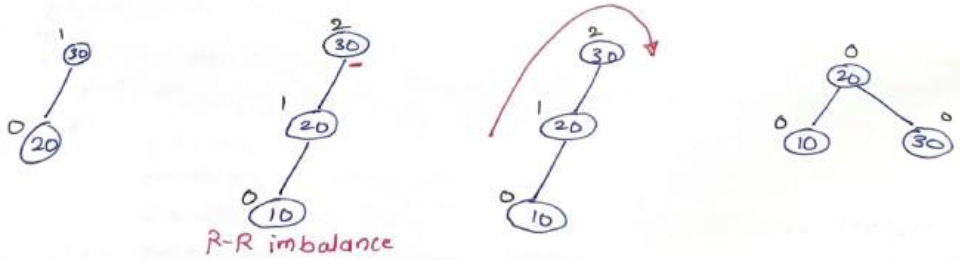
→ balance factor is calculated on every node.

e.g:



⇒ To balance the node, rotations are performed

Initial | After Insertion(10) | Perform L-L Rotation | After LL Rotation



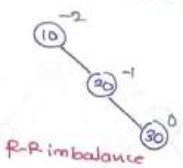
R-R imbalance

LL-Rotation, RR-Rotation } → Single Rotation

LR-Rotation, RL-Rotation } → Double Rotation

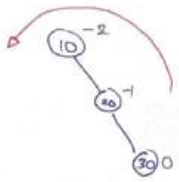
Initial

After Inserting 30

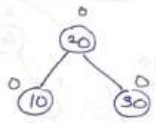


R-R imbalance

Perform RR Rotation

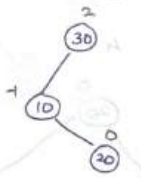


After RR Rotation



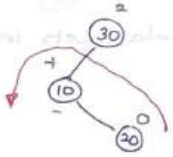
Initial

After Inserting 20

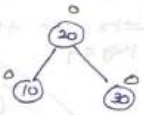


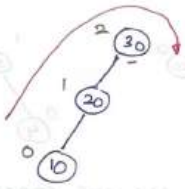
LR imbalance

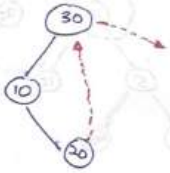
Perform LR Rotation



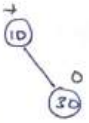
After LR Rotation.



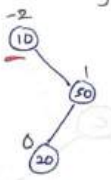




Initial

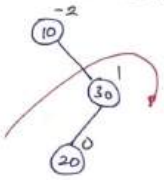


After Inserting 20

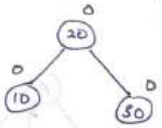


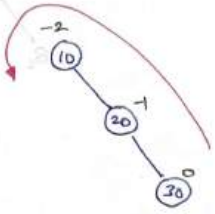
R-L imbalance

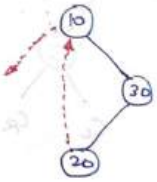
Perform RL Rotation



After RL Rotation



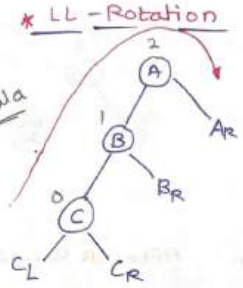


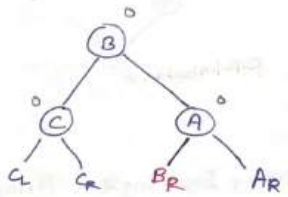


* Formula of Rotations for Insertion:

* LL - Rotation

Formula

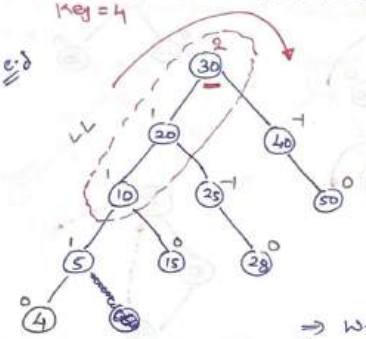


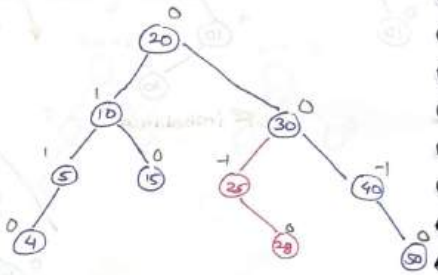


→ Lets use this formula. Lets insert 4

key = 4

e.g

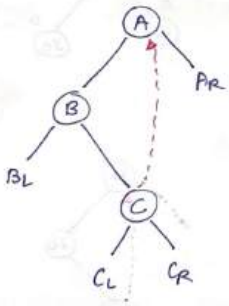


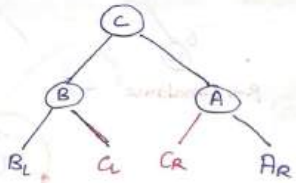


⇒ We are concerned only first three node from imbalance node for L-L or any Rotation fixing.

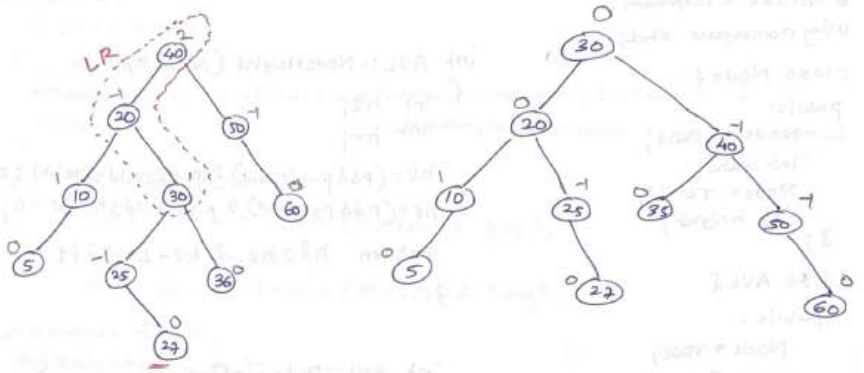
② L-R Rotation

Formula





e.g: key=28 Added



AVL Trees

```
#include <iostream>
using namespace std;
class Node{
public:
    Node *lchild;
    int data;
    Node* rchild;
    int height;
};
class AVL{
  public:
    Node * root;
    AVL(){root=nullptr;}
// Helper Method for insert
    :
    :
// Insert
Node* rInsert( ---- );
// Transversal
    :
    :
};
```

```
Node * AVL:: LLRotation(Node *p)
{
  Node * pl= p->lchild;
  Node * plr= pl->rchild;

  pl->rchild = p;
  p-> lchild = plr;

  // Updat Height

p->height =NodeHeight(p);
Pl->height = NodeHeight(pl);

// Update root
  if (root == p)
       root = pl;

  return Pl;

}
```

```
int AVL:: NodeHeight (Node *p)
{
  int hl;
  int hr;
  hl = (p&& p->lchild) ? p->lchild->height : 0;
  hr =(p&& p->rchild) ? p->rchild->height : 0;

  return  hl > hr  ? hl+1 : hr+1;
}


int  AVL:: BalanceFactor (Node *p)
{
  int hl;
  int hr;
hl=(p&&p->lchild) ? p->lchild->Height : 0;
hr=(p&&p->rchild)? p->rchild->Height : 0;

  return  hl -hr;
}

Node *AVL :: RRRotation (Node *p)
{
    Node * pr =  p->rchild;
    Node * prl =  pr->lchild;

     pr-> lchild = p;
     p-> rchild = prl;

     // Update Height
     p->height = NodeHeight (p);
     pr->height = NodeHeight (pr);

     // Update root
      if (root == p)
           root= pr;

       return pr;
}
```

```
Node * AVL::rInsert(Node *p, int key)
{
  Node *t;
  if(p==nullptr)
  {
    t=new Node;
    t->data=key; t->lchild=t->rchild=nullptr; t->height=1
    return t;                    starting height from 1 onwards instead of 0.
  }
if(key < p->data)
      p->lchild = rInsert(p->lchild, key);
else if(key > p->data)
      p->rchild=rInsert(p->rchild, key);

// Update Height;
 p->Height = NodeHeight(p);

// Balance factor and Rotation.
 if(Balancefactor(p)==2 && Balancefactor(p->lchild)==1)
       return LLRotation();
else if(Balancefactor(p)==2 && Balancefactor(p->lchild)==-1)
       return LRRotation();
else if(Balancefactor(p)=-2 && Balancefactor(p->rchild)==-1)
     return RRRotation();
else if(Balancefactor(p)==-2 && Balancefactor(p->rchild==1)
       return RLRotation();
}
return p;
}
int main()
{
// LL Rotation
 AVL tll;
tll.root=tll.rInsert(tll.root,30);
tll.root=tll.rInsert(tll.root,20);
tll.root=tll.rInsert(tll.root,10);
 tll.Inorder();

// RR Rotation
 AVL trr;
  trr.root=trr.rInsert(trr.root,10);
  trr.root=trr.rInsert(trr.root,20);
  trr.root=trr.rInsert(trr.root,30);
```

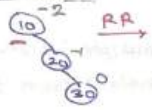
```
void AVL::Inorder(Node *p)
{
  if(p)
  {
    Inorder(p->lchild);
    cout << p->data;
    Inorder(p->rchild);
  }
}
```

* Generating AVL Trees

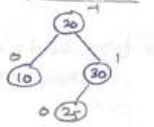
keys: 10, 20, 30, 25, 28, 27, 5

Insert 10,20

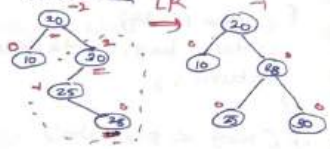
Insert 30



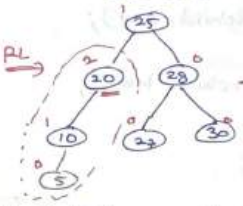
Insert 25



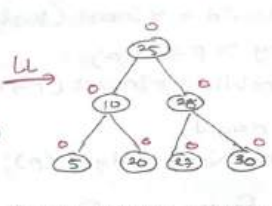
Insert 28



Insert 27



Insert 5

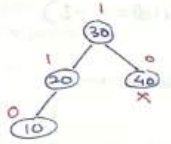


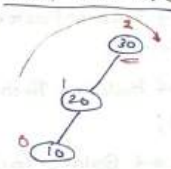
Direction of insertion is "RL" (27)

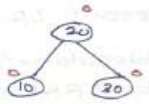
* Creat Binary search tree for same element & compare Height.

* Rotations for Deletion from AVL Trees:

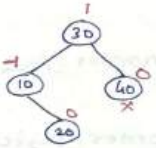
L1

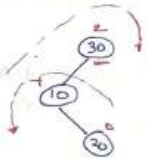


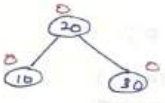




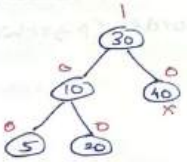
L-1

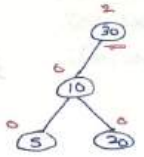




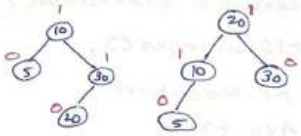


L0





L1 or L-1



⇒ If the element is deleted from right → L1, L-1, L0 rotations

⇒ Same, if the element is deleted from left → R1, R-1, R0 rotations.

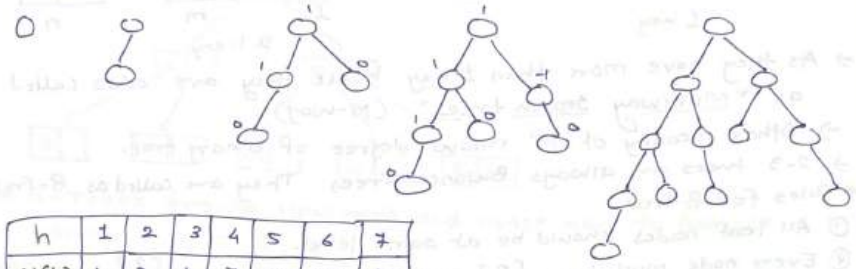
* Height Analysis of AVL Trees.

→ If Height is given find

① Max Node $\left[n_{max} = 2^h - 1\right]$

② Min. Node. (need to derive formula from observation.)

Let 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(h)	1	2	4	7	12	20	33

$$\left[\text{Min Node } N_{Min}(h) = \begin{cases} 0 & \text{if } h=0 \\ 1 & \text{if } h=1 \\ N(h-2) + N(h-1) + 1 & \text{for any other } h \end{cases}\right]$$

→ If Nodes are given find

① Min Height $\left[h_{min} = \log_2(n+1)\right]$

② Max Height $\left[h_{max} = \text{need to locate from above table.}\right]$

e.g: If n= 2 h=2

If n= 3 h=2

n= 4 h=3

n= 5 h=3

n= 12 h=5

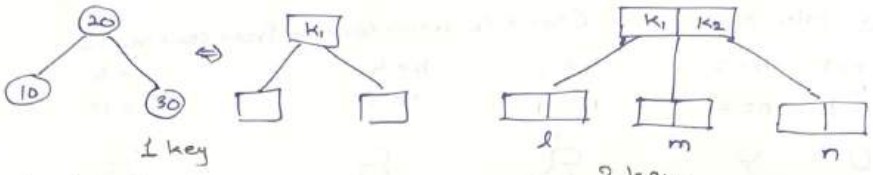
n=15 h=5

n=16 h=5

n=20 h=6

Section 18: Search Trees

→ 2-3 Trees are search trees like binary search trees. Same concept is extended to multiple keys.



→ As they have more than 1 key hence they are also called as "Multiway Search trees" (M-way)

→ Other meaning of 'M' means degree of binary tree.

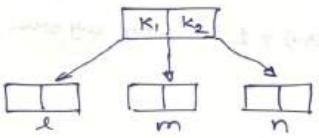
→ 2-3 trees are always Balanced Trees. They are called as B-Tree.

* Rules for B-Tree:

① All leaf nodes should be at same level.

② Every node must have $\left\lceil \frac{n}{2} \right\rceil$ children. n = degree $\left\lceil \frac{3}{2} \right\rceil = 2$

* 2-3 * Trees



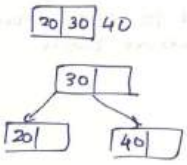
→ $K_1 < K_2$
→ $l < K_1$
→ $n > K_2$
→ $K_1 < m < K_2$

* Creation of 2-3 Trees:

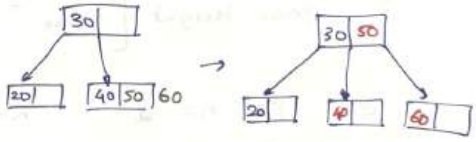
Keys: 20, 30, 40, 50, 60, 10, 15, 70, 80, 90

Insert 20, 30

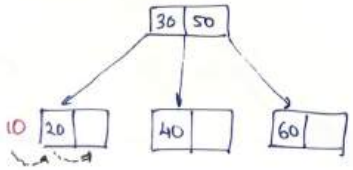
Insert 40



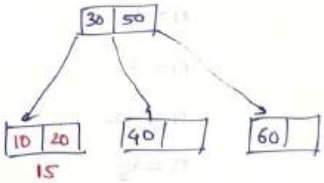
Insert 50, 60

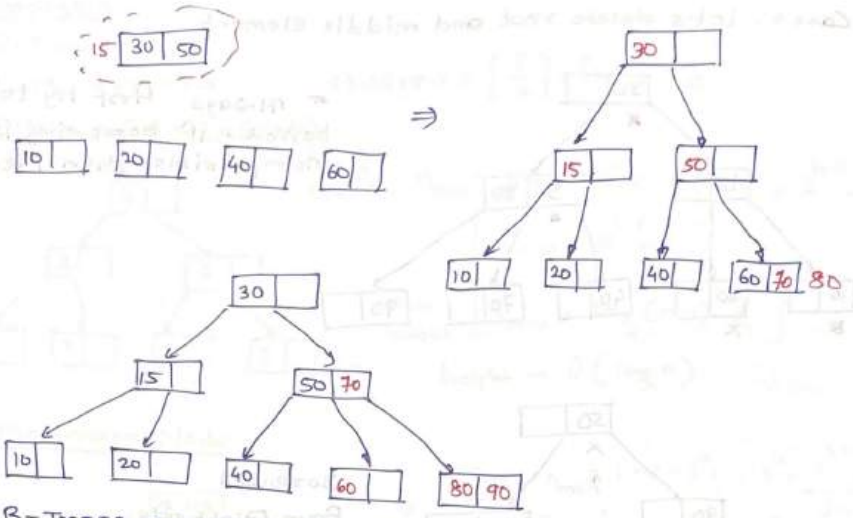


Insert 10



Insert 15





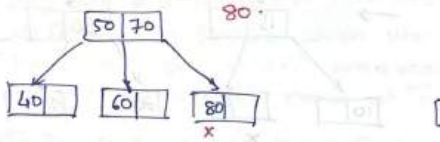
⇒ B-Trees grows upwards and hence able to balance.

* Deletion in 2-3 Trees

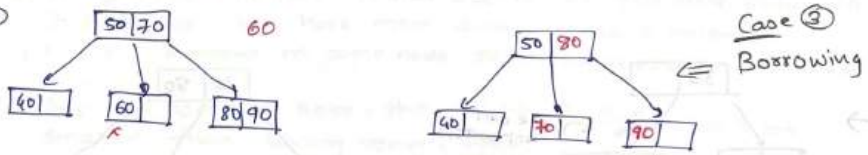
Case ①: Simply Delete.

Case ②: Delete and Merge

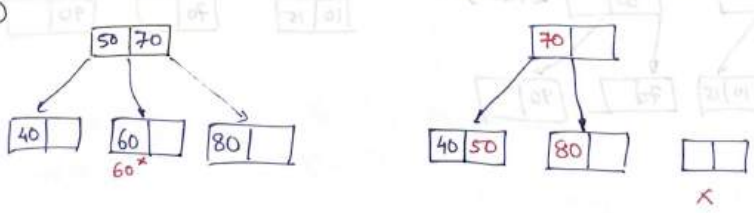
1)



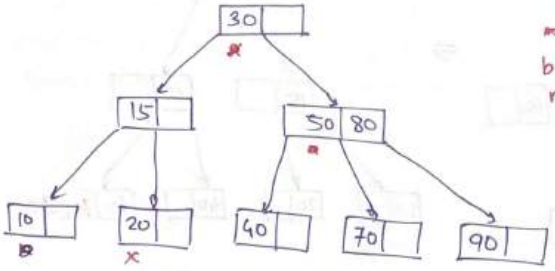
2)



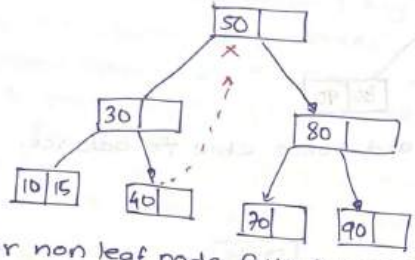
②



Case 4: Let's delete root and middle element.



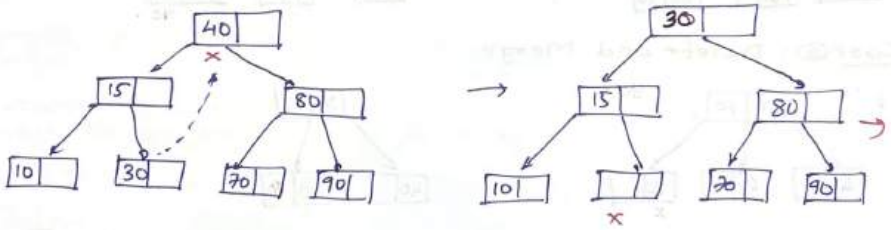
* Always first try to borrow. if borrowing is not possible then Merge.

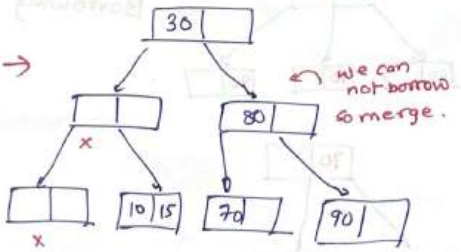


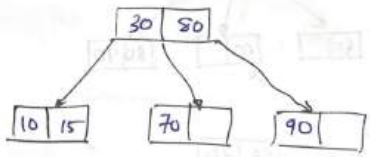
Borrowed from Right side.

~~first try~~

⇒ for non leaf node follow procedure like Binary Search tree. If node is deleted then its predecessor or successor should take its place.





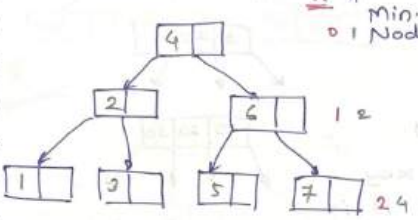


Analysis

→ B-Tree

→ Here degree = 3 children = $\left[\frac{3}{2}\right] = [1.5] = 2$

① Minimum Elements (Node)



h n

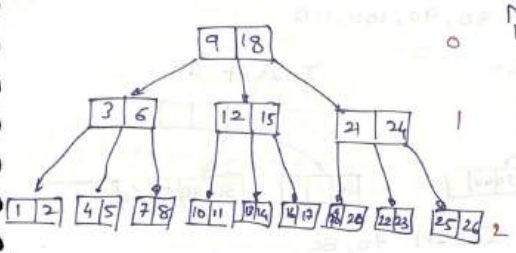
Min. Node $n_{min} = 1 + 2 + 2^2 + \dots + 2^h = 2^{h+1} - 1$

$$\left[n_{min} = 2^{h+1} - 1 \right]$$

Max. height $\left[h_{max} = \log_2 (n-1) - 1 \right]$

height → $O(\log_2 n)$

② Maximum Node:



Max Node $n_{max} = 1 + 3 + 3^2 + \dots + 3^h = \frac{3^{h+1} - 1}{3-1}$

$$\left[n_{max} = \frac{3^{h+1} - 1}{3-1} \right]$$

$h_{min} = \log_3 [n(3-1) + 1] - 1$

$O(\log_3 n)$

⇒ So this proves that height of 2-3 tree or B-Tree is always in order of $\log n$.

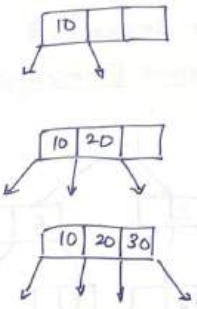
⇒ Why we ~~use~~ B-Trees are use over Binary Tree ?

→① In binary tree when we reach at one node. then we have only one value. whereas in B-Tree multiple value we get when we reach at any particular node.

→② In B-Tree we have faster access for the next value so it is faster. We have more direct access elements. i.e. More element in same node so direct access between them.

→③ for some 'n' keys., the height of B-Tree will be smaller than Binary search tree.

* 2-3-4 Trees:



→ It is B-Tree of degree = 4

→ Every node must have $\lceil \frac{4}{2} \rceil = 2$ children

→ All leaf node should be at same level.

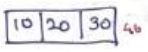
→ Every node can have minimum two key or maximum four key.

→ Every node can have either two key or three key or four key hence it is called as 2-3-4 Trees.

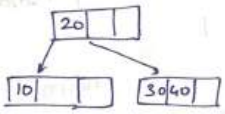
* Insertion in 2-3-4 Trees.

key: 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110.

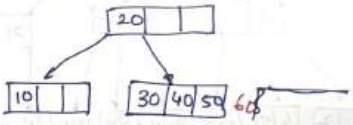
Insert 10, 20, 30



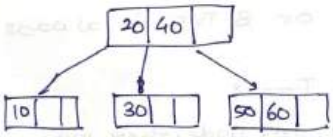
Insert 40



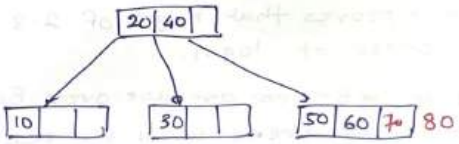
Insert 50



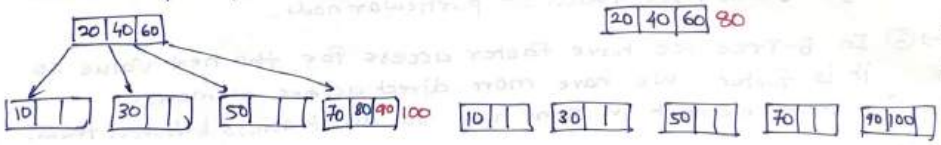
Insert 60

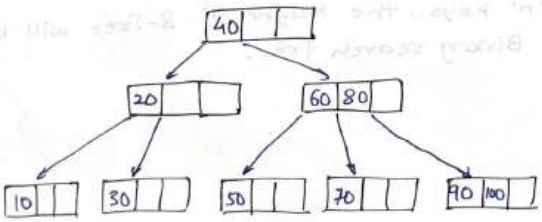


Insert 70, 80



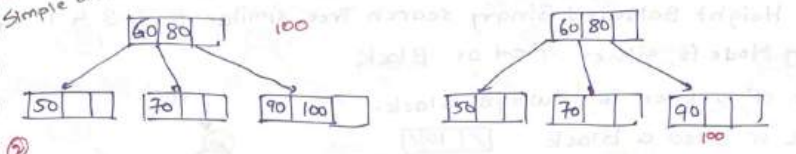
Insert 80, 90, 100



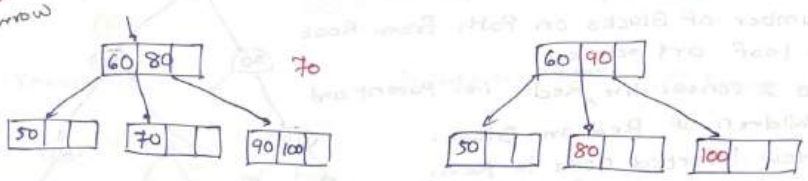


* Deletion in 2-3-4 Trees.

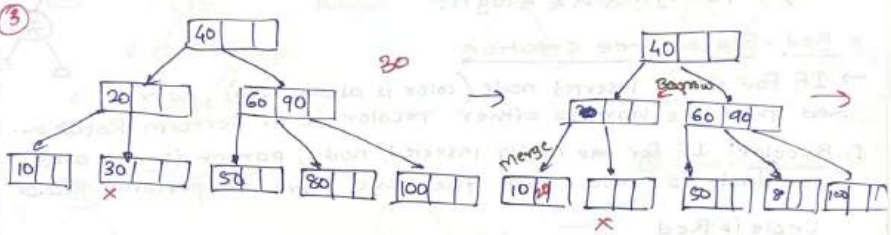
① Simple delete

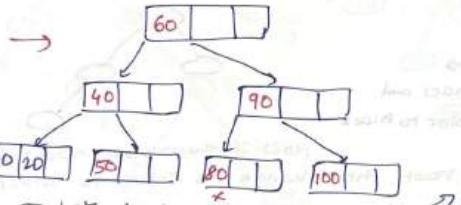


② Borrow

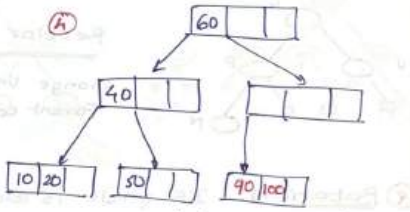


③

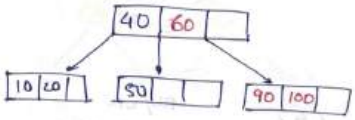




④ Lets delete 80, can not borrow then merge



→ Cannot ~~merge~~ borrow than merge.



* Red - Black Tree

→ It's Height Balanced Binary search Tree, similar to 2-3-4 Tree.

→ Every Node is either Red or Black

→ Root of a tree is always Black.

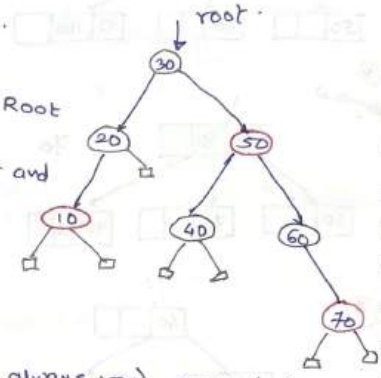
→ NULL is also a Black

→ Number of Blacks on Path from Root to leaf are same.

→ No 2 consecutive can be Red. i.e. Parent and children of Red are Black.

→ New inserted node is Red.

→ Height is $\log n \leq h \leq 2\log n$.

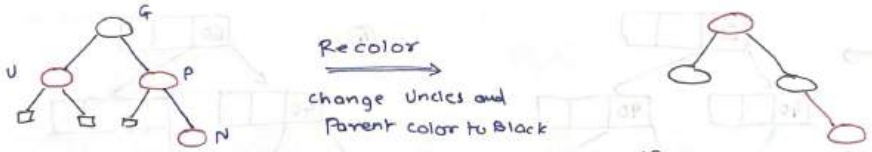


* Red - Black Tree creation:

→ If for newly inserted node (color is always red), parent is Red then we have to either recolor it or perform Rotation.

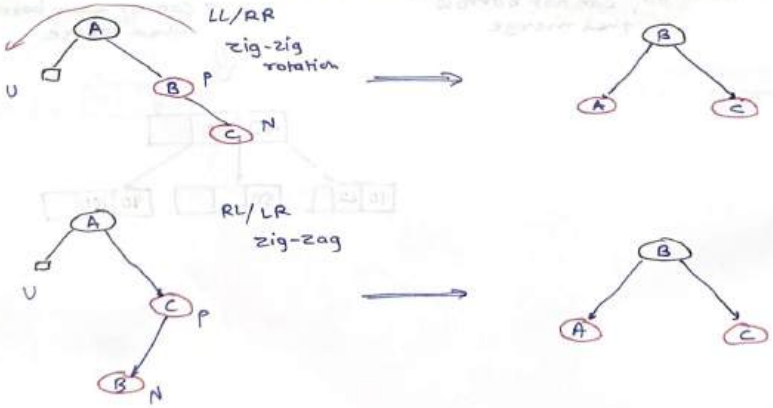
① **Recolor**: If for newly inserted node, parent is red and Uncle is also red then we have to perform Recolor.

Uncle is Red

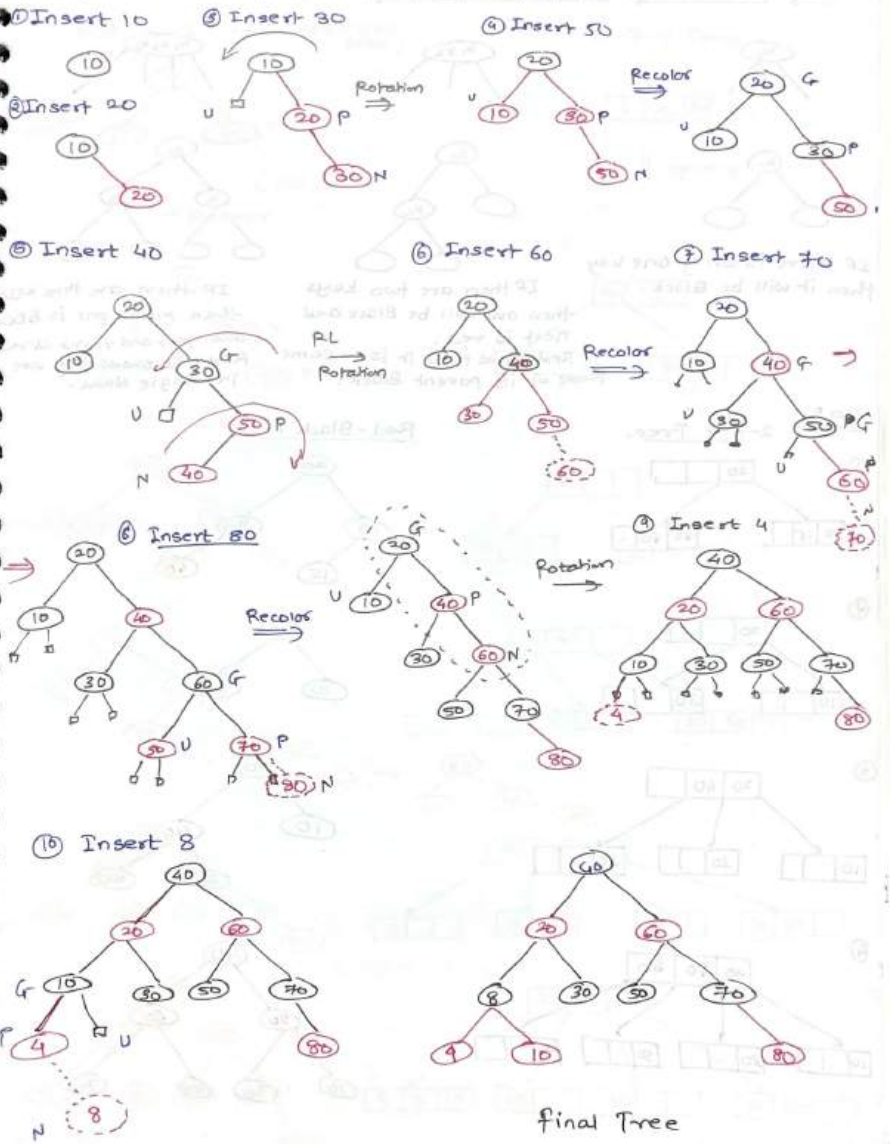


Now if the above node is root, then change its color to Black.

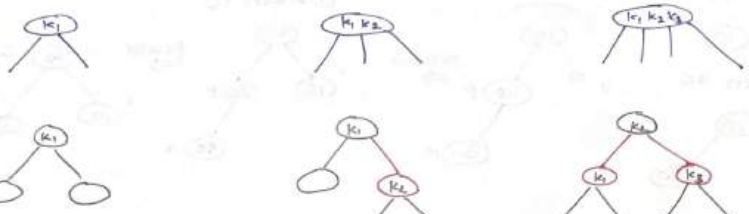
② **Rotation**: If Uncle is Black.



Keys : 10, 20, 30, 50, 40, 60, 70, 80, 4, 8



Red-Black Trees vs 2-3-4 Trees.



If there is only one key then it will be Black.

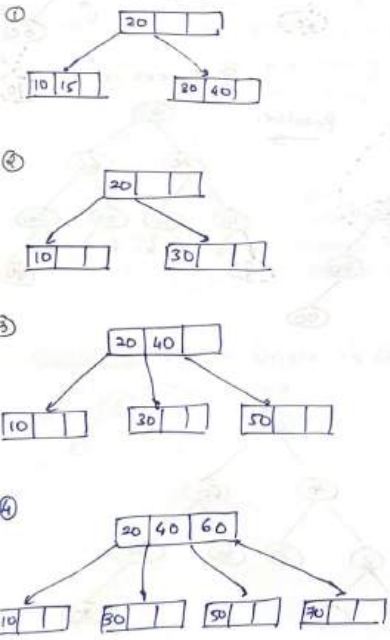
If there are two keys then one will be Black and next is red. Red shows that it is in same Node of its parent Black.

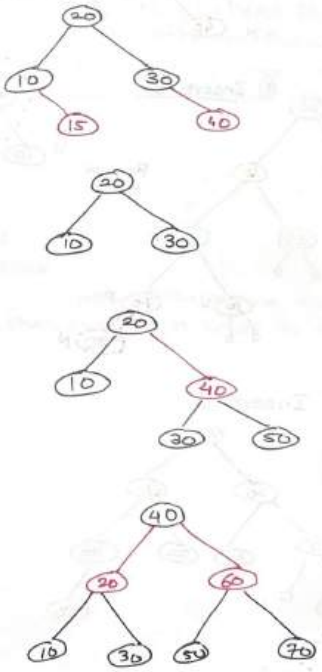
If there are three keys, then Middle one is Black and left and right are Red. It shows they are in single Node.

e.g:

2-3-4 Tree.

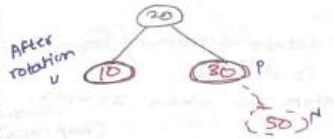
Red-Black Tree



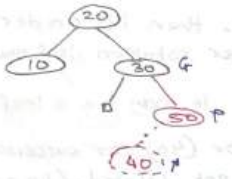


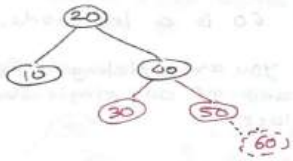
Keys: 10, 20, 30, 50, 40, 60, 70, 80, 4, 8

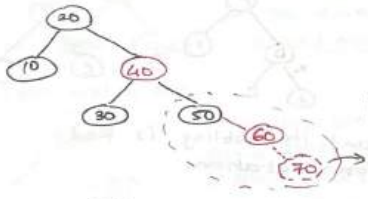
Red-Black Tree (Refer Back Page)

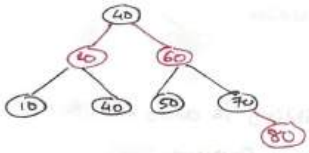


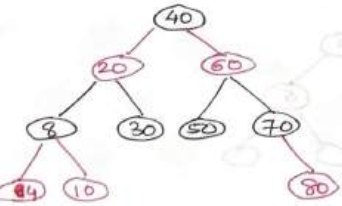
⇓ Recolor



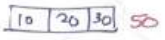




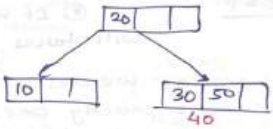


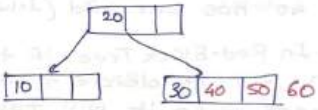


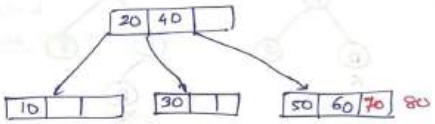
2-3-4 Tree

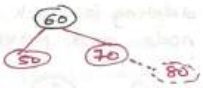


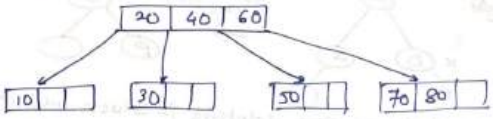
⇓ splitting

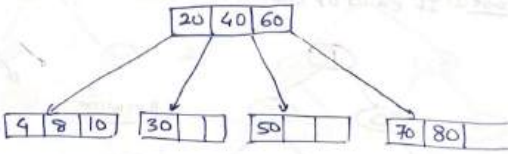






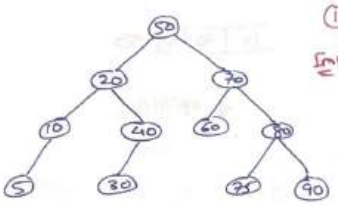






* Red-Black Tree Deletion:

⇒ Deletion is Red-Black Tree is kind of similar to deletion in Binary Search Tree.



BST

① ⇒ Whenever we delete something from B.S.T.
Imp a leaf node is deleted.
e.g: If we delete 30 then 30 will be deleted (leaf node)

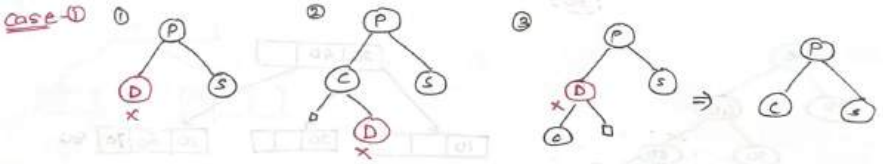
② If we delete 50, then its predecessor 30 will takes its place and 30 is deleted.

③ If we delete 20, then its predecessor will takes place and after rotation, leaf node is deleted.

② Imp ⇒ So whenever we delete node in BST it may be a leaf node or it will have exactly one child.
e.g: If we delete 50 then its predecessor (40) or successor (60) will take its place and they will get deleted. (40 or 60 node)
Here 40 has one child (lchild) and 60 is a leaf node.

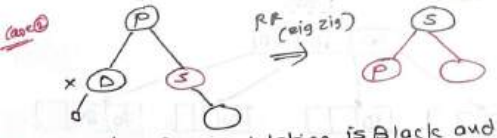
③ Now, In Red-Black Tree if the Node you are deleting is Red, then Blindly delete that node. and If any single child is there then it will take its place.

case ①



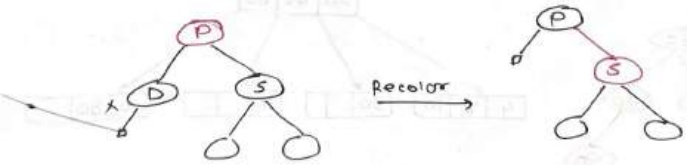
④ Now, If Node we are deleting is Black and its sibbling is Red. then delete Black node and perform rotation.

case ②

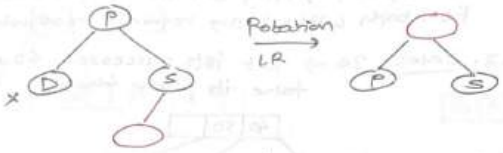


⑤ If Node we are deleting is Black and its sibling is also Black.

case ① If child of sibling is Black then perform Recolor.

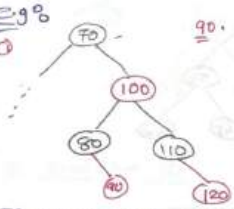


Case ② If child of sibling is Red then perform rotation.



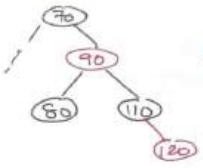
→ Deleting Black node is complex as we have to maintain same number of Black node on any path. i.e. Black Height.

Eg:



If we want to delete 90 they simply delete 90

② Lets delete 100, so either its predecessor 90, can takes its place or Successor 120 Can takes its place.

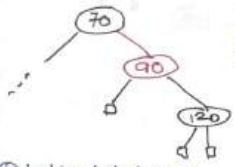


Note:
Though we are deleting 100. But actually 100 gets replaced and 90's node get deleted.

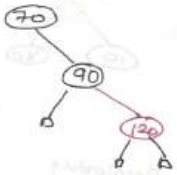
③ Lets delete 110.
No predecessor so sucessor wil take its place

→ 110 got replaced and node of 120 is got deleted
i.e. Red node got deleted

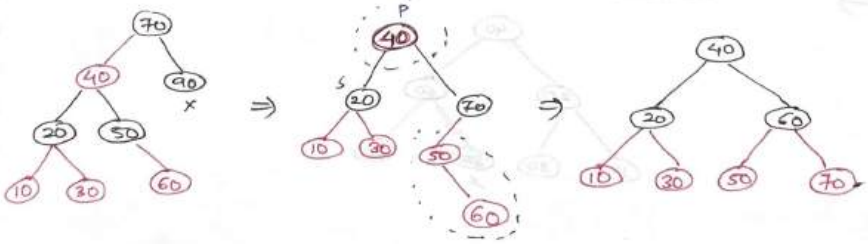
④ Lets delek 80.



As Now we are deleting Black Node 80. As the sibling is Black and its children an also Black. Hence recolor

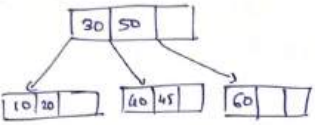


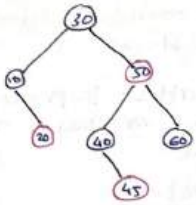
⑤ Let's delete 90



* Comparison of Deletion from Red-Black Tree vs 2-3-4 Tree.

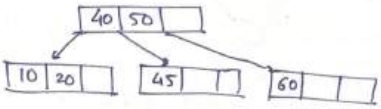
1. Delete 20, 45. → They can be simply deleted from both without any requiring readjustment.

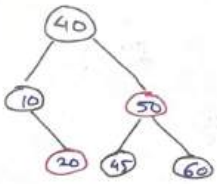




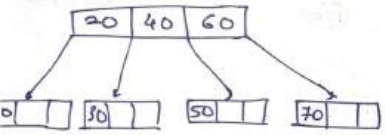
Red color deletion

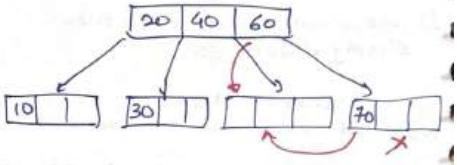
2. Delete 30 → Let its successor 40 will take its place then



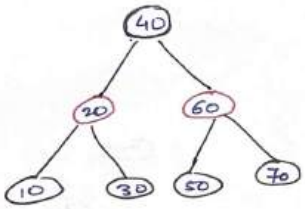


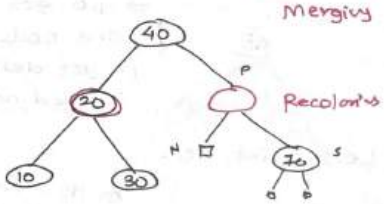
3. Delete 50



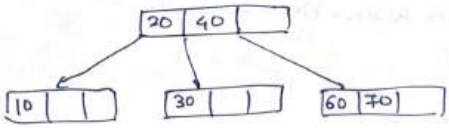


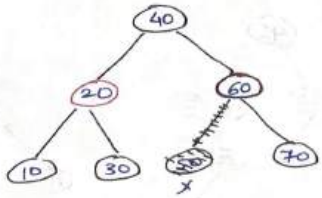
Merging



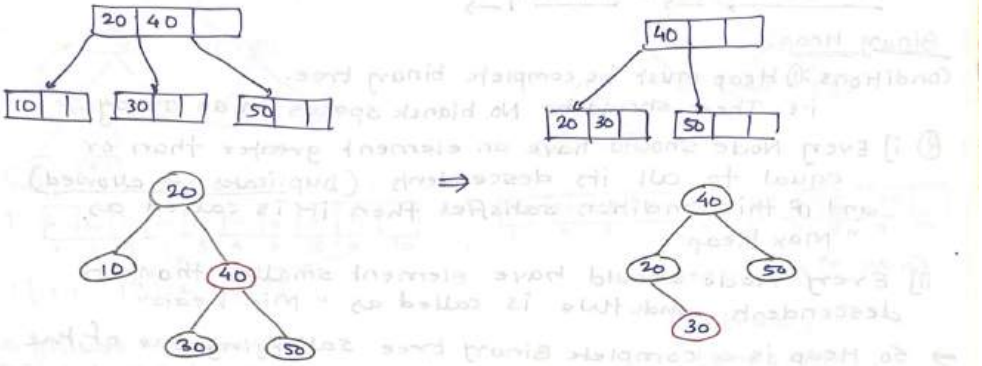


Recoloring

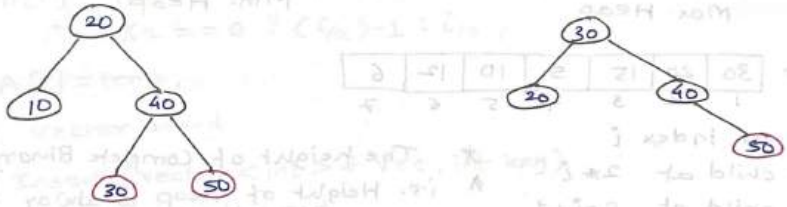




4. Delete 10



5. Delete 10



Section 19 :

Binary Heap:

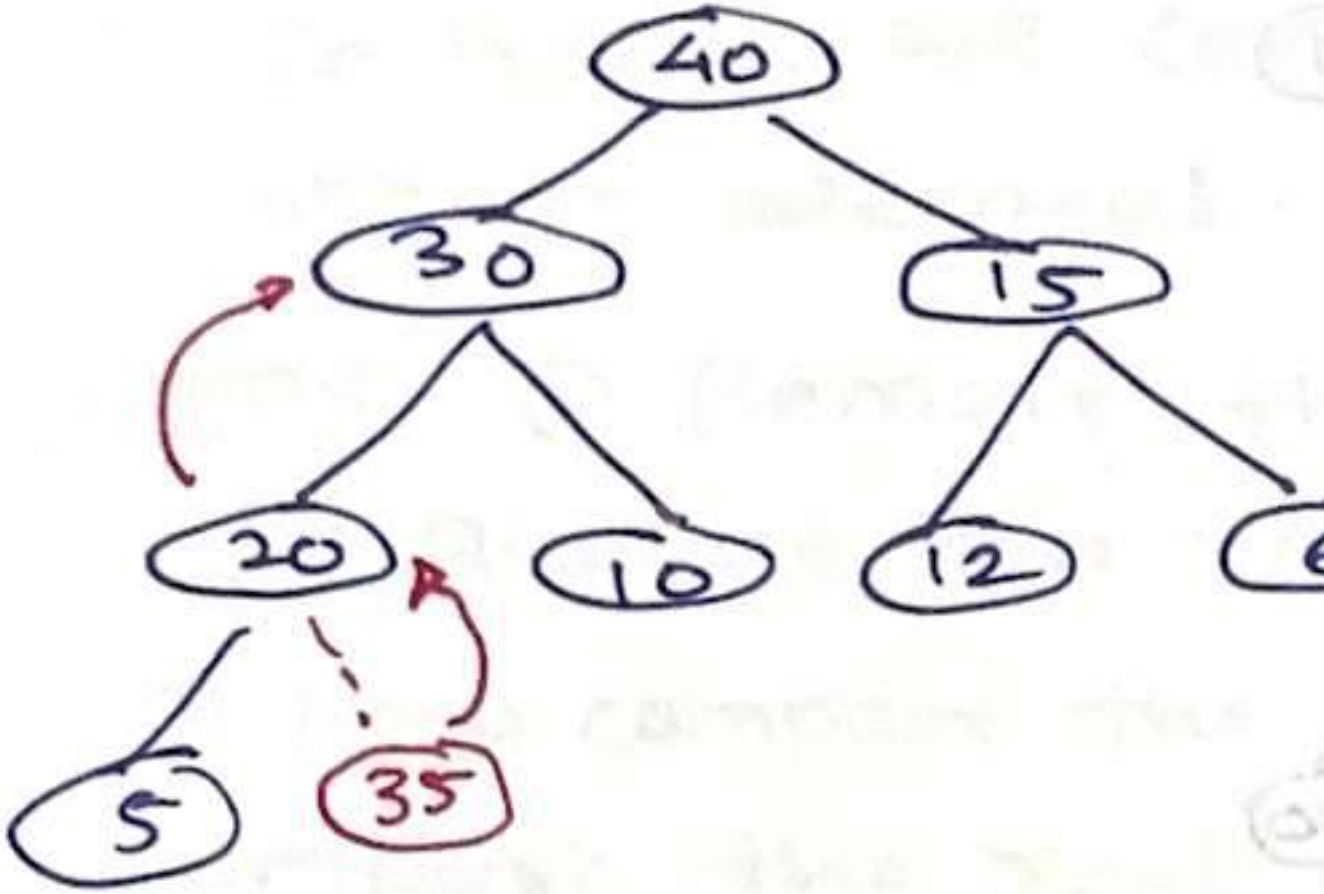
Conditions : ① Heap must b
i.e. There shou

② i] Every Node should
equal to all
and if this conditi
" Max heap.

ii] Every Node should
descendents and H

⇒ So Heap is a compl
above condition for

⑧ Element 35 :

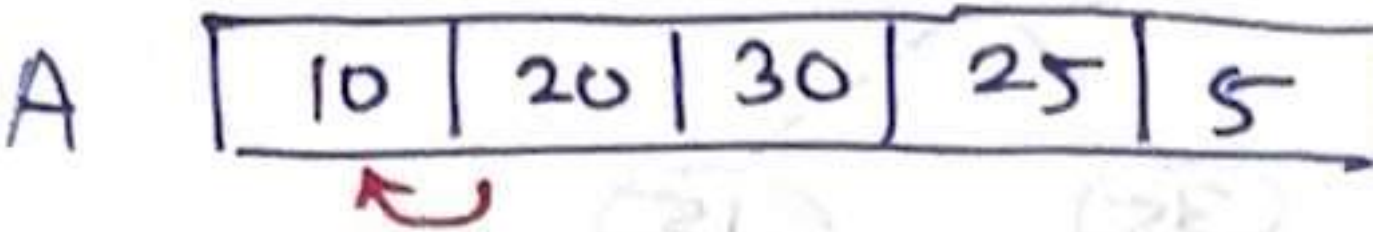


T	40	30	15	20	10	12	6	
	1	2	3	4	5	6	7	

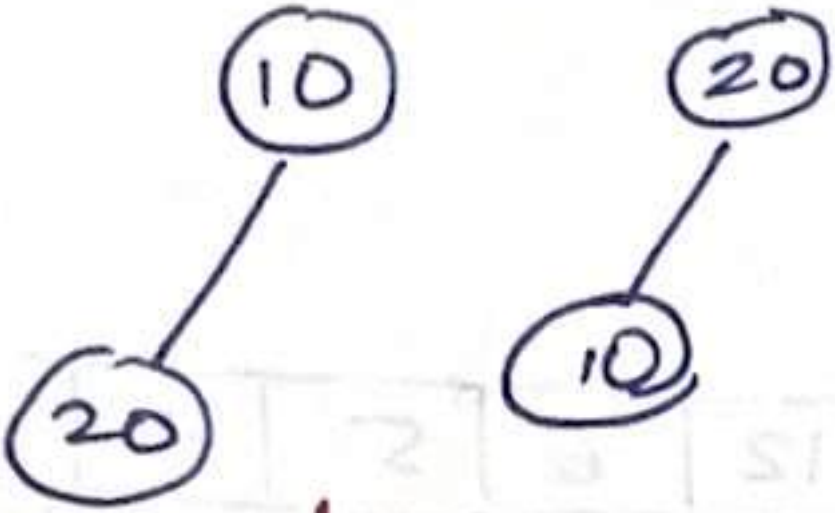
$\lfloor \frac{9}{2} \rfloor = 4$ $\lfloor \frac{4}{2} \rfloor = 2$ $\lfloor \frac{2}{2} \rfloor = 1$

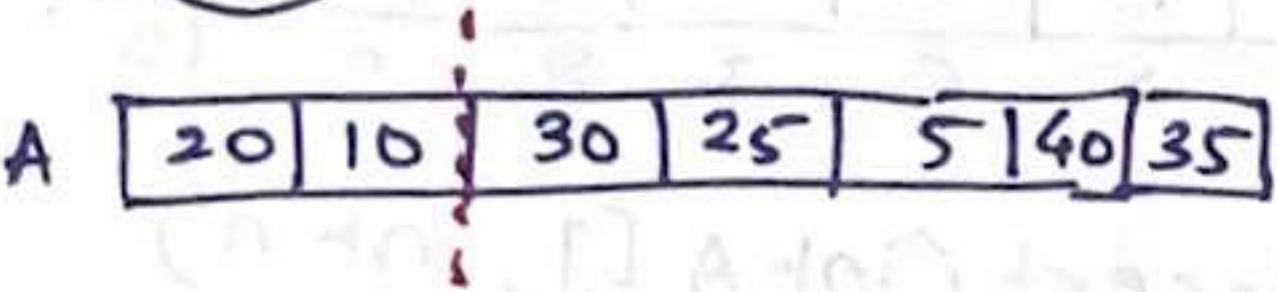
```
#include <iostream>
#include <vector>
using namespace std;
```

* Creat Heap:

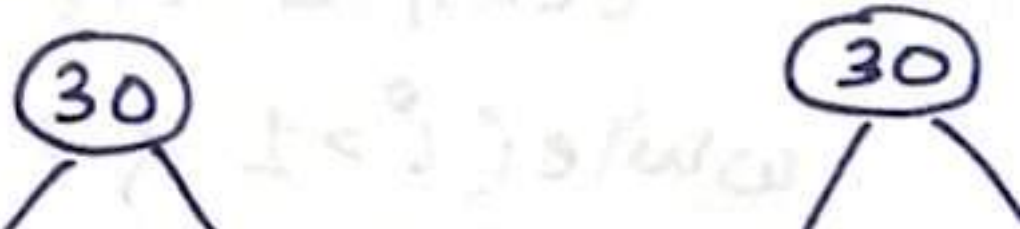


1. Insert 20 →





3. Insert 25 →

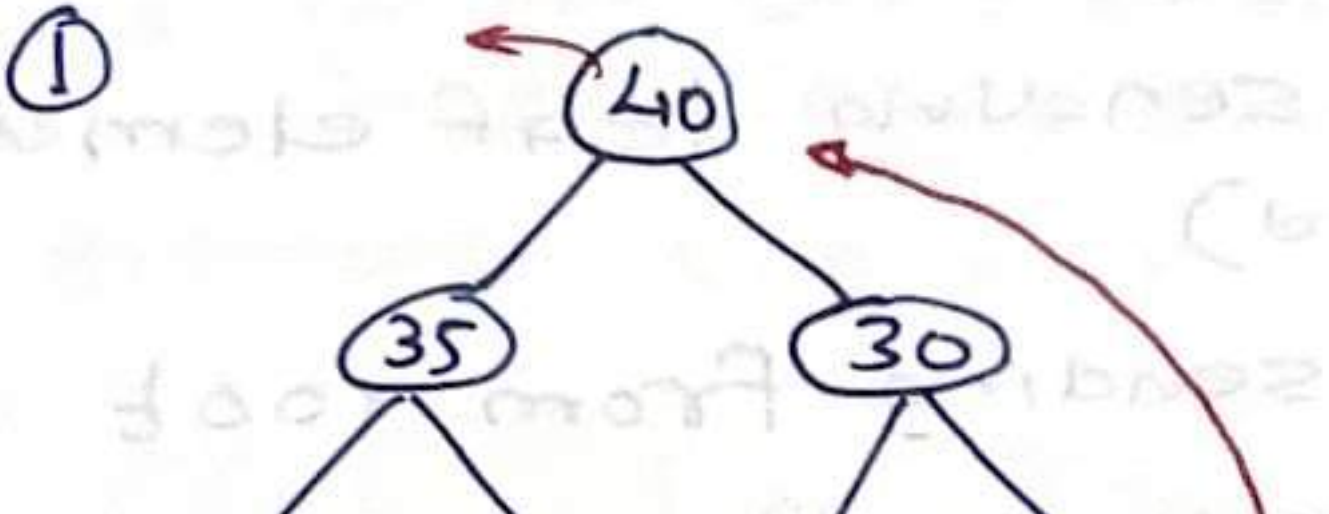


* Deleting Element from

⇒ In heap, we can de
other element.

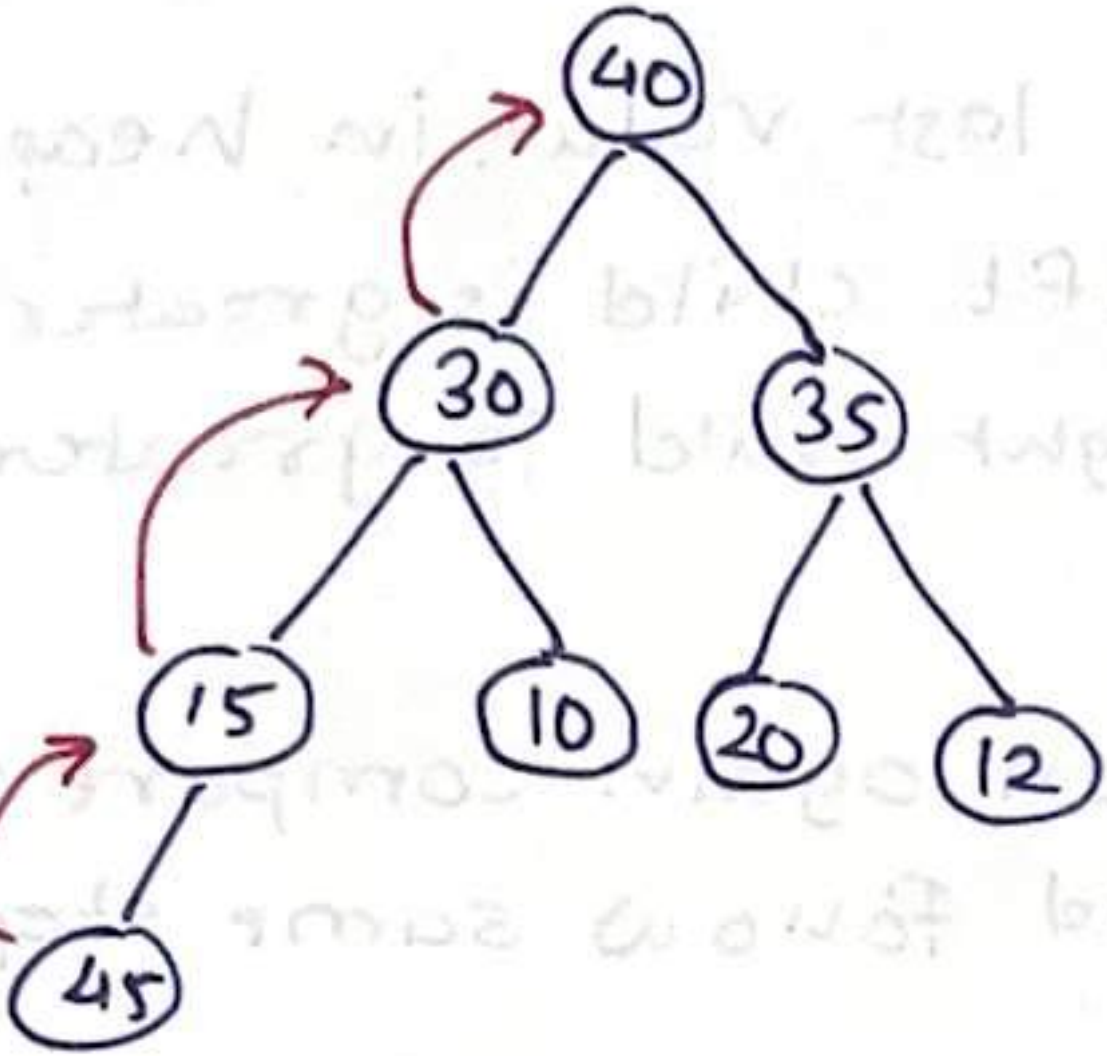
steps: ① Remove the
② Replace the value
③ Now compare the cu
compare the root and
compare root and R
④ If the root is repla
its below descedent

①



* Heapify : faster

Insert



In Insertion, elements towards

* Heap as priority

⇒ In deletion from He

⇒ This property can be
the priority for
that which smalle
priority and depend

* comparison of
using array

→ Larger the elem
higher is the p

Section 20: S

* Sorting Methods:

1. Bubble
2. Insertion
3. Selection
4. Heap sort
5. Merge sort
6. Quick Sort
7. M
8. Sh
9. Co
10. B
11. R

* Criteria for Analysi

1. Number of compar

① * Bubble sort:

A	8	5	7	3	2

1st pass

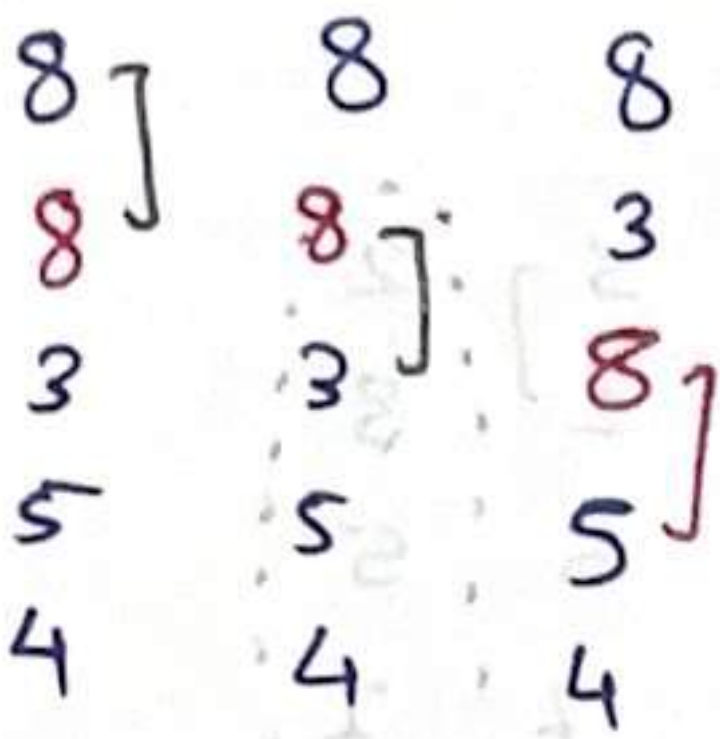
8	5	5	5	5
5	8	7	7	7
7	7	8	3	3
3	3	3	8	2
2	2	2	2	8

4 comparisons

Max 4 swaps

* Analysis →

⑤ Stable:



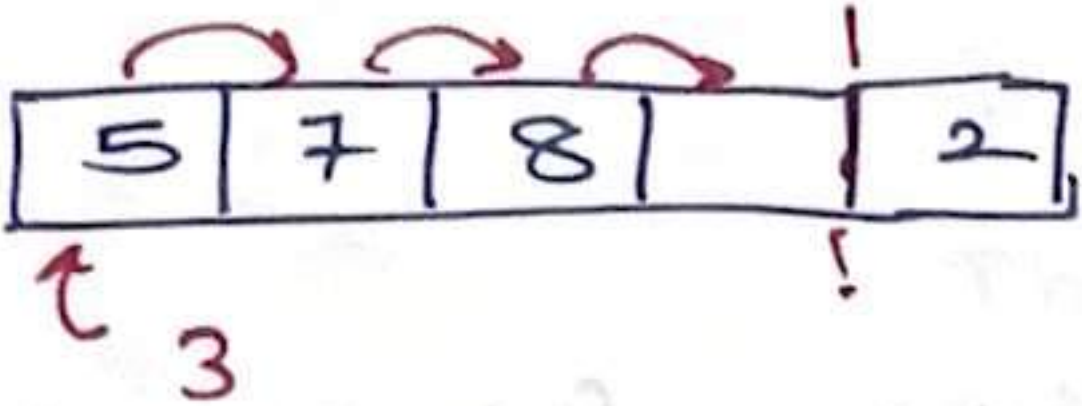
⑥ Insertion Sor

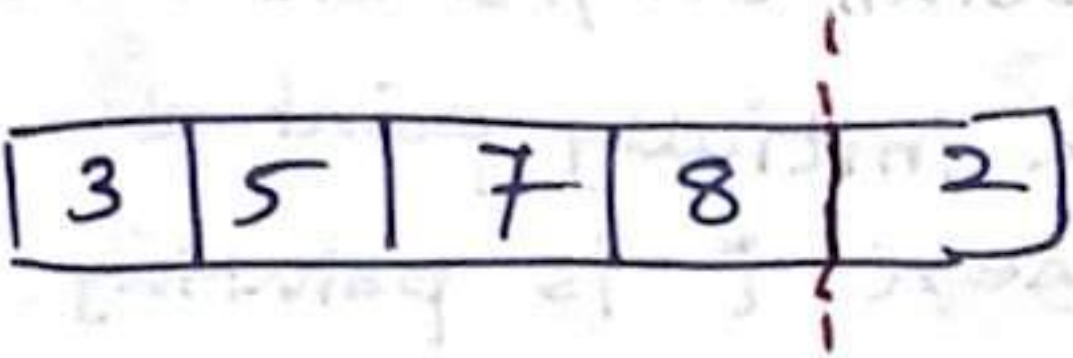
→ what is insertion?

Let insert '12'

3rd pass

Insert 3





3 comp

Max → 3 swap

① No. of p

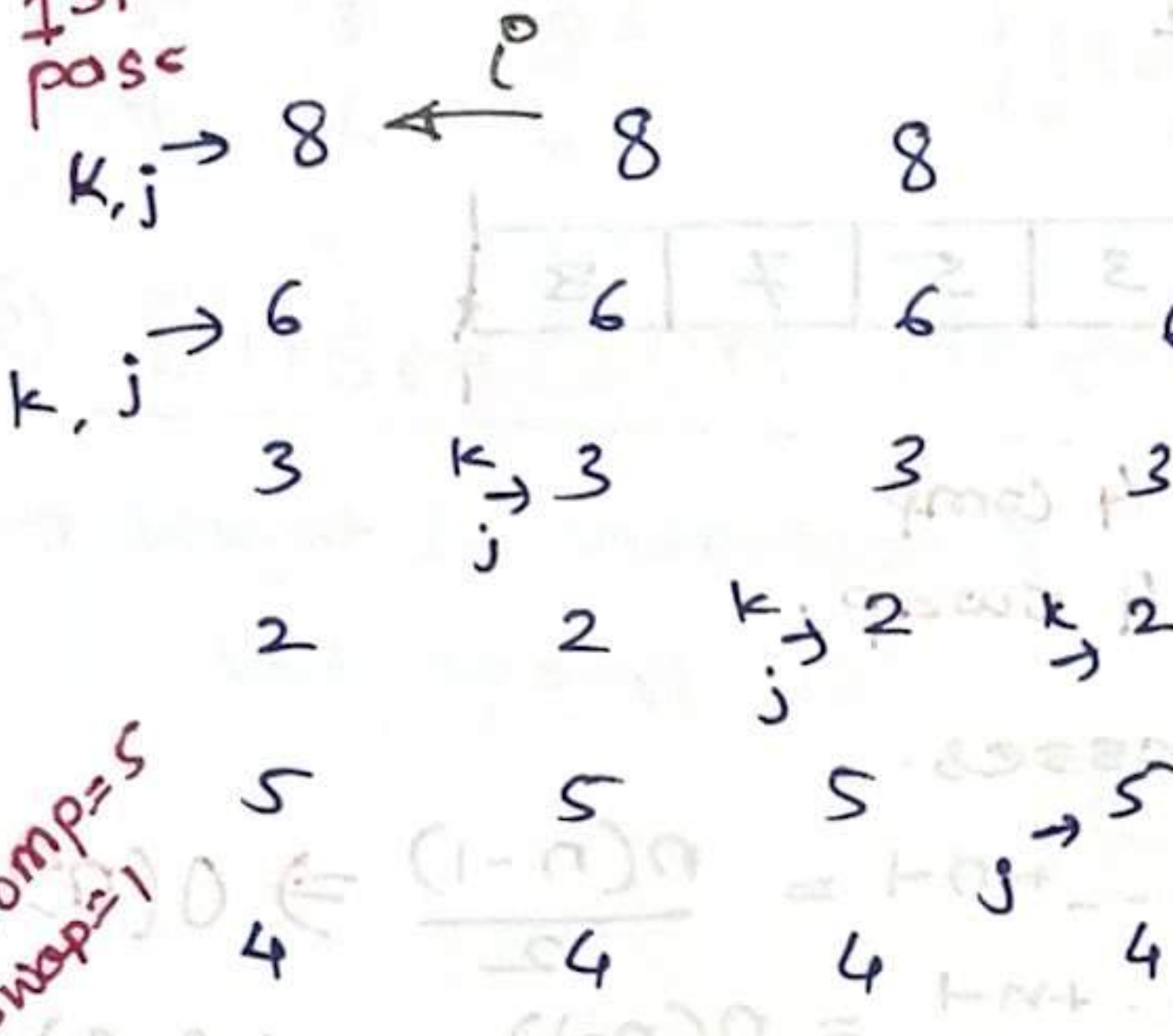
② No. of comparison

③ No. of swaps →

③ Selection Sor

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

1st pass



④ Adaptive:

In selection sort

0 2 ← i
1 4 ← k
2 8
3 10
4 12
5 16 ← j

→ j will

There

this m

Sorte

Hence

⑤ stable:

8 ← i
3

2
3

Here, the

of dupli

ents is

preser

(4) Quick Sort →

(10) 30 20

80 70 40

40 30 20

→ so is sorted a
while on Right

* Quick sort →

Pivot

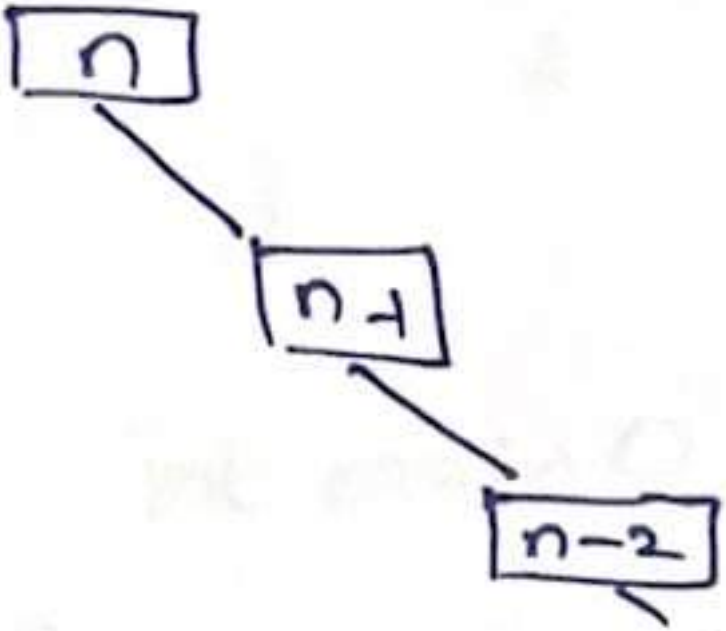
(50) 70 60 90 4
i ←

* Analysis → worst case

Imp ⇒ If the list is alr
order then partitio
position or right m
of list ; and one of

so No. of = n+(n-
Comparisons

→ Time Complexity is



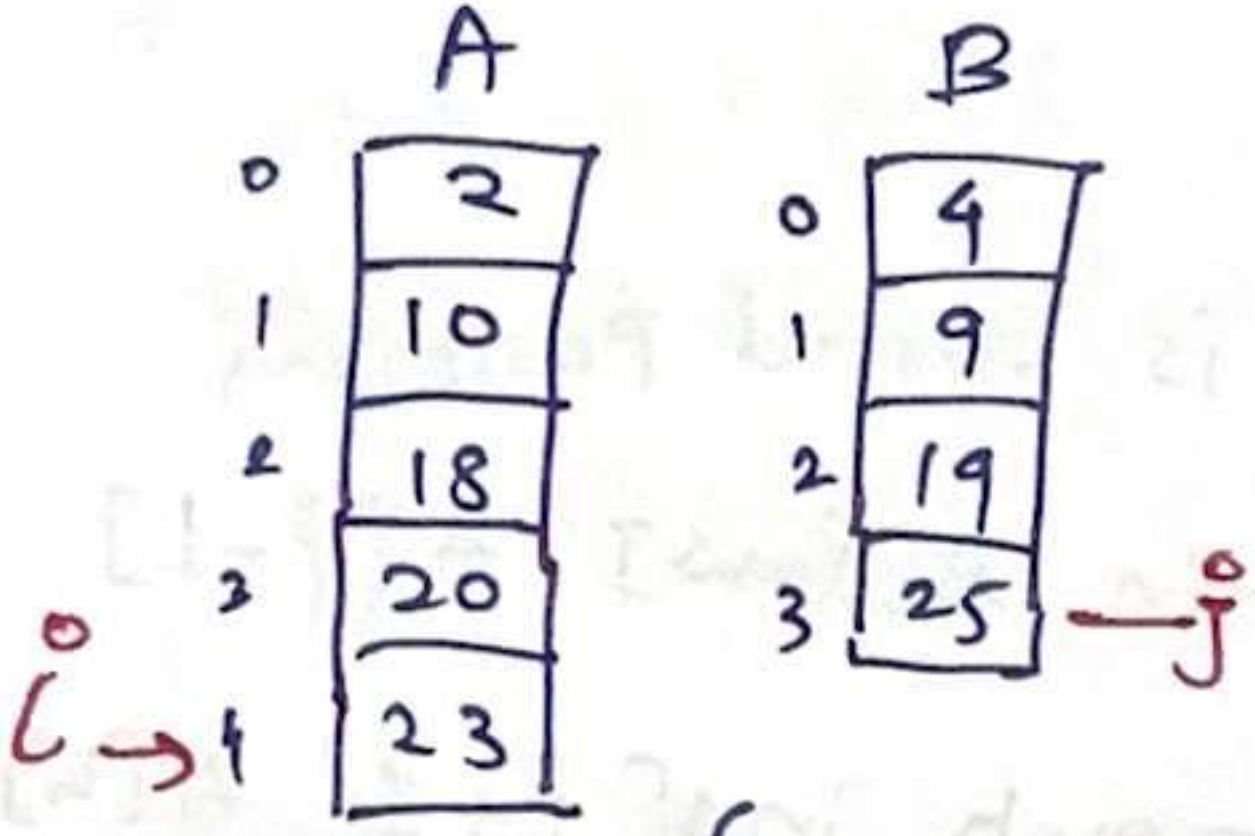
```
# include <iostream>
using namespace std;

void Print (int A[] , int
{
  cout << " A [] : " ;
 for(int i=0 ; i<n ; i++
      cout << " " << A[
}
void swap (int &i , int &
 {  int temp= i;
     i = j;
     j = temp;
```

```
void QuickSort (int
{
    int j;
    if (l < h)
    {
        j = partitio
        QuickSort (
        QuickSort (
    }
}

int main()
```

⑤ Merge S

* Merging :→

① Merging two sep



C

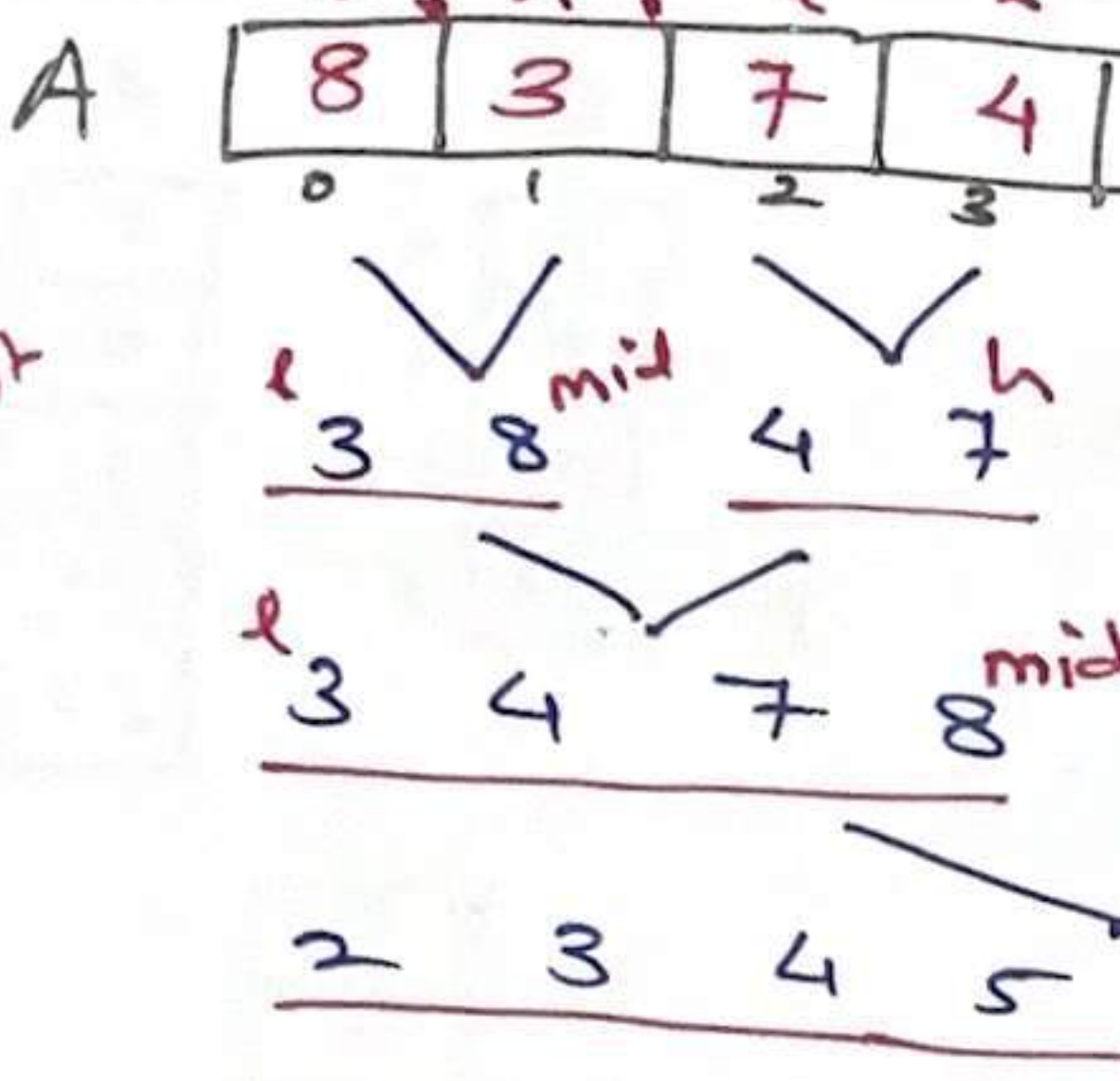
0	2
1	4
2	9

```
void Merge(int A[
{
    int i, j, k;
    int B[h+1];
    i=l ; j=mid+1 ;
    while (i<=mid &
    {
        if(A[i] <
        {
            B[K] =
            i++;
        }
        if else {
            B[K] =
            j++
        }
    k++
}
```

Imp

* Iterative Merg

→ In this sorting, we
then merge one k



h is
l & at
index posit
i=0
i=1

Void Iter.MergeSort(

* Recursive Merg

→ Here also we follo

→ when the array i
Now, there is no
sorted. Hence ag
Contain only sing
will always be s

Imp

Imp

⟹ Now start. Merging

```
voidRMerge Sort(
{
    if (l < h)
    {
1.  mid = (l+h)/2
```

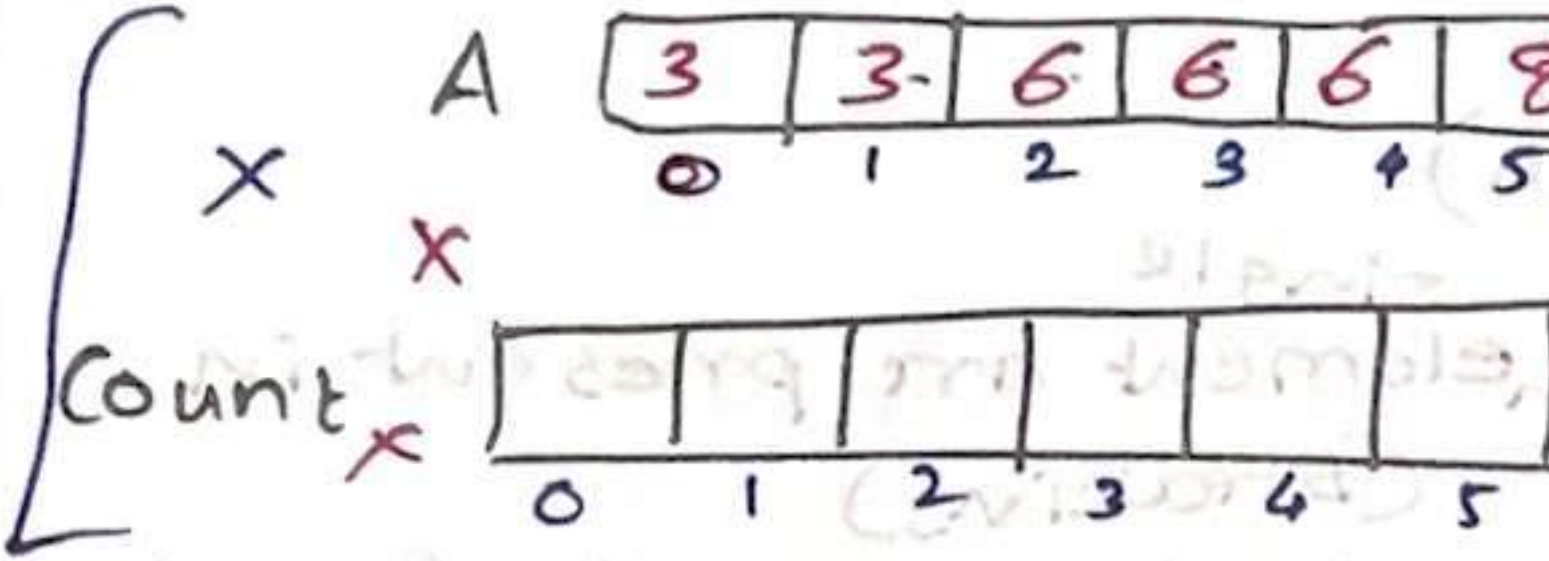
⑥ Count Sort →

→ It is the easiest sort as we have to creat value element in th

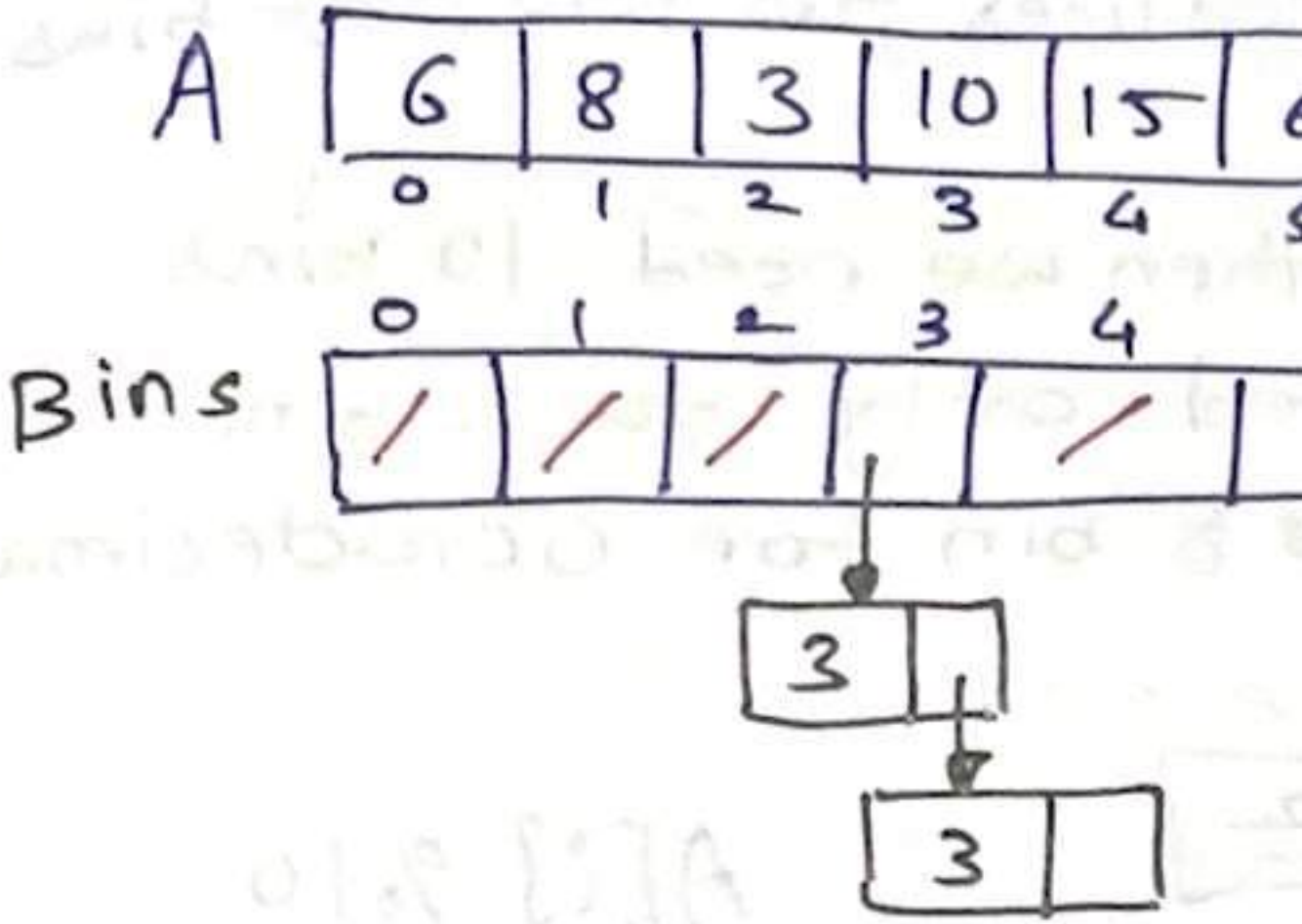
Steps: 1. Creat one coun

2. Increment value at index

3. Refill array (list) bas



(7) Bin / Bucket



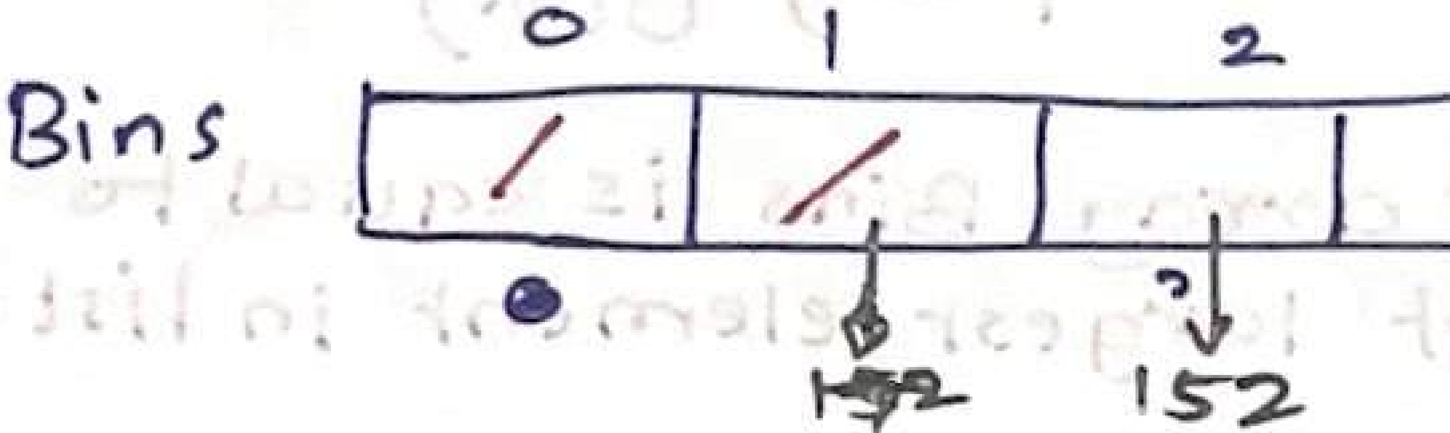

```
void BinSort(int A[], int
{
  int max = Max(A, n);
  // Create bins array
```

(8) Radix Sort

→ This method is simila[r]
required we use F
→ If the number are
→ It the number are B
similarly 6 bins Fo

1st pass

A	237	146	259	348	152
	0	1	2	3	4

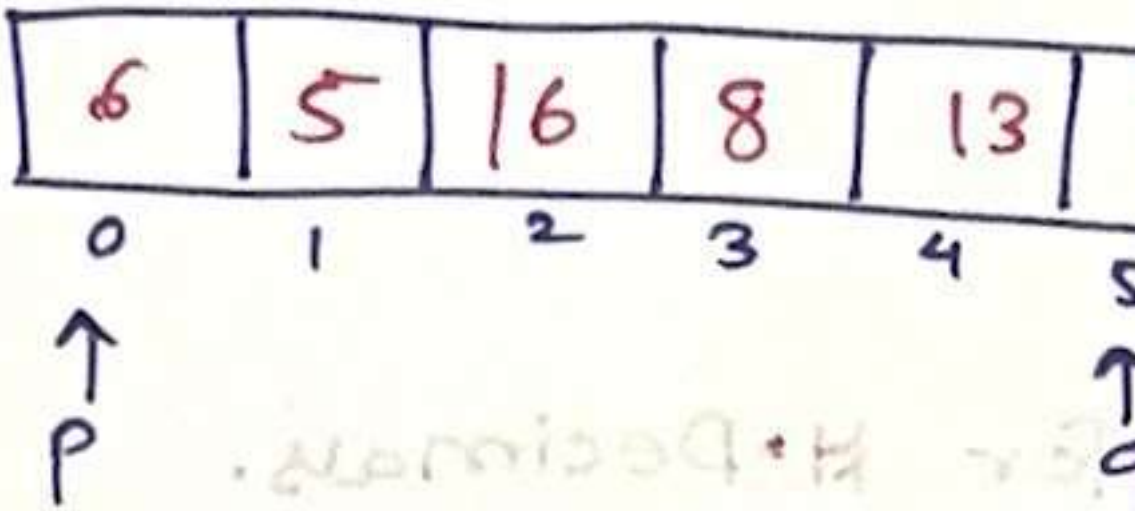


```
Void RadixSort(int A[],
{
    int max = Max (A, n);
    int npass = CountDigits (n
// Create bin array
    Node ** bins = new Nod
// Initialize bins array with n
    InitializeBins(bins, 10);
// Update bins and A for
for(int pass=0; pass<npa
{
    //update bins based o
    for(int i=0; i<n; i+
    {
        int binIdx = getBinI
```

⑨ Shell Sort

1st Pass

A



→ We assume first ele

from first ~~the~~ take

→ Compare this two ele

then keep them as

Now 6 < 9 H

→ Now shift to next

→ Now shift to next

Code

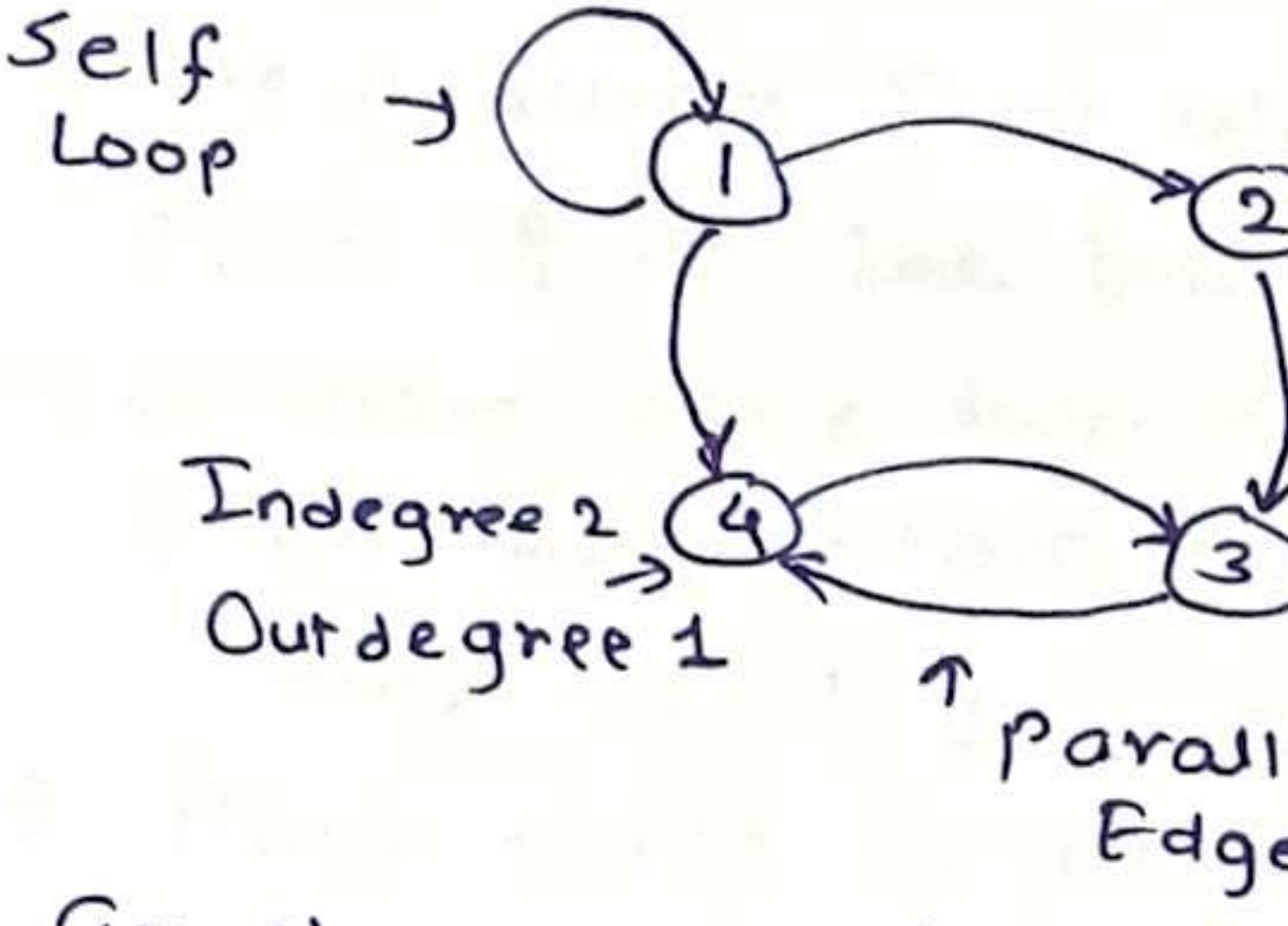
```
#include <iostream>
using namespace std;
template < class T >
void Print ( T & vec, in
{
    cout << s << ":
    for ( int i=0 ; i< n
    {
        cout << vec [
        if ( i< n-1)
            cout <<
    }
    cout << "]" <<
}
```

Section 22 : G

⇒ Graph is defined as c

e.g:

① Directed Graph



⇒ Graph means Undirec

* Representation of Un

⇒ There are more than

① Adjacency Matrix

② Adjacency List

③ Compact List.

① Adjacency Matrix →

→ The graph is represen

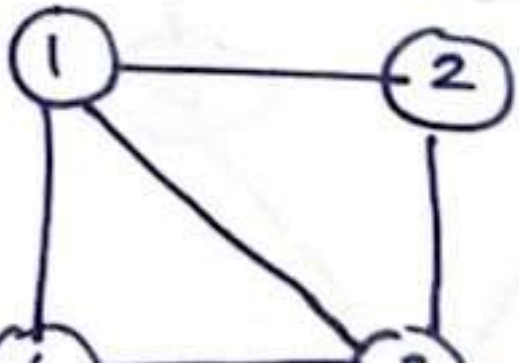
set of vertices an

⇒ The No. of vertices

⇒ The No. of edges

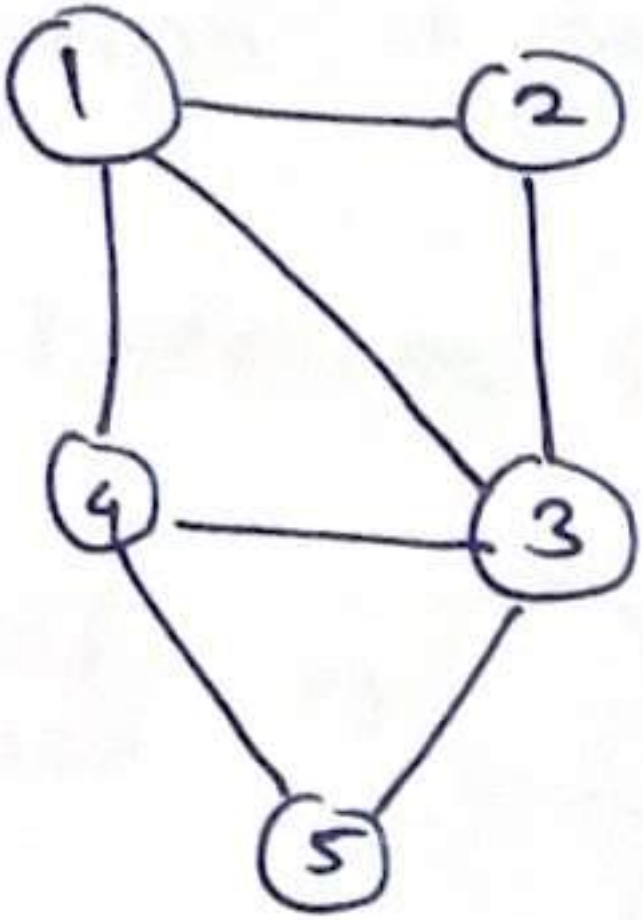
⇒ If there is edge betwe

mark it as zero.



⇒ $A = \begin{matrix} & 1 & 2 \\ 1 & \\ 2 & \\ 3 & \end{matrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$

③ Compact list:

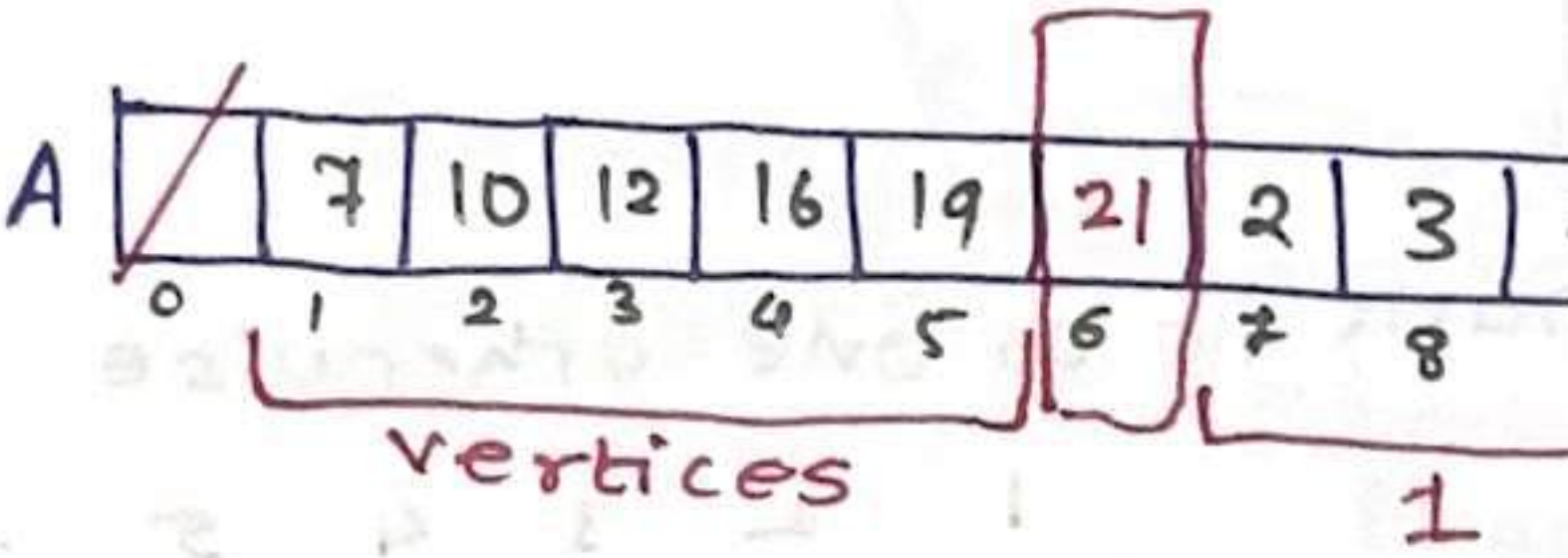


G = (V

|V| = n

|E| = e

Array si



* Breadth First

⇒ Graph can be easil

level order Metho

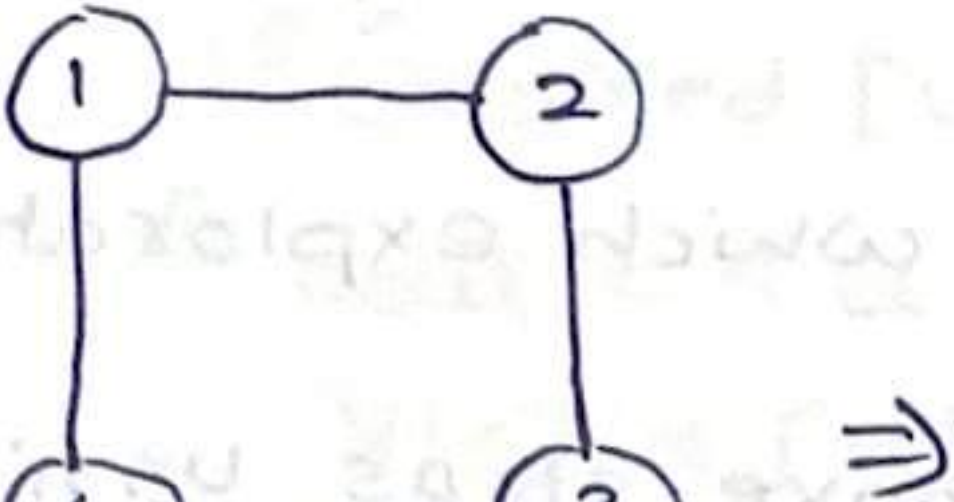
⇒ Depending upon t
can be traced.

Visiting : visiting vert

Exploring : visiting a

⇒ we can visit in any

e.g :



* Depth First Se

→ Graph can be conver
done like Pre-order
called as "Depth

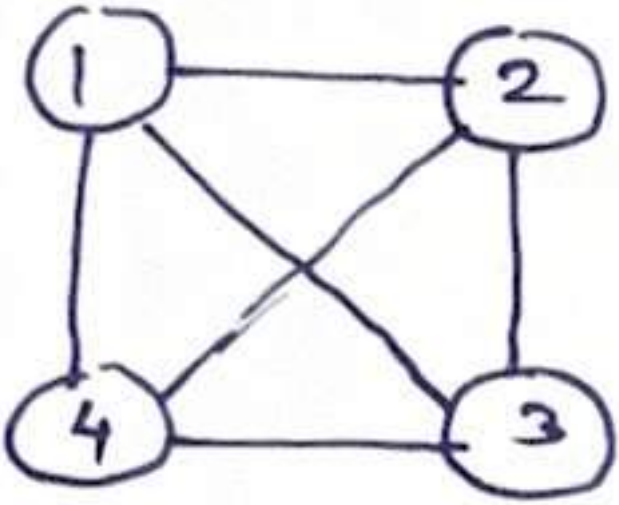
Procedure:

1. Visit first vertex
2. Explore vertex (
when you go on
and push previous
for next vertex (2
3. Follow this untill
is done.

```
void DFS (int A[][8],
{
    int u = vtx;
    int visited [8] {0};
    Stack <int > stk;
    stk. emplace(u);

    while (! stk. empty()
    {
        u = stk. top();
        stk. pop();
        if (visited [u] != 1
        {
            cout << u <<"
            visited [u] = 1;
            for (int v=0 ; v
```

* Spanning Tree

⇒ Spanning Tree is a
of a graph. but



G = (

n = |V| =

e = |E| =

```
#include <iostream>
#define V 8
#define I 32767.
using namespace std;
void PrintMST(int T[]
{
  cout << " Minimum sp
  int sum[0];
  for(int i[0] ; i < V-2
  {
    int c = G[T[0][i]]
    cout << "[" << T[0]
    sum+=c;
  }
  cout<<endl;
  cout << " Total cost of
}
```

② Kruskal's Mini

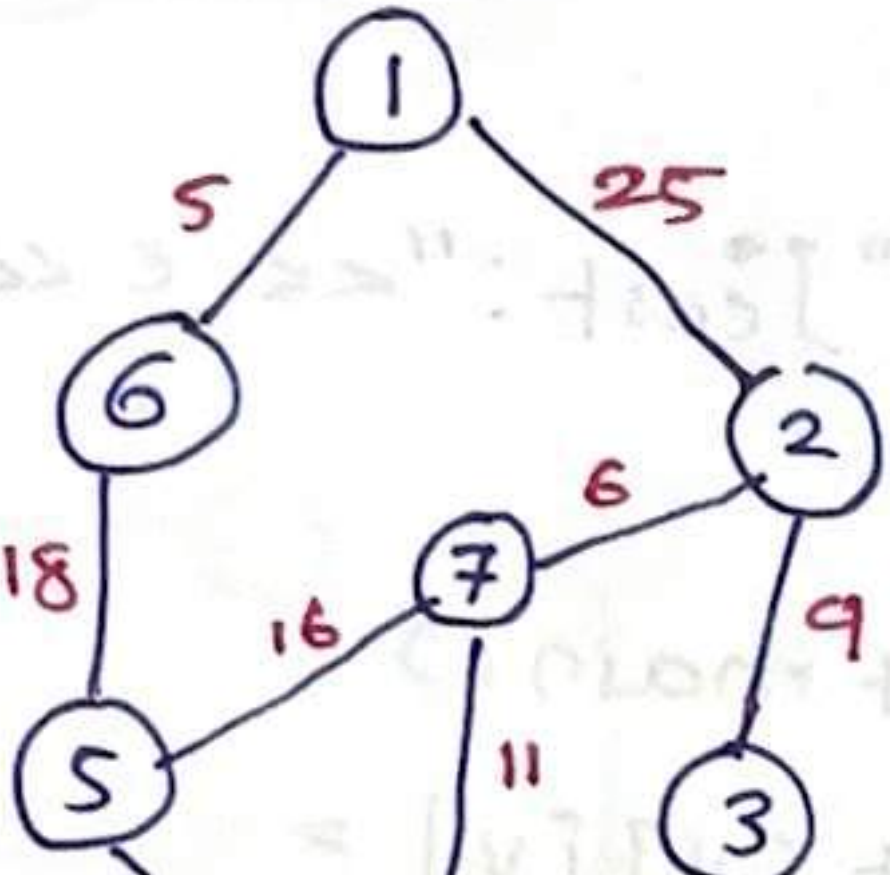
→ In this there are n

step: Kruskal's say th

but make sure

If they are forming

other next minimu

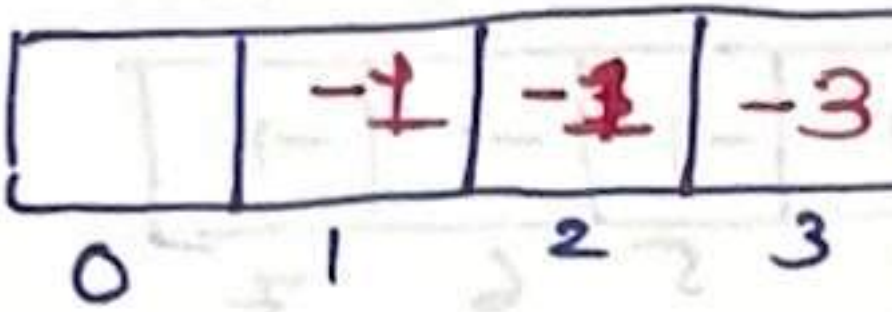


* Disjoint Subset

⇒ Disjoint Subset m
there is no com

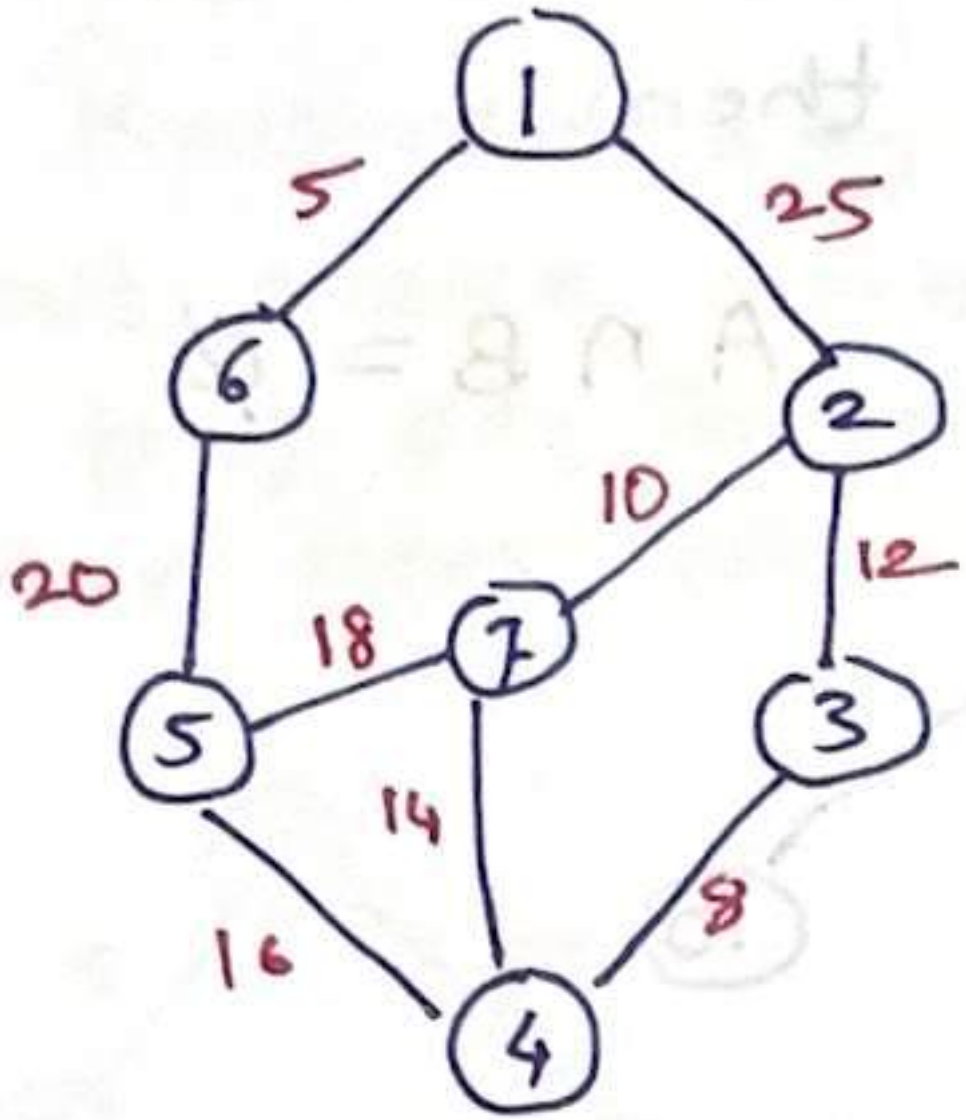
E.g: $A = \{3, 5, 9\}$

Digramatic Representation



① At index 5, it is 3

* Program



```
//Set Operation

  Union
void Union(int u, int v, in
{ if(s[u] < s[v])
    {
      set[u]=set[u]+se
       set[v]=u;
    }
  else
    {
       set[v] = set[u]
        set[u]=v;
    }
}
void KruskalsMCST(
{
    int T[2][v-1];
```

Section 21: Ha

→ Hashing is useful fo

→ Already there are M

① Linear Tim

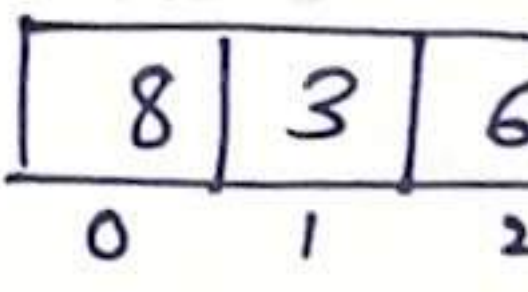
② Binary Tim

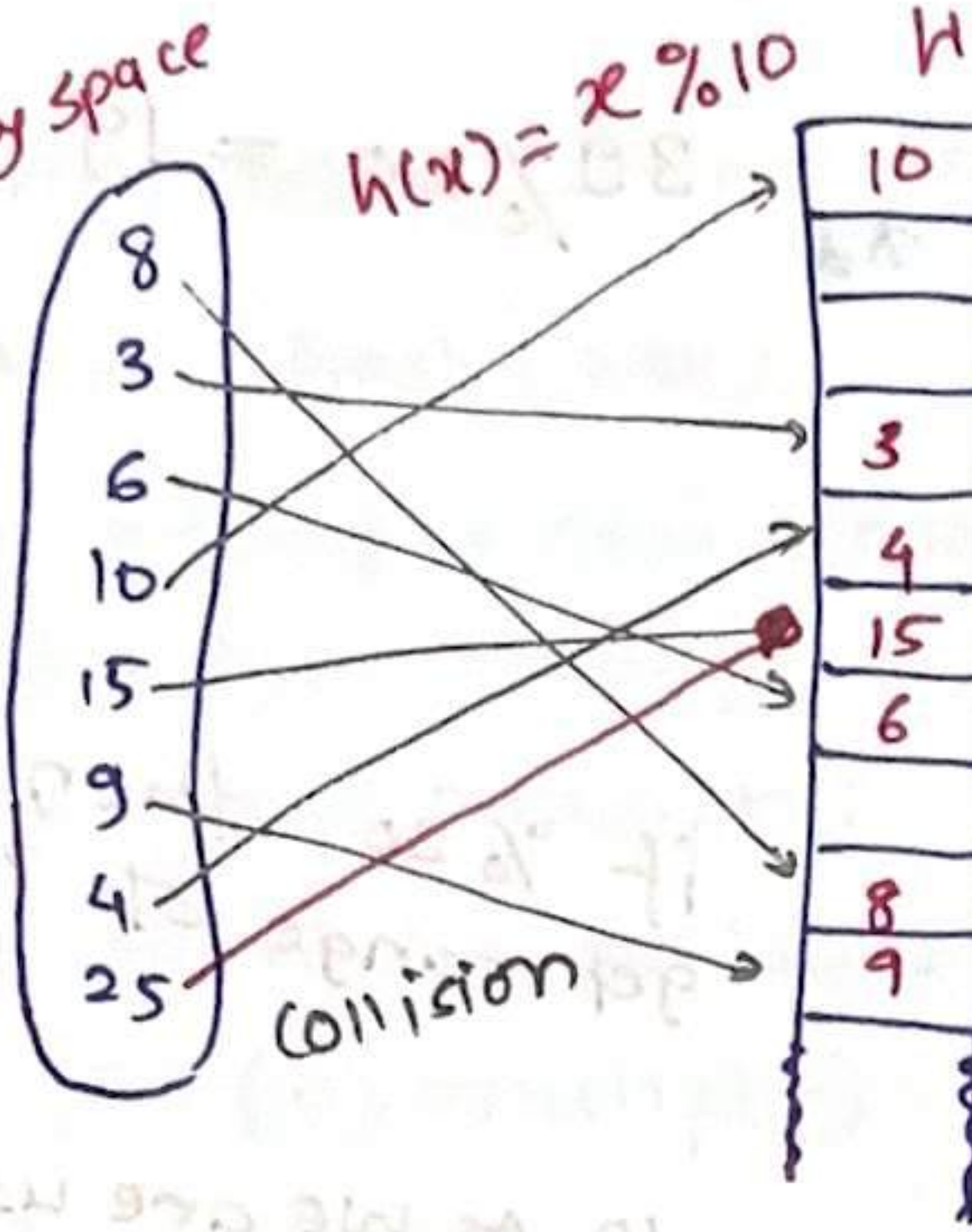
→ We want more faster
niques are used.

→ for Binary search

e.g: keys: 8, 3, 6, 10, 1

Linear →





⇒ The main Difference

* In Closed Hashing, w
we will not increa
* In open Hashing, w

Code

```
#include <iostream>
using namespace std;

class Node
{
  public:
    int data;
    Node * next;
  };

class HashTable
{
public:
  Node * * HT;
```

```
void HashTable :: Insert
{
    int a = hash(key);
    Node * temp = new N
    temp -> data = key;
    temp -> next = nullptr;
//Case : No nodes in Link
    if (HT[a] = nullptr)
    {
        HT[a] = temp;
    }
    else
    {
        Node * p = HT[a]
```

② Closed Hashing

① Linear Probing →

→ As this is closed H
Hash Table only.

→ The simple idea,
then try to resolve
the next position i

→ If at next position
happening at next

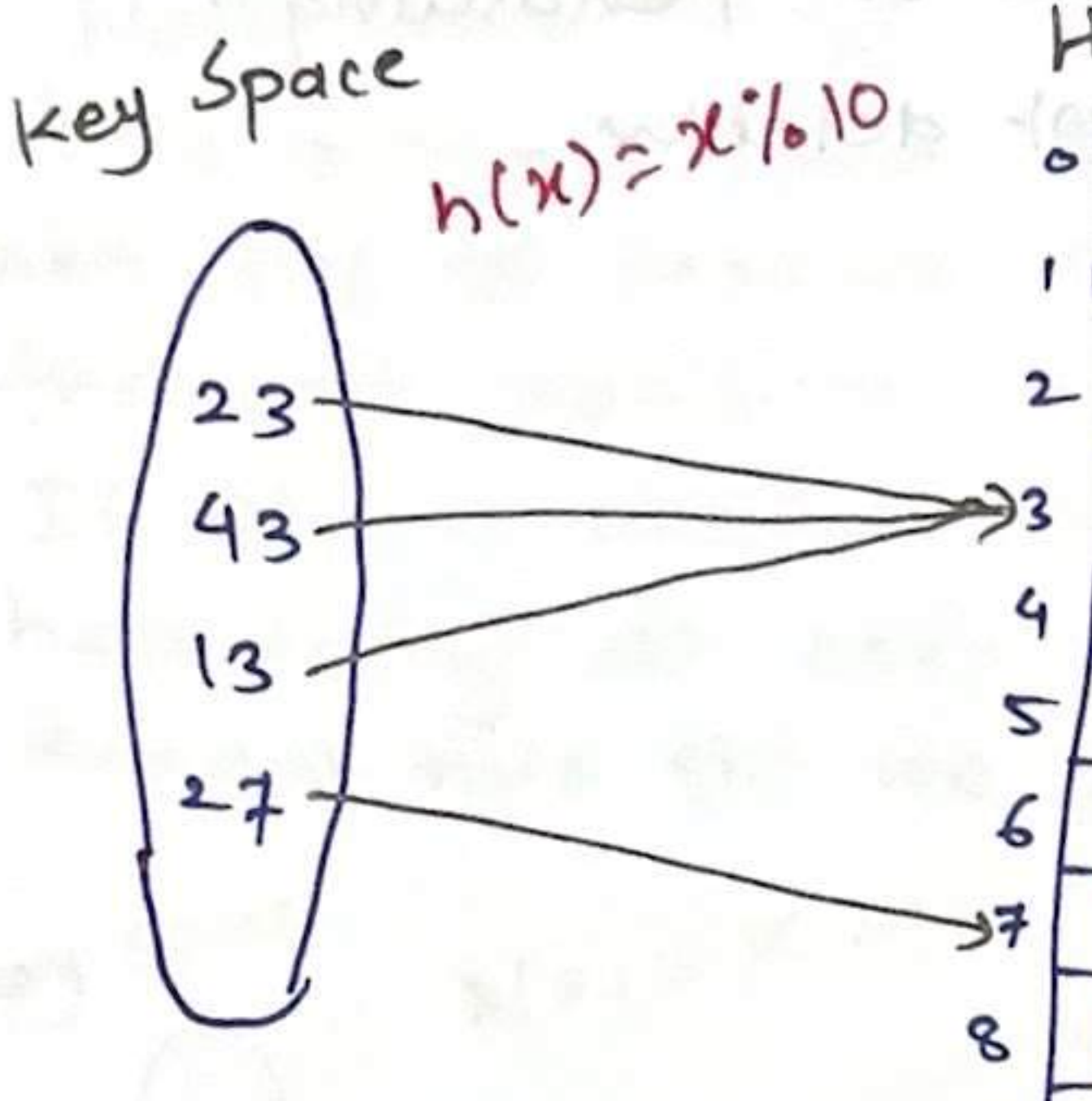
→ Repeat this till we

As. if we simply dele
won't be able to sear
→ so if we really want
element from Hash Ta
alot of work. and thi
→ Hence, in Linear prob

```
#define size 10;
int Hash(int key)
{
    return key % size
}

int LinearProbe(int H
{
    int a = Hash(key
    int i = 0;
```

(2) Quadratic Probin

It is similar to Linea
function f(i). Here



code

```
#define s
#define p
int Hash (int key)
{
    return by % size.
}

int DoubleHash (int H
{
    int a = Hash (key);
    int i = 0;
    while ( H[CHash (
        {
            i++;
        }
    return (a + i * P
}
```

* Ideas for Hash

VIMP

→ Hash function shou
Values in Hash Tab

→ for closing (close
exceed 0.5. He
double the numbe

→ Hash Table size
based on the V

→ It is suggested
taken as prime nu