* SAFE ARRAY :
→ Copy Constructor in terms of Deep copy.
→ Checking the bound of array
→ Calling a destructor
• Properties of safe Array.

"RULE OF THREE"

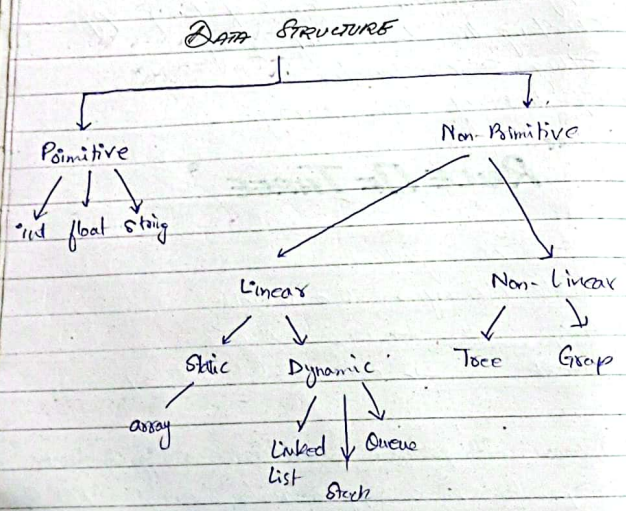
⇒ Dynamic Memory Allocation
① Static Memory → static variable
② Heap " → Dynamic Allocation
③ Stack. → other normal variables & functions.

① "Class with pointers should have a Copy Constructor"
Agr pointers ho or copy constructor user defined nhi bna oga to shallow copy hojayega.

② "Class with destructor needs copy constructor & assignment operator"
Agr humne khud se copy constructor bnaya hai to memory allocate ki hogi to us pointer ko destroy krne ke liye destructor zaroori hai.

Infinix NOTE
CamScanner

③ "Destructors & Multiple objects pointing to same address"



"TYPES OF SORTING"

- Elementary Sorting (for small data)
- Advance Sorting (for large data)

a[i] | i | j | temp | a[j]

| 5 | 4 | 2,1 | 5 < | 8 |

ELEMENTARY SORTING

(INSERTION SORT)

→ Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration.

→ 1st element ko hum sorted consider karenge. Or hum pehle jo baaki array bachega usko unsorted. Ab hum har unsorted k 1st element ko sorted array part k elements se compare karenge or usko place karenge.

8	2	6	10	5
0	1	2	3	4

sorted | unsorted

2	8	6	10	5

sorted | unsorted

sorted k is done elements se compare karenge.

2	6	8	10	5

sorted | unsorted

Is ko 8 se compare karenge agr 10 > 8 to small wali to usse compare nhi karenge a/k sorted array bar we to loop se bahar aayegi use.

```
for (int i=1; i<length; i++){
    temp = a[i];
    for (int j=i-1; j>=0 && a[j]>temp; j--){
        a[j+1] = a[j];
    }
    a[j+1] = temp;
}
```

jab tak temp variable k wrong appropriate place nhi nhi jaata hum 1 place right shift krte rhenge.

8	4	2	3	2

8	9	12	2	6	4

Adaptive & non-Adaptive:

→ Those sorting algorithms which doesn't waste their time on sorted element is called adaptive else non-adaptive.

• Important Points:

→ Insertion sort only sorts the array when it's necessary if array is in order no substantial moves are performed.

→ It is Adaptive & Stable.

• Time Complexity:

⇒ Worst Case → $O(n^2)$

⇒ Average Case → $O(n^2)$

⇒ Best Case → $O(N)$ as it will not enter in the inner loop.

Stable & Not Stable

→ If a sorting algorithm after sorting the contents does not change the sequence of similar content in which they appear, it is called stable sorting.

→ Yani agar ek array mein ~~2 bain~~ do 26 hai or ek ki index value 3 or dusre ki 6 hai to sort hone k baad agar 3 wala 6 se pehle hi rhe to stable agar 6th wala 3rd se aage aajye to unstable. Relative order same rhenge change.

2	26	3	26

→

2	3	26	26

✓ →

2	3	26	26

X

agar koi element kisi ek by 1 compare krein to isko matlab window length -1 jaise bubble sort mein hota h

"SELECTION SORT"

→ The element with the lowest value in whole array is selected & exchanged with the element in 1st position.

→ Then the smallest value among the remaining elements $a[1] \ldots a[n-1]$ is found and put in 2nd position.

10	4	10	11	1

i 0

j 1 2 3 4

```
for (int i=0 ; i< length ; i++) {
    min = i
    for (int j=i+1 ; j< length ; j++) {
        if (a[j] < a[min])
            min = j;
    }
    if (min != i)
        swap(a[min], a[i]);
}
```

→ Non-Adaptive & Not Stable

• Time Complexity

⇒ Worst Case, Best Case, Avg Case → $O(n^2)$.

"BUBBLE SORT"

```
for (int i=0; i<n-1 ; i++){
    swapped = false
    for (int j=0; j<(n-1-i) ; j++){
        if (a[j] < a[j+1]){
            swap(a[j], a[j+1]);
            swapped = true;
        }
    }
    if (swapped == False) → agar swap ki value false
        break; bhai hai to wo aage ke pass
                mein jayega hi nhi.
}
```

→ NON adaptive but stable

- Time Complexity:

⇒ Worst Case, Avg → $O(n^2)$

→ Best Case → $O(n)$ → agar le bhai is wala kaam sirf ek baar chalega or andar wala n times

"COMB SORT"

→ length of array is ÷ by 1.3.

→ By dividing let initial gap = 10

→ After shrinking gap value i.e 10/3 = 7

0 1 2 3 4 5 6 7 8 9

[8] 4 1 56 3 -44 23 [-6] 28 0

(0 + 7 = 7) → Now we will swap (if necessary) the 0th element with 7th

-6 [4] 1 56 3 -44 23 8 [28] 0

(1 + 7 = 8) → not swap as 4 < 28

(2 + 7 = 9) → so applying swap

(3 + 7 = 10) → greater than array length so now new gap value i.e 7/1.3 = 5

[-6] 4 0 56 3 [-44] 23 8 28 1

(0 + 5 = 5) → swapping seq.

-44 [4] 0 56 3 -6 [23] 8 28 1

(1 + 5 = 6) → not swap

(2 + 5 = 7) → not seq

(3 + 5 = 8) → seq

-44 4 0 28 3 -6 23 8 56 1

(4 + 5 = 9)

-44 4 0 28 1 -6 23 8 56 3

```
for (int i=0; i<length; i++)
    gap = newgap(gap);
    for (j=0; j+gap<length; j++)
        if (a[j] > a[j+gap])
            swap
```

```
int getNextGap(int gap)
{   gap = (gap * 10)/13
    if (gap < 1);
        return 1;
    else
        return gap;
}

int gap = n
bool swapped = true;
while (gap != 1 || swapped == true){
        gap = getNextgap(gap);
        swapped = False;
        for (int i=0; i<n-gap ; i++){
            if (a[i] > a[i+gap]){
                swap(a[i], a[i+gap]);
                swapped = true;
            }
        }
}
```

"SHELL ART"

→ Extended version of shell sort.
→ Has improved average time complexity
→ Efficent for Medium sized data.
→ We will round down the gap values.

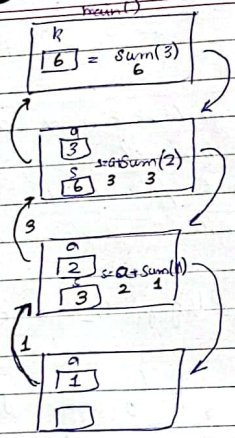
```
for (gap = n/2 ; gap ≥ 1 ; gap = gap/2){
        for (j = gap ; j<n ; j++){
            for (i = j-gap ; i>=0 ; i-gap){
                if (a[i+gap] > a[i])
                    break;
                else
                    swap(a[i+gap], a[i]);
            }
        }
}
```

ye loop jab second time chalega to backward element se comparision check krega.

→ It's Adaptive but Not Stable
→ Time Complexity:
Best → $O(n\log(n))$
Worst, Avg → $O(n(\log(n))^2)$

RECURSSION

```
int main(){
    int R;
    R = sum(3);
    printf("%d", R);
}
int sum( int a){
    int s;
    if(a==1)
        return a;

    s = a+ sum(a-1)
    return (s);
}
```




```
if(n==0||n==1)
    return n;
int ans = f(n-1)+f(n-2);
```

Backtracking uses recursion to solve a problem
↳ jb base condition use hoti hai jab use optimal solution mil jata hai.

QUICK SORT

→ Also called Partioned Exchange Sort.
→ We will choose a pivot element from our array.
↳ pivot element can be any element from array
→ either 1st, last, middle or any random
→ Now will will partion the array
→ Such that the values less than pivot are on left side & values greater than pivot are on R.H.S of pivot.

key element

Partion 1	pivot	Partion 2
values < pivot		val > pivot

→ equal values in any of the side
→ Now applying same logic on partion 1 & 2. We will continue this until only one element is left in subarray.
→ There will be 2 variables. "Start" & "End".
Start ↓ pointing to 1st; End ↓ pointing to last
→ We will increment the start till we reach an element greater than equal pivot.
→ We will decrement the end till we reach less than equal to pivot.

pivot = 7 → a[0]

7	6	10	5	9	2	1	15	7

start, start: to par stop because greater than pivot; end: end ↑ par stop because 7 is equal to pivot.

→ Now swapping values of start & end!

7	6	7	5	9	2	1	15	10

start start end end

7	6	7	5	1	2	9	15	10

start end start end

→ Now we will not swap as start has crossed end.
→ we will follow swapping until start < end
→ Now we will swap End with pivot element.

pivot

2	6	7	5	1	7	9	15	10

less than 7 — greater than 7

now same logic on this part — and this part

Quick Sort (A , lb , ub)

```
{ ( lb < ub ) {
    int loc;
    loc = Partion ( A, lb, ub )
    Quick Sort( A, lb, loc - 1 )
    Quick Sort ( A, loc+1, ub )
}
}
```

this will return end position

lower band — upper band — loc - 1

2	6	7	5	1
0	1	2	3	4

9	15	10
6	7	8

loc + 1 ie 5+1 = 6 — upper band

ऐसा सिर्फ चलता रहेगा

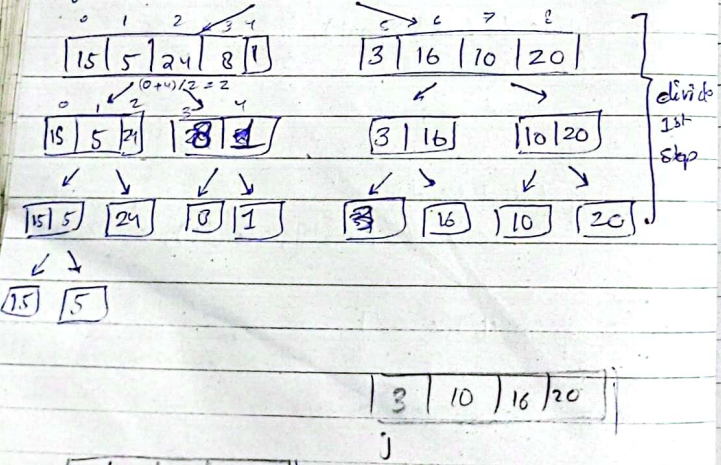
```
int Partion ( A , lb , ub ) {

    pivot = a [lb] ;        // first element pivot
    start = lb ;
    end = ub ;

    while (start < end)     //checking whether start crossed
                            //end or not
    {
        while (a[start] <= pivot)
        {
            start ++ ;
        }
        while (a[end] > pivot)
        {
            end -- ;
        }
        if (start < end)
            swap(a[start], a[end]);
    }
    swap(a[lb], a[end]);    // ye job dalegi jab
                            cross hojayegen end & start
                            then sap pivot with end.
    return end;
}
```

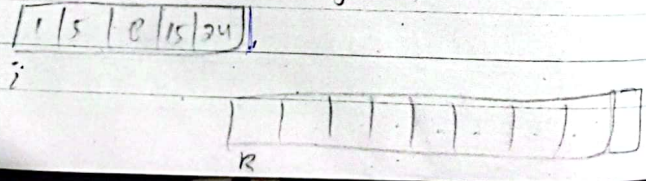
"MERGE SORT"

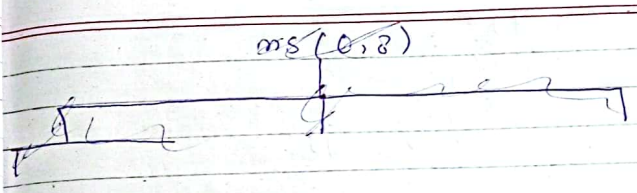
→ The complete sublist is divided into n sublist
→ ~~Each sublist~~ we will keep on dividing the sublist until the sublist has one element only.
→ After that we are going to merge the adjacent sublist.
→ we will keeping on merging sublist to produce a sorted ~~sub~~ list.

0	1	2	3	4	5	6	7	8
15	5	24	8	1	3	16	10	20

→ we will divide this into two sublist. For that we need to find middle. $[(0+8)/2 = 4]$








```
MergeSort(A, lb, ub){
    if ( lb < ub) {
        mid = (lb+ub)/2
        MergeSort(A, lb, mid);
        MergeSort(A, mid+1, ub);
        Merge (A, lb, mid, ub);
    }
}

Merge(A, lb, mid, ub) {                 size = ub-lb+1
                                        arr b[size];
    i = lb;  j = mid+1  ;  k = lb;
    while (i<=mid && j<= ub) {
        if (a[i] <= a[j]) {   agar i wala data kam ; so to ise merge
            b[k] = a[i];      store hoyega or i increment.
            i++;
        } else {
            b[k] = a[j];      agar j wala kam to use data jayega.
            j++;
        }
        k++;
    }
}
```

→

```
if(i > mid) { // agr i k element khatam hogaya
    while ( j <= ub) { j mein kuch bacaa h. to unhe direct place kardenge.
        b[k] = a[j];
        j++;
        k++;
    }
else
{ // vaise agr j k khatam hogai or i k
  while (i <= mid) { bache hui to unhe direct place kardenge. q k unke phele se sorted hai!
        b[k] = a[i];
        j++;
        k++;
    }
}

for (k=lb ; k<=ub ; k++) // copying elements from b to a
    a[k] = b[k];

delete[] b ;
}
```

→ Binary Search & Interpolation Search is done on sorted data. It is used to reduced time complexity.

→ To find Mid = $Low + \frac{Hi - Low}{2}$

→ If arr[mid] > key then our arr will break down and we will now search on L.H.S.

0	1	2	3	4	5	6
10	20	30	40	50	60	70

Mid = 3

key = 20

arr[mid] > 20

40 > 20

"Counting Sort"

→ Sorting acc to keys

→ Key is the maximum value in the array. ['k' value means largest value in array]

→ Counting the elements having distinct key values.

→ Hum has ek element k count krenge or usko ek array mein store krdenge.

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

idea

count → | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
0 1 2 3 4 5 6 7 8

→ size of count array = k+1

→ a[0]=2 to ab hum count[2] par 1 plus kadenge.

→ a[1]=1 " " " " [1] " " " "

→ a[2]=1 " " [1] " " " "

→ a[3]=0 " " [0] " " " "

⋮

→ a[16]=9 " " count[9] " " "

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

→ Ab hamara leken sorted array is type ka hoga

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

→ Ab hum apne count array ko modify krenge.

→ Hamare sorted array mein jo kar key value ki max position hai wo count mein store krenge jese 0 ki 3rd, 1 ki 6th, 2 ki 10th ----.

→ To uske liye count array mein pichle wali value ko aage waali value se + kadenge.

0	1	2	3	4	5	6	7	8	9
count → 3	6	10	10	11	12	17	14	15	17

→ sabki max position store hogai.

3+3=6 6+4 10+1

ye pichle wale count ki values thi.

→ Ab hum in count ki values k mutabiq apne sorted array mein values place krenge.

→ Hum apne original array ki end se traverse start krenge (to maintain stability) or has value par jese agr '9' hai to ab hum count ko 9th index par jayenge or waha jo value pari hai usko decrement krenge yaani 17 ko 16 kar ab 16th index par 9 store kraa denge.

↳ ye index kisi doosre array ka hoga. baad mein usko original k andar copy kadenge.

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

```
CountSort(a, n, k) {

    int count[k+1];
    int b[n];
                                    // storing count of each key
    for (int i=0; i<n; i++) {       // value in Count[]
        ++ Count[a[i]];  or  val = a[i]
                             ++ Count[val]
    }

    for (i=1; i<=k; i++) {          // modifying count value to
        Count[i] = count[i] + count[i-1]   positions
    }

    for (i=n-1; i>=0; i--) {        // sorting the values &
                                       storing in b[].
        b[-- Count[a[i]]] = a[i]
    }

    for (i=0; i<n; i++) {
        a[i] = b[i];        // copying elements
    }

}
```

Drawbacks:

① If no. of elements in array is 100 and 'k' value is 70,000 then there will be many elements in count array having value 0. So 'k' value must be $O(n)$ like 100, 200, 600 but not n^2.

② Not work on negative values. [seq. modification]

③ There are different values in any range.
652, 0, 999, 1111, 10000, 7, 432

• TIME COMPLEXITY: $O(n+k)$

"RADIX SORT"

```
void RadixSort(a, n) {
    int max = getMax(a, n);
    for (pos = 1; max/pos > 0; pos *= 10) {
        countSort(a, n, pos);
    }                  // calling count sort for each position.
void CountSort(int a[], int n, int pos) {
    int count[10] = {0}
    int b[10];
```

→

```
for (int i=0 ; i<n ; i++){
        ++count[(a[i]/pos)%10];
}

for (int i=1 ; i<=10 ; i++){
        count[i] = count[i]+ count[i-1];
}
for (int i=n-1 ; i>=0 ; i--){
        b[--count[(a[i]/pos)%10]] = a[i];

for (int i=0; i<n ; i++){
        a[i] = b[i];
}

}
```

Time Complexity: $O(d*(n+k))$

↓ no. of digits

Formula for interpolation: $pos = L + \left(\frac{H-L}{a[H]-a[L]} * x - a[L]\right)$

"BINARY SEARCH"

→ Array ko do halves mein divide kareyge.
→ Agar hamara key greater than mid element hai to hum right side search kareyge else left side.
→ Right side ke liye recursive call bhejeyge mid+1 se ub tak or left ke liye lb se mid-1 tak.
→ This goes on until we find element.

```
Binary_search (int arr[], int lb, int ub, int key){
    if (ub >= lb)
        return mid = lb + (ub-lb)/2
    if
        if( arr[mid] == key)
            return mid;
        if(arr[mid] > key)   // searching on left
            return Binary_Search(arr, lb, mid-1, key);
        else
            return Binary_Search (arr, mid+1, ub, key);
            // searching on right.
    }

    return -1;   // we reach here when element
                 not found.
}
```

"QUEUE"

-Using ARRAYS-

```
# define N 5
int queue [N]
int front = -1;
int rear = -1;

void enque (int x)
{
    if (rear == N-1)
        cout<< "Overflow";
    else if (front == -1 && rear == -1)
    {
        front = rear = 0;
        queue [rear] = x;
    }
    else {
        rear++;
        queue [rear] = x;
    }
}

void dequeue () {
    if (front == -1 & row == -1)
        cout << "Underflow";
    else if (found == rear)
        found = rear = -1;
    else { cout << queue [front];
        front++; }
```

```
void display () {
    if (front == -1 & rear == -1)
        cout << " Queue is empty";
    else {
        for (int i = front ; i <= rear ; i++) {
            cout << queue [i];
        }
    }
}

void peek () {
    if (fr == -1 && -)
        cout << " Queue is Empty";
    else
    {
        cout << queue [front];
    }
```

temp = front

temp = front

1	2	3	4	5

top

top2

— USING LINKLIST —

```
class Node {
    int data;
    node * next;
}

class Queue {
    node * front, * rear ;

    void enque (int x) {

        node * newnode = new node(x);

        if (front == 0 && rear == 0)
            front = rear = newnode;
        else
        {
            rear -> next = newnode;
            rear = newnode;
        }
}
```

```
void dequeue () {

    if ( front == 0 )
        cout << " Queue is Empty! ";
    else
    {
        node * temp = front ;
        front = front -> next ;
        free (temp);
    }
}

void display() {

    node * temp = front ;
    while ( temp != 0) {
        ~~pr~~ cout << temp -> data;
        temp = temp -> next ;
    }

void pieek() {
        cout << front -> data ;
}
```

Infinix NOTE

CamScanner

Circular Queue
— Arrays —

```
void enqueue (int x) {

    if (front == -1 && rear == -1)
    {
        front = rear = 0;
        queue[rear] = x;
    }
    else if (((rear+1)%N) == front) {
        cout << "Queue is Full";
    }
    else {
        rear = (rear+1) % N;
        queue[rear] = x;
    }
}

void dequeue () {
    if (fro ——— ) { cout << "Em" ; }
    else if ( front == rear ) {   means only one element is left in queue.
        front = rear = -1;
    }
    else {
        cout << queue[front];
        front = (front+1) % N;
    }
}
```

```
void display()
    while(i != rear){
        cout << queue[rear]
        i = (i+1) % N;
    }
    cout << queue[rear];
}
```

" Using Link List "

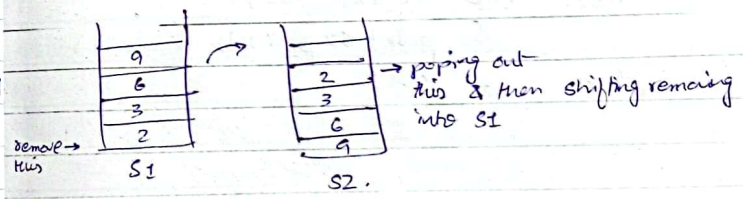
→ we will implement circular queue using circular link list.

" Queue Using Stack "

→ We will use 2 stacks, S_1 & S_2.

→ Enque k liye hum direct S_1 mein push() kate chenge ge.

→ Jaba Dequeue k liye hum phele sarra data S_1 se S_2 mein shift krenge phir S_2 ka phela element pop() krenge phir wapis S_1 mein data shift hoga.




```
void enque (int x) {
    S1.push(x);
    Count++;
}
```

```
void dequeue() {
    if(s1.top==-1 && s2.top==-1){
        cout << "Queue is Empty";
    }
    else {
        for(i=0; i<count; i++){
            s2.push(s1.pop());
        }

        cout << s2.pop();
        count--;
        for(int i=0; i<count; i++){
            s1.push(s2.pop());
        }
    }
}

void display() {
    for(int i=0; i<s1.top; i++)
        cout << s1[i];
}
```

DOUBLE ENDED QUEUE

→ Insertion & deletion is allowed from both ends.
→ It supports both the properties of Stack (LIFO) & Queue (FIFO)

⇒ TYPES OF DEQUE:

① Input Restricted – Insertion by only 1 end deletion can be from both ends.

② Output Restricted – Deletion by only 1 end insertion can be from both ends.

⇒ Deque can be implemented using Circular Array or Doubly Link List.

```
void enqueuefront(int x) {
    if((rear+1)%N) == front) {
        cout << "Queue is full";
    else if(front==-1 && rear==-1) {
        front==rear=0;
        deque[front]=x;
    } else if(front==0) {        //matlab jab front 0 pr hoga to
                                 hum -- nhi krsakte hum usko
        front = N-1;             last index par bhejna hoga
        deque[front]=x;
    } else {
        front--;
        deque[f ]=x;
    }
}
```

```
if((r == 0 && r == N-1) || (front == rear+1))
```

```
void enqueueRear(int x){

    if(((rear+1)%N) == front){
            Queue is full.
    else if (front == -1 && rear == -1){
            f = r = 0;
            deque[r] = x;
    }
    else if (rear = n-1){
            rear = 0; deque[rear] = x; }
    else{
            rear++; deque[rear] = x; }

}

void display(){
    int i = front;
    while (i != rear){
            cout<<
            i = (i+1)%N;
    }
    cout<<    → to print last one
}
```

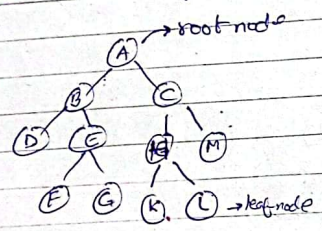
```
void dequeFront(){

    if (f == -1 && r == -1)
            empty;
    else if (f == r)
            f = r = -1;
    else if (f == N-1){
            cout << deque[f];
            f = 0;
    }else{
            cout<<
            front++;
    }
}

void dequeRear(){
    if(f == -1 && r == -1)
            empty;
    else if (f == r)
            f = r = -1;
    else if (r == 0)
            cout<<;
            r = N-1;
    else {
            cout<<;
            r--;
    }
```

Infinix NOTE

CamScanner

TREE

"Trees can be defined as a collection of entities (nodes) linked together to stimulate a hierarchy."



→ The immediate predecessor of any node is calle parent

→ the immediate sucessor of a node is called child

→ The node having no child is called leaf.

→ any predecessor node on the path from root to that node.

Ancestors of L : A, C, A

Desendants of B : E, F, G

→ All children of same parents are called sibblings.

D & E are sibblings.

→ The degree of a node = no. of children

degree of B = 2, M = 0, H = 2

→ The max. degree of any node is the degree of tree.

→ Depth of Node → Length of path from root to that node.

or no. of edges " " " " "

Depth of L : 3

" of M = 2

" of A = 0

→ Height of Node → no. of edges in the longest path from that node to a leaf.

height of G = 0, C = 3, A = 4

→ level of a node : It is also the depth of a node.

level of tree = depth of tree.

n nodes ⇒ n-1 edges.

" BINARY TREE "

→ each node can have atmost 2 childrens. (0, 1, 2)

→ at each level i the max nodes will be 2^i.

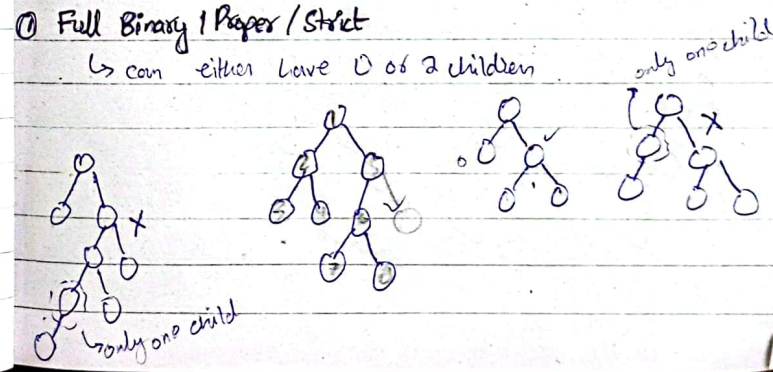
→ max no. of nodes if height h = $(2^{h+1} - 1)$

min no. of nodes if height h = h+1

→ TYPES OF BINARY TREE :

① **Full Binary / Proper / Strict**

↳ can either have 0 or 2 children

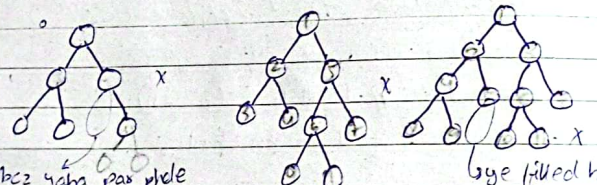


→ no. of leaf node = no. of internal nodes + 1
(every node except leaf node is called internal node including root node)

→ Max nodes = $2^{h+1} - 1$
→ Min no. of nodes = $2h+1$
→ Min height = $[\log_2(n+1)] - 1$
→ Max height = $(n-1)/2$

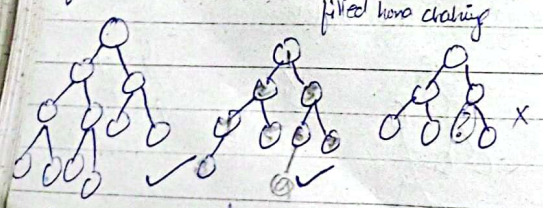
→ ② Complete Perfect Binary Tree

A binary tree is complete binary tree if all the levels are completely filled. Except possibly the last level. And the last level has nodes as left as possible.



bcz yaha par pehle node hona chahiye tha left most will be filled first.

Saare levels completely filled hai lekin leaf nodes main left most filled hona chahiye

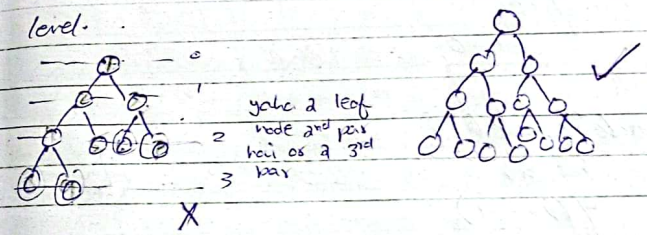


leftmost bhi filled or leafnode k ilawa bhi saare levels filled hai.

→ Max. no. of nodes = $2^{h+1} - 1$
→ Min no. of nodes = 2^h
→ Min heigh = $[\log_2(n+1)] - 1$
→ Max height = $\log n$

③ Perfect Binary Tree:

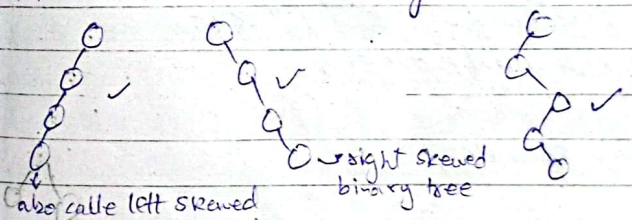
A tree can be a perfect binary tree all internal nodes have 2 children & all leaves are at same level.



→ Every perfect binary is Complete & full Binary Tree
→ Vice versa is not true

④ Degenerate Binary Tree:

All internal nodes have only one child.

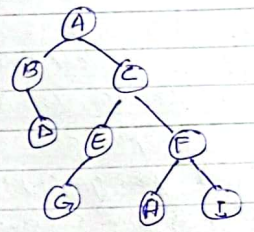


also calle left skewed binary tree when all nodes have left children only.

BINARY TREE IMPLEMENTATION

```
class Node {
    int data;
    node *left, *right;
}

class BinaryTree {
    node *root;

    BinaryTree(){
        root=0;
    }
    node *create(){
        int x;
        if(x==-1)
            return 0;
        cout<< "Enter data: ";
        cin>>x;
        node *newnode = new node(x);
        cout<< "Enter left child of "<<x;
        newnode->left= create();
        cout<< "Enter right child of "<<x;
        newnode->right = create();

        return newnode;
    }
```

BINARY TREE TRAVERSAL



Inorder : left Root Right [Inorder means root reach main]

BDAGECHFI

→ Pnde A se start kareng A hamara root hai ts cos print nhi hoga phir A k left yani B par B ek subtree hai wo bhi root hai neeche wale ka ts cos print nhi hoa kyon B k left par NULL hai or left k baad Root par jaate hai ts is wajo se B print hogaya phir Right yani 'D' print hogaya phir backtrack karke B k root par jayenge which is A to A print hogaya phir A ka right 'C' but C not print bcz it is a subtree is a root of the nodes below it so C k left par gaye i.e E phir E bhi root hai neeche wale ka to G, G is printed bcz no more nodes from Root is printed i.e E phir E ka right nhi hai to C print kra which is root phir C ka right F F not printed bcz root so H, H is printed then F(bcz root), then right i.e I.

Preorder: Root Left Right [pre matlab root phele]

ABDCEGFHI

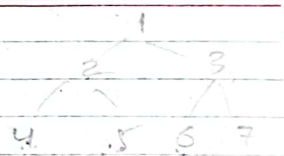
Phele root print krna phir left mein gaye i.e 'B' B is root as it is sub tree so 'B' is printed than B ka left NULL so right par gaye D is printed phir backtrack krke A par gaye phir A ka left hogaya K ab right which is C and C is a root of subtree so C printed than left of C i.e E, E is a root of subtree so E printed E ka left G is printed. E ka right NULL so backtrack to C phir C ka right i.e F, F is a root so F printed phir left i.e H than right i.e I.

PostOrder: Left Right Root [post means root sabse end mein]

DBGEHIFCA

A se start kra A is root so not printed phir left gaye i.e B is also root so not printed phir left null so right par gaye i.e D which is a leaf node so D printed phir root node of D is printed i.e B phir back track krke A par gaye A ka left hogaya ab right krna hai i.e C is also root so left i.e E, E is also root so G, G is printed. than no right so root of G i.e E is printed ab C ka right i.e F lekin F root hai K H, H is printed phir I, phir wapis C phir wapis C ka root i.e A.

```
void Preorder (node *root){
    if (root == NULL)
        return;

    cout<< root->data;
    Preorder(root->left);
    Preorder(root->right);
}

void Inorder (node *root){
    if (root == NULL)
        return;

    Inorder(root->left);
    cout<< root->data;
    Inorder(root->right);
}

void Postorder (node* root){
    if (root == NULL){
        return;

    Postorder(root->left)
    Postorder(root->right)
    cout<< root->data;
```



Infinix NOTE
CamScanner

LEVEL ORDER TRAVERSAL :

```
void levelOrderTraversal(){
    Queue<node*> q;
    q.push(root);
    q.push(NULL) ;    // pushing NULL for printing
                      // each level in a seperate line
                      // using NULL as a seperator

    while(!q.empty()){
        node* temp = q.peek();
        q.deque();

        if(temp==NULL){ // means one level complete
            cout<<endl;
            if(!q.empty()){
                q.enque(NULL);
            }
        }else{
            cout<< temp->data <<" ";
            if(temp->left != NULL)
                q.enque(temp->left);
            if(temp->right != NULL)
                q.enque(temp->right);
        }
    }
}
```

→ Creating a Tree Using LevelOrder Traversal:

```
void BuildfromLOT()
{
    int data;
    cin>> data;
    root = new node(data);
    Queue<node*> q;
    q.enque(root);

    while(!q.isEmpty(){
        node* temp = q.peek();
        q.deque();

        cout<< "Enter left of" <<temp->data;
        cin>>left;
        if(left != -1){
            temp->left = new node(left);
            q.enque(temp->left);
        }
        cout<< "Enter right of" <<temp->data;
        cin>> right;
        if(right != -1){
            temp->right = new node(right);
            q.enque(temp->right);
        }
    }
```

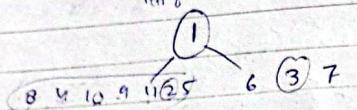
⇒ Constructing BT from Preorder: (Rnt L R) & Inorder

Preorder: 1 2 4 8 9 10 11 5 3 6 7

root is always 1st in pre order

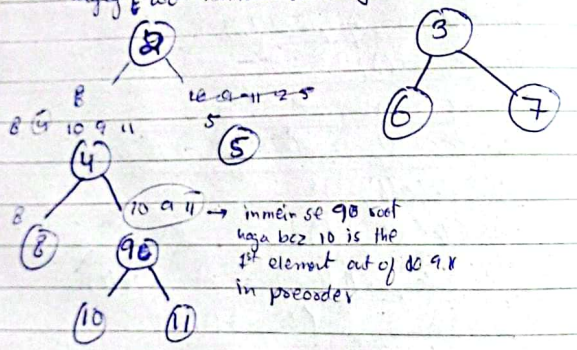
Inorder: 8 4 10 9 11 2 5 1 6 3 7

left of root



ab humein is sub-tree ka root find karna hai.

→ in elements mein se jo element preorder mein sabse pehle aayega wo hamara root hoga. in this case it's 2

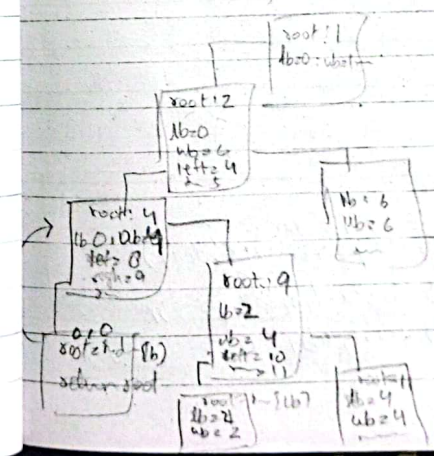


→ inmein se 9 root hoga bcz 10 is the 1st element out of 10 9 11 in preorder

→ Left & right 9n preorder se dekhenge.

→ Root jo hai wo preorder se dekhenge.

```
root = preorder[0]
root->left = buildTree(inorder, lb, mid-1, lb);
root->right = buildTree(i+1, ub);

return root;
}

for (int i=lb; i<ub; i++)

for (int j=0; j<n; j++)
    if (inorder[i] == pre[j])
        if (j < minindex)
            minI = j;
        break;
break;
}
```



```
node* build(int lb, int ub, int *inorder, int *pre, int pre_len){

    if(lb==ub){
        node *newnode = newnode(inorder[lb]);
        return newnode;
    }
    int minindex=100;
    int index;

    for(int i=lb; i<=ub; i++){
      for(int j=0; j<=pre_len; j++){

        if(inorder[i]==preorder[j])
        {
          if(j<minindex)
          {     minindex=j;
                index=i;
          }
          break;
        }
      }
    }

node *newnode = newnode(inorder[index]);
newnode->left = Build(lb, index-1, inorder, pre, pre_len);
newnode->right = Build(index+1, ub, " " " );
return newnode;
}
```

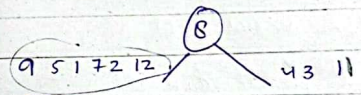
⇒ Construct Binary Tree from Post-Order & Inorder.

Postorder: 9 1 2 12 7 5 3 11 4 8

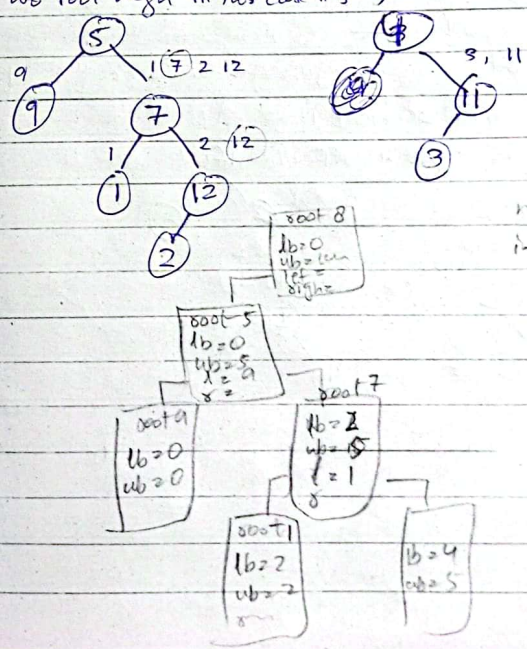
Inorder: 9 5 1 7 2 12 8 4 3 1

→ For Root we check postorder (last is root)

→ For left & right inorder



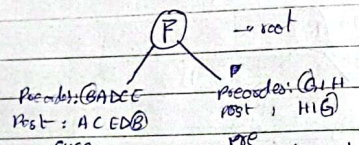
In main se jo element post k andar last se phele aatha hai wo root hoga in this case it's 5



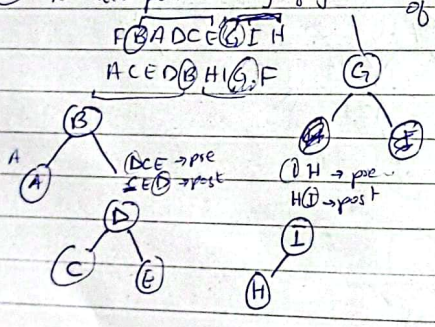
Construct Binary Tree From Post & Preorder.

→ By using postorder and preorder we cannot construct a unique binary tree but it's possible to construct unique full binary tree.

Preorder: FBADCEGIH
Postorder: ACEDBHIGF



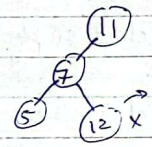
F ka jo ~~prede~~ successor hai ~~post~~ pre mein wohe jo left side wale elements hai including it post preorder mein wo humare ~~prem~~ (F) ka left part bam jayega. remaining will be part of right subtree



"Binary Search Tree"

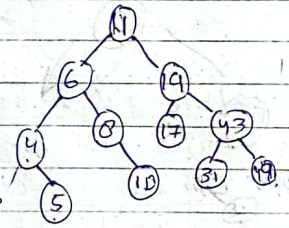
→ There will be atmost 2 nodes.

→ The left ~~node~~ subtree of a root will have value less than root & right one will have greater value than root



→ ye 7 se baari hai lekin 11 se bhi baari hai to 11 ke left par nhi aaskti

Q: Draw BST by inserting the following no. from left to right. 11, 6, 8, 19, 4, 10, 5, 17, 43, 49, 31




```
node* insert (node* root, int val)
    if (root==NULL) {
        root = new node(val);
        return root;
    }
    if (val < root->data)
        root->left = insert (root->left, val)
    else
        root->right = insert (root->right, val)
```

v = 6 & 10

→ DELETION IN BST:

There are 3 case:

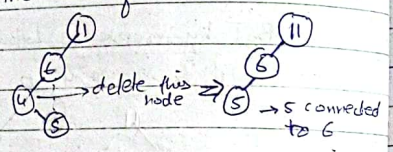
→ The node we want to delete has 0 child

→ " " " " " " 1 child

→ " " " " " " 2 child.

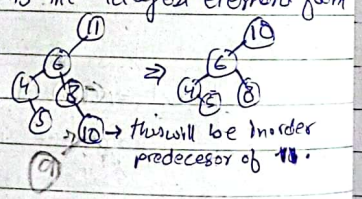
CASE#01: If a node has 0 child simply break the link of that node with it's root

CASE#02: If a node has 1 child then the child will be linked to the root of the node which we want to delete.

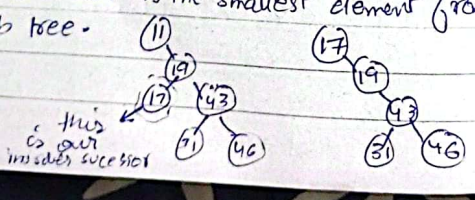


CASE#03: If a node has 2 child. then we can replace that node either with it's inorder predecessor OR inorder sucessor.

→ The inorder predecessor is the largest element from left subtree.



→ inorder sucessor is the smallest element from right sub tree.

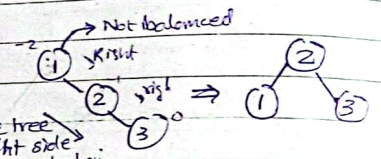



```
val = root->data;
delete(right)
delete(root)
if(root = null)
root = NULL
val = root->left->data;
if(val < root->data)
if(val == root->data)
    return deleteB(root)

if(val < root->data)
    left = delet(root->left, val);
    return root;
else
    right = delete(root->right, val);

if(root->left == NULL && root->right == NULL) {
    free(root);
    return null;
else if(root->left != NULL && root->right == NULL)
    temp = root->left;
    free(root);
    return temp;
else if(root->left == NULL && root->right != NULL)
    temp = root->right;
    free(root);
    return temp;
else {
```

"AVL TREES"

→ It is a BST.
→ Each node should be balanced.
→ A balanced node mean (height of left subtree − height of right subtree) must be equal to −1, 0, 1.
→ Duplicate elements not allowed.
→ After every insertion of a node the balance factor of each node will be checked.

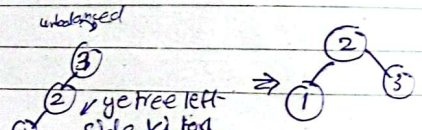
CASE# 01:
RR
right right



ye tree right side par jaa rha hai to yaha Left rotation kareng

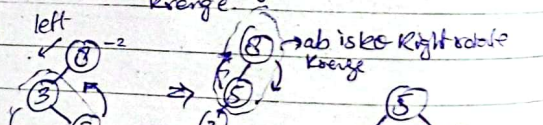
→ Middle element hamesha root bn jayeg or upper wale ko left rotate krlia.

CASE# 02:
LL
left left



ye tree left side ki taraf hai to right rotate krenge.

CASE# 03:
LR.
↓
Left right combination hoga to phele left rotate phir right



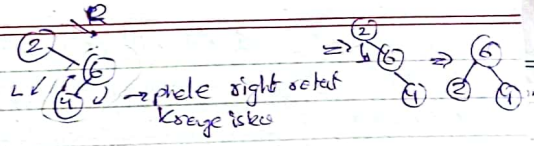
→ phele neeche wala rotate kareng
ab ye right side ki taraf gaya hai to left rotate

→ ab is ko Right rotate kareng

eg: Right left combination hoga to phele Right rotate phir left

RL

CASE# 04:



→ phele right rotate kareng is ko

"PRO'S & CON'S OF EACH SORTING ALGORITHIM"

★ INSERTION: B → O(n), W → O(n²), A → O(n²)

ADVANTAGE:
↳ For nearly sorted data, it's efficient (nearly O(n))
↳ Requires only a constant ammount of memory space O(1).
↳ Efficient for small data sets
↳ Stable.

Note: Selection req. fewer swaps than insertion, but more comparisions

DISADVANTAGE:
↳ Needs a large no of element shift.
↳ Has a lot of swaps

★ SELECTION SORT: B → A → W → O(n²)

ADVANTAGE:
↳ 60% more efficient than Bubble Sort.
↳ Good for small data set, but no better than Insertion.
↳ Doesn't need extra memory

~~Ha~~

DISADVANTAGES: $B \to A \to W \to O(n^2)$

↳ Has a lot of comparisions

↳ Unstable

* MERGESORT: $B \to A \to W$ $O(n \log(n))$

ADVANTAGES:

↳ Stable

↳ Performs well on large data sets Worst case → $O(n \log n)$

DISADVANTAGES:

↳ Requires extra memory to store the merged sub arrays

↳ Not in-place : Req. additional memory.

↳ Not optimal for small dataset.

* QUICK SORT: $B \to O(n \log(n))$, $A \to O(n \log(n))$, $W \to O(n^2)$

ADVANTAGES:

↳ Efficent on large datasets

↳ It has a low overhead, as it req. only small ammount of memory to function.

DISADVANTAGES:

↳ Worst case is $O(n^2)$, which occurs when pivot is choosen poorly

↳ Not good for small data sets

↳ Not Stable

* COUNTING SORT: $B \to A \to W$ $O(n+m)$ where n = size of input array, m = k + counting array

ADVANTAGES:

↳ Works faster than all comparision sort like, Merge, Quick, if the range of input is of the order of the no. of input.

↳ Stable

DISADVANTAGE:

↳ Not work on decimal values

↳ not efficent if the range of values to be sorted is large

↳ Not an in-place : uses extra space.

* RADIX SORT: $O(d*(n+b))$ where d = no. of digits, n = no. of elements, b = base of the number system being used

"HEAP"

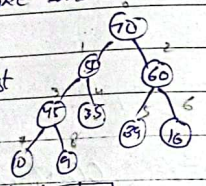
→ A tree based data structure
→ It is a Complete Binary Tree.
→ TYPES OF HEAP:
① Max Heap
② Min Heap

→ condition not true for root as root have no parent

① MAX HEAP: $A[parent(i)] \geq A[i]$

⇒ Har node ki value uske child node k barabar ya uske bari hogi.

⇒ Root Node will be the largest element



X	70	60	60	45	35	39	16	10	9
0	1	2	3	4	5	6	7	8	9

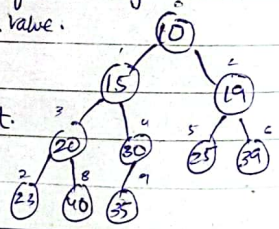
→ except root

② MIN HEAP: $A[parent(i)] \leq A[i]$

⇒ Har node ki value uske child se chotti hogi ya equal.

⇒ For every node i, the value of node is greater than or equal to it's parent ~~node~~. value.

⇒ Minimum element will be root.



X	10	15	19	20	30	25	39	23	40	35
0	1	2	3	4	5	6	7	8	9	10

→ How To Find a parent of a Node?

Parent = $\lfloor i/2 \rfloor$ (floor value will be considered)

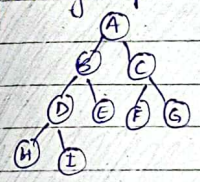
a[4] = 35

parent(a[4]) = 4/2 = 2 ⇒ a[2] = 50 ✓

⇒ INSERTION IN MAX HEAP:

- We will always insert from left most side as it's a complete BT.
- Then will check if the value of current node is greater than it's parent then swapping will be done else no swapping.

Array Representation of CBT:



A	B	C	D	E	F	G	H	I
0	1	2	3	4	5	6	7	8

case#I

A	B	C	D	E	F	G	H	I
1	2	3	4	5	6	7	8	9

case#II

Case#01
↳ If a node is at i^{th} index
↳ left child = $[(2*i)+1]$
↳ right child: $[(2*i)+2]$
↳ Parent child: $\left[\frac{(i-1)}{2}\right]$

Case#02
↳ left child: $(2*i)$
↳ right child: $[(2*i)+1]$
↳ Parent: $\left[\frac{i}{2}\right]$

Infinix NOTE
CamScanner

```
class Heap {
    int A[100];   // size
    int n;
    Heap() {  A[0] = -1;
              n = 0;   // size initialy 0.
    }
InsertHeap (int *A, int n, int val)
                        for Case #02
{
    n = n+1;    // size will be increased
    A[n] = value;
int i = n;

while (i > 1)    // means loop will run till root node
{
int parent = ⌊i/2⌋
  if (A[parent] < A[i])
  {
      swap(A[parent], A[i]);
      i = parent;   // update i value & check
                       again.
  }
  else
  {     return;
  }
}
}
```

Time Complexity: $O(n)$.

→ DELETION IN MAX HEAP:

→ T.C : $O(\log(n))$

→ Only root node can be deleted.

→ last element of array will be replaced with root.

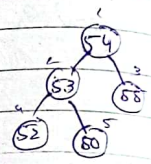
→ Size will be decremented.

→ Now we compare the child nodes, the one which is greater will be compared to root node if root is less than that child value swapping will be done.

→ The process goes on to last index.

```
void DeleteMaxHeap( ) {
        int i = 1;
      A[1] = A[n] ⇒ swap(A[1], A[n]);
        n--;
      while (n < size)
      {     max child          left child   right child
        int child = FindMax(2*i , (2*i)+1);
        if (A[child] > A[i]) {
              swap (A[child], A[i]);
              i = child;
        }
        else {
              return;
        }
}     }
```

"HEAPIFY ALGORITHM"



→ Har leaf node khud ek heap hoti hai.

→ Ek array representation mein $(n/2+1)$ se n^{th} index tak leaf nodes hote hai.

54	53	50	52	50
1	2	3	4	5

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

→ Heapify function ka kaam hai ek node ko usi sh jaga par leke jana.

→ Agr humein bola 53 ko heapify karo to wo 53 k sub tree mein 53 ko sh jaga par le jayega.

→ Ab ~~chunke~~ chu nke saari leaf nodes apni correct positions par hai to unhe heapify nhi kreng we have to ~~to~~ only traverse from $n/2 \rightarrow 0$ index (right → left)

→ agr kai i^{th} index diya wa hai to wo i^{th} index se leker neeche tak poore tree ko heap mein convert kardegi.

void Heapify (int *A, int n, int i)

while (i < n)

if (A[i] < left child && right child)

max = i

else {

max = R (l, right

swap (A[max], a[i])

i = max;

}

}

"HEAP SORT"

→ First we will convert they array into a Max Heap using heapify Algo.

→ We will delete the 1st element as it will be the largest & ~~plac~~ swap it with last element of array.

→ Then follow the same process of deletion.

→ After completing deletion we will again delete the 1st element & swap with last & this goes on untill whole tree is deleted.

30	20	7	9	15
1	2	3	4	5

void HeapSort (int *a, int size) {

while (size > 0)

DeleteMaxHeap (R size);

}

"HASHING"

→ Hashing is a method or a technique by which we can search or retrieve a data in O(1).
→ It is a general concept that involves converting input data into fixed size value.

→ Hashmap is a specific datastructure that uses hashing to implement an associative array.
→ Hash table is another ds that uses hashing. It is similar to a hashmap.
→ Insertion, Deletion, Searching O(1).

```
// creation
unordered_map <string,int> m;
```

key and its corresponding value.

```
//insertion
// 1
pair <string, int> p= make_pair("Sam",3);
m.insert(p);

//2
pair <string, int > pair2("hello", 2);
m.insert(pair2)
//3
m["world"]=7
m["world"]=5;
```

this will update prev value.
→ Possibar insert karenge to updation hogi.

```
// Search
cout<< m["sam"] << endl.
cout << m.at["hello"];

cout<< m.at["ali"]; // error as we have not inserted this key.

cout<< m["ali"]; // no error because [] will create a new key and give it a '0' value.

//size
cout << m.size();

// Count
cout<< m.count("syed"); → 0 // as 'syed' is not present
cout << m.count("sam"); → 1

//erase
m.erase("hello");
```

Infinix NOTE

CamScanner

// for traversing whole map.

unordered_~~pair~~ map<int, int>:: Iterator it = m.begin();

while (it != m.end()) {
    cout << it->first << " " << it->second;
    it++
}

⇒ unorder_map mein sequence wise display nahi hoga.

⇒ ~~order~~ map mein sequence wise hota hai.

→ Unorder have constant time - O(1)

→ Orderemaps have O(logn)

Keys: 8, 3, 6, 10, 15, 18, 4

If we want to search any value by linear O(n), or binary O(logn). Lekin Hashing k zariye O(1) mein hojayega.

→ We have to arrange these keys in hash table in such a way that we can find them in O(1).

→ For this we will use Hash Function.

→ We have to map the key values into hash table using hash function.

① $h(x) = k \bmod n$; where k = key, n = size of hash table

② ~~Division Method~~

③ Folding Method

If we have an array of 0-99 and a key value 1123 this cannot be stored in 0-99 so now

11
23
34 → now we will store 1123 at 34th index.

→ Some times there is a collision like we need to store key values 6 & 26 in a Hashtable of size 10.

6 % 10 = 6
26 % 10 = 6 } so both values cannot be stored at 6 this is "COLLISION".

" COLLISION RESOLVING TECHNIQUES "

" we cannot remove a collision, we can minimize a collision "

① OPEN HASHING (closed Addressing)

⇒ CHAINING METHOD:

A = 3, 2, 9, 6, 11, 13, 7, 12

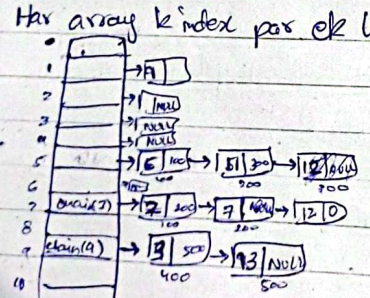
$h(k) = 2k+3$ $m = 10$

Use division method (k % m) & closed addressing to store these values. h(k)

key	Location ([2k+3] % 10)
~~key~~ 3	[2(3)+3] % 10 = 9
2	[2(2)+3] % 10 = 7
9	[2(9)+3] % 10 = 1
6	[2(6)+3] % 10 = 5
11	[2(11)+3] % 10 = 5
13	[2(13)+3] % 10 = 9
7	[2(7)+3] % 10 = 7
12	= 7

```
struct node {
    int data;
    node* next;
}
Node * chain[size];
void init () {
    for(i -> size)
        chain[i]=NULL;
}
void insert(val)
    newnode (val)
    key = val % size
    if(chain[key]== NULL)
        chain[key]= newnode;
    else
        temp = chain[key];
        while(temp->next!=0){
            temp=temp->next;
        }
        temp->next = newnode;
```

Har array k index par ek link list ka head hoga



→ string ko HASH table mein insert krnne ka tareeqa
→ Add the ASCII values for the string mod with n.

② CLOSED HASHING (Open Addressing)

→ Linear Probing
→ Quadratic Probing
→ Double Hashing

% m isliye kiya table ngs last index ko to starting par chale jayein.

LINEAR PROBING:

Insert k[i] at 1st free location from (location+i)%m where i range from 0 → m-1

→ Same Qs & same Hashfunction as prev.

bar bar jo i ki value wahi attempt.

→ Probing means searching for a free place.

→ When a collision occurs we fill linearly search for a free space after that index & place the value.

→ Agr last element hoga to 1st par search krenge.

→ we also need to keep the count of the attempt because this will help in searching we can directly write [key+attempt] to find the element.

Key	locations	Probes/attempts
3	9	1
2	7	1
9	1	1
6	5	1
11	5	2
13	9	2
7	7	2
12	7	6

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

order of elements in hashtable

13, 9, 12, _, _, 6, 11, 2, 7, 3

→ Advantages:

↳ No extra space

→ Disadvantages:

↳ Searching time $O(n)$ → searching mein jese hi empty space aayega matlab element not found!

↳ Deletion Difficult.

↳ Primary Clustering → a group of keys at one place

↳ Secondary "

↳ space waste (as half of table is empty).

To get best time complexity ~~all the key~~ no. of elements in hash table must be less than or equal to half the size of table. If m=10 then max element 5.

0	
1	22
2	43
3	64
4	8
5	
6	9
7	1

cluster

⇒ Best & Avg. case for searching $O(1)$, but for worst $O(n)$.

☆ QUADRATIC PROBING:

→ Some examples.

→ Insert k_i at first free location from $(loc + i^2)\%m$ where $i = 0 \to (m-1)$

Key	loc	Probe
3	9	1
2	7	1
9	1	1
6	5	1
4	5	2 → $(5+0^2)\%10 = 5$ ✗ $(5+1^2)\%10 = 6$ ✓
13	9	2 → $(9+0^2)\%10 = 9$ ✗ $(9+1^2)\%10 = 0$ ✓
7	7	2 → $(7+0^2)\%10 = 7$ ✗ $(7+1^2)\%10 = 8$ ✓
12	7	5 → $(7+0^2)\%10 = 7$ ✗ $(7+1^2)\%10 = 8$ ✗ $(7+2^2)\%10 = 1$ ✗ $(7+3^2)\%10 = 6$ ✗ $(7+4^2)\%10 = 3$ ✓

	hash table
0	~~12~~
1	~~9~~
2	
3	~~12~~
4	
5	6
6	11
7	2
8	7
9	3

☆ DOUBLE HASHING:

→ We have 2 hash function. If 'c' collision occur we use the another function to resolve.

$h_1(k) = k\%10$

$h_2(k) = R - (k\%R)$; where R is a prime number less than size. ∴ size = 10 so R = 7

$7 - (k\%7)$

$h'(x) = (h_1(x) + i * h_2(x))\%10$ → this will be used if collision occurs.

$i = 0, 1, 2, \ldots$

Key space

~~25~~
25
15
35
95

H.T

0	
1	15
2	35
3	
4	95
5	~~25~~
6	~~95~~
7	
8	25
9	

$h'(25) = (5 + 1*3)\%10 = 8$

$7 - (25\%7)$

$7 - 4 = 3$

$h'(15) = (5 + 1*6)\%10 = 1$

$7 - (15\%7)$

$7 - 1 = 6$

$h'(35) = (5 + 1*7)\%10 = 2$

$7 - (35\%7)$

$7 - 0 = 7$

$h'(95) = (5 + 1*3)\%10 = 8$ ✗

$= (5 + 2*3)\%10 = 11$ ✗

$= (5 + 3*3)\%10 = 4$

Infinix NOTE

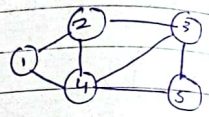
CamScanner

"GRAPHS"

"GRAPH REPRESENTATION"

① ADJACENCY MATRIX:

→ n×n Matrix → n = no. of vertices



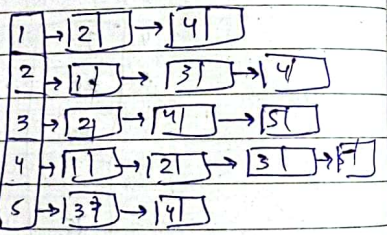
$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 1 & 1 & 0 \\ 3 & 0 & 1 & 0 & 1 & 1 \\ 4 & 1 & 1 & 1 & 0 & 1 \\ 5 & 0 & 0 & 1 & 1 & 0 \end{array}$$

a[i][j] = 1 if i & j are adjacent

otherwise = 0

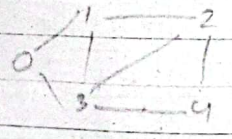
② ADJACENCY LIST:

1 → 2, 4
2 → 1, 3, 4
3 → 2, 4, 5
4 → 1, 2, 3, 5
5 → 3, 4



jese seperate chaining karte the.

0 → 1, 3
1 → 0, 2, 3
2 → 1, 3, 4
3 → 0, 1, 2, 4
4 → 2, 3



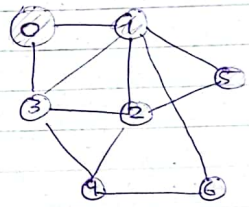
"BFS TRAVERSAL"

→ Breadth First Search (level order traversal)

→ Can start from any node

→ ~~Can~~ Starting from '0'.

→ We try to traverse all the adjacent nodes to the root, first.



visited | 0 | 1 | 3 | 2 | 5 | 6 | 4

QUEUE: | ~~0~~ | ~~1~~ | ~~3~~ | 2 | 5 | 6 | 4

Result: | 0 | 1 | 3 | 2 | 5 | 6 | 4

| 0 | 0 | 1 | 0 | 3 | 1 | 1 |

Jesei result mein zero save hoga ke queue se delete hojayegi or usके adjacent वाले को que मेंं डाल देंगे.

phir '1' के लिए adjacent check करेंगे. Que मेंं सिर्फ unvisited vertices push होंगे जैसे 1 → 0, 3, 2, 5, 6 adjacent हैं लेकिन हम 2, 5 & 6 push करेंगे.

जो जो elements que मेंं push होंगे वो visited के array मेंं भी push होते रहेंगे taके visited को track रख सकेंगे.

→ There can be many results for the BFS traversal OR we can make a bool array.

0 3 4 6 1 2 5

"DFS TRAVERSAL"

→ Depth First Search
→ We will use stack.
→ Any node as root node.
→ First Root will be inserted. Then printed.
→ Then any one of the adjacent vertex of root will be push in stack.
→ Only unvisited will be pushed.
→ Agr hum ese point par pohnch jayein jaha se aage koi unvisited nodes nhi hai hum backtrack karenge.
→ current node ko stack se pop karenge phir jo next hoga usko ~~vis~~ lige unvisited dekhenge. & goes till stack is empty.

```
void BFS (int s_vertex)
{
    queue <int> q;
    bool * vis = new bool[vertices] {false};

    q.push(s_vertex);
    vis[s_vertex] = true;
    while (!q.empty)
    {
        int temp = q.front();
        q.pop();
        cout<< temp<< " ";
        node * temp2 = AdjList.Links[temp];
        while (temp2 != 0)
        {
            if (!visited[temp2->data])
            {
                q.push(temp2->data);
                vis[temp2->data] = true;
            }
            temp2 = temp2->next;
        }
    }
}
```

Infinix NOTE

CamScanner

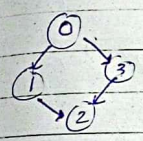
```
void DFS (int s_vertex){
    stack<int> s;
    s.push(s_vertex);
    bool * visited = new bool[vertices] {false};
    visited [s_vertex] = true;
    cout << s.peek() << " ";
    while (!s.empty())
        int temp = s.peek();
        node * temp2 = AdjList.links[temp];
        while (temp2 != 0)

            if (!visited[temp2->data])
            {
                s.push(temp2->data);
                cout<<temp2->data<<" ";
                visited[temp2->data] = true;
                break;
            }
            else
            {
                temp2 = temp2->next;
            }
        }
        if (temp2==0)
            s.pop();
}
```

"TOPOLOGICAL SORT"

→ DAG → Directed Acyclic Graph (no cycle)

→ Topological Sort only implemented on DAG. ~~why~~

↳ linear ordering of vertices such that for every edge $U \to V$ :: U always appears before V in that ordering



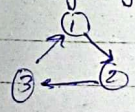
⇒ 0, 1, 3, 2

0 → 1, 3
1 → 2
2 → ?
3 → 2

0 → 1 ek edge hai or order mein 0, 1 se phele aata hai. 1 se 2 edge hai to order mein 1, 2 se phele hai. Same for other vertice. So valid topological sort.

⇒ 3, 2, 1, 0 (0 → 1 se edge hai lek ordering mein 0 1 se phele nhi hai.

• Why only on Acyclic?



1, 2, 3

3 se 1 edge hai
lekin order mein
1 3 se phele hai

→ So if we are not able to get a valid topological sort it means cycle is present.

Using DFS:

→ Is job stack se pop krna shr hote hai ya kisi node se wapis aaye hongi to usko kisi dusre stack mein push krenge. To jo stack mein data hoga wo topological sorted hoga.

Using BFS (Kahn's Algo):

→ We need to have an ~~queue~~ array containing indegree of each vertice

→ The element having indegree '0' is push in

→ Phir us element k neighbours dekhenge or

→ phir queue se jis element ko pop krenge of us element jiska neighbour hoga uski indegree -1 krenge.

→ Or un neighbours mein jis in.deg 0 hai wo push hojayega.

→ job tak queue empty nhi hojaati this goes on.

→ or jo element queue se pop() hoga wo result array mein store hojayega.

→ If no.of vertices in result array is not equal to total vertices that it is not a valid topological sort. Mean a cycle is present.

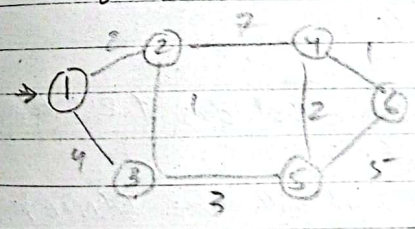
→ SHORTEST PATH UNDIRECTED (NON-WEIGHTED)

→ Phele BFS se pora traverse krke hai ~~aur~~ vertice k parents store krlenge.

→ ~~phir~~ phir hum parent array ke destination se src tak traverse krenge to shortest path miljayega.

```
int index = dest;
s.push(index);
while (index != src)
        index = parent[index];
        s.push(index);
}
```

→ DIJKSTRA'S ALGO:



	2	3	4	5	6
1	∞	∞	∞	∞	∞
1	12	4	∞	∞	∞
		3	9		
			9	6	
			8		

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

priority_queue <int, vector<int>, greater<int>> pq;
↳ STL syntax for min heap.

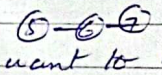
→ ~~pa~~ EK distance array bnega jismein sirf soc node ka distance 0 hoga baaqi sabka ∞.

→ phir hum ~~h~~ soc node k neighbours ko dekhenge or unka distance update karenge [update jabhi hoga jab purane distance se chota ho hamara new distance] or phir priority que mein push karenge.

→ Jab tak program chalega jab tak Q empty na hojaye.

(DISJOINT SETS).

For eg we have a graph having 2 components



if we want to know that does 1 & 5 belong to same component.

→ By using DFS & BFS it will take O(N+E) time complexity.

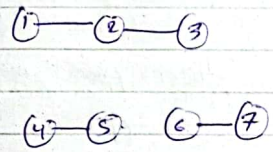
→ By using disjoint sets we can solve this in CONSTANT time.

→ It gives us two functionalities
① Finding Parent ()
② Union () → this can be achieved by Rank or size.

→ Union ka matlab hai do ~~edges~~ nodes ko apas mein combine krna.

→ Initially every node is single none is connected to another.

(1, 2) → this means joining 1 & 2
(2, 3)
(4, 5
(6, 7)
(5, 6)
(3, 7)



→ If at this stage we want to know whether 1 & 4 are in same component then our ans will be [No]

→ If we go one stage further than our ans will be [yes].

→ This is [Dynamic graph] in which the configuration of graph changes at each step

→ If we are to ask that whether U & V are in same component at any step than disjoint set will give us ans in constant time.

• UNION BY Rank:

→ Two arrays will be used Rank & Parent.

→ Initially the Rank will be 0 for all.

→ And parent will be the same value as index bcz every node is initially parent of itself.

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

Union (U, V)

① Find ultimate parent of U & V, pU & pV.

② Find rank of pU & pV. [parents will be connected not U & V

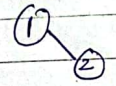
③ Connect smaller rank to larger rank always. If equal rank you can connect with any one.

(ultimate parent means top most parent of a node).

Union (1, 2).

↳ pU = 1, pV = 2

rank = 0, rank = 0



Now parent of 2 is 1 & the rank of 1 has incr. because ① is connected to above 2:

r → | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
1 2 3 4 5 6 7

| 1 | 1 | 3 | 4 | 5 | 6 | 7 | → p
1 2 3 4 5 6 7

Union (2, 3):

↳ pU = 1, pV = 3

rank = 1, rank = 3

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

| 1 | 1 | 1 | 4 | | | | → p
1 2 3 4 5 6 7

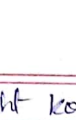
↳ ? we have connected a smaller rank with larger so rank not incr.

→ Rank basically height ko depth ko incr. krenge. Jab 2, 1 se connect hoa to 1 ki height incr. hogai so rank++. Jab 3 attach hoa to height mein farq nhi para so rank not inc.

Un (4, 5)

pU = 4, pV = 5

r = 0, r = 0



→ rank(4) = 1

→ p(5) = 4

Un (6, 7)

pU = 6, pV = 7

r = 0, r = 0

→ rank(6) = 1

→ p(7) = 4

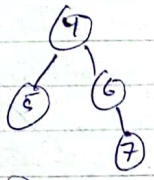
p | 1 | 1 | 1 | 4 | 4 | 6 | 6 |
1 2 3 4 5 6

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

Un (5, 6)

pU = 4, pV = 6

r = 1, r = 1



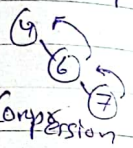
Now by attaching 6—7 to 4 the height of 4 increases so we add 1 to rank of 4. & parent of 6 also chge. i.e 4.

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

| 1 | 1 | 1 | 4 | 4 | 4 | 6 |
1 2 3 4 5 6 7

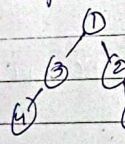
→ Now if any one ask that whether 1 & 7 belong to same component?

⇒ Find Parent → ultimate parent find kerta hai to agr mujhe 7 ka malloom kerna hai to phele wo 6 par jayeg phir 4 par or phir 4 ka apna check kerega.



this will take log(n) time but we need O(1) so will we do Path Compression that is ? we no that the ultimate par. of 7 is 4 so we directly attach 7 with 4.

→ Path compression mein rank reduce nhi kerenge

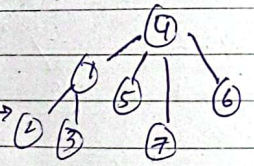


if we compress 7 & reduce rank but the other side height is 2 so we cannot change rank.

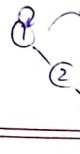
Union (3,7)

pU = 1 , pV = 4

r = 1 , r = 2



rank(4) remain same ? we attach a larger rank with smaller.



agar humein rakhna hai to 1 kaun ke parent hai

Find P(u)

```
{
    if (u == p[u])
        set u;

    set parent[u] = Find(parent[u]);
}
```

→ or k ultimate parent ka parent wo khud hote hai

} path compression.

```
Class DisjointSet {
    int * rank;
    int * parent;

    DisjointSet(s) {
        rank = new int[s];
        p — "
        // initialize value rank = 0 & parent itself
    }

    int FindUParent(int node){
        if (node == parent[node])
            set node;

        set parent[node] = FindUParent(parent[node]);
    }
```

→ ultimate

```
void UnionByRank(int u, int v) {

    int up = FindUParent(u);
    int vp = F_ _ _ (v);

    if(up == vp)    // both have same ultimate parent
        return;

    if(rank[up] < rank[vp]) {    // no inc. in
        parent[up] = vp;         // rank
    }
    else if (rank[vp] < rank[up]) {
        par[vp] = up;
    }
    else {    // both equal
        parent[vp] = up;
        rank[up]++;
    }
```

hum base wale k saath chote wale ko jor rhe hai take rank update na karna pade

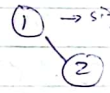
UNION BY SIZE :

→ Size array initialize to 1 & parent with itself.

size → | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
1 2 3 4 5 6

par → | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
1 2 3 4 5 6 7

(1,2) → pu = 1 , pv = 2
size = 1 , s = 1

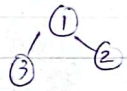


→ size of 1 inc.

→ same size to kisi ko kisi kisi k sath jodo.

(2,3)

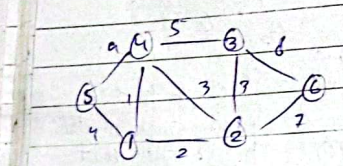
pu = 1 , pv = 3
size = 2 , s = 1 → so now 3 will attach to 1 & size not inc. since smaller is att to larger. but parents will be changed.




```
void UnionBySize(int u, int v){
    int pu = F_ _ (u)
        pv = " (v);

    if(pu == pv)
        return

    if(size[pu] < size[pv]){
        par[pu] = pv;
        size[pv] += size[pu];
    else {
        par[pv] = pu;    // equal case
        size[pu] += size[pv];
```

KRUSKAL'S ALGO

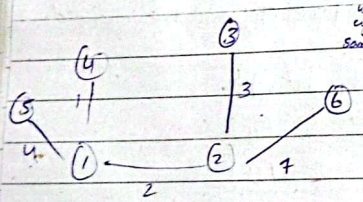
→ First sort all edges by weights.
→ Then Phir hum has edge ko union karte
Sherge lekin sirf un edge ka krenge jo
ka ek component ka parent nhi hai (means dono
k ultimate parent same nhi hai). Agar same
hoe to union nhi krenge. [same component hum cycle ki wajase dekh rhe hai]



Sort acc to edge

w	U	V
✓1	1	4
✓2	1	2
✓3	2	3
✗3	2	4
✓4	1	5
✗5	3	4
✓7	2	6
✗8	1	6
✗9	4	5

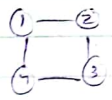
as 2 & 4 will make cycle both have same ult par.



MST = 17

FLOYD WARSHAL ALGO:

→ Ye bhi shortest path calculate krta hai har vertex se har vertex ka.



to 1 se 2, 3, 4 ; phir 2 se 1, 3, 4 sabka sabse shortest nikaalega.

→ EK distance array hoga jismein sirf direct distance store hoga.

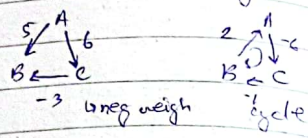
→ hum has vertex ka via has vertex traverse krenge. yani ag 1 se 3 jana hai to phele 1→2 phir 2→3 dekhenge or usko comparision krenge 1→3 wale path ka jo shortest hoga likhenge.

```
for(int k=0; k<size; k++){

    for(int i=0; i<size; i++){

        for(int j=0; j<size; j++){

            if(a[i][k]+a[k][j] < a[i][j])
                a[i][j] = a[i][k]+a[k][j];
        }
    }
}
```

→ After doing this if any diagnol element contains -ve value it means it contains a -ve cycle.

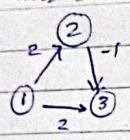
BELLMAN FORD

→ Used for shortest path with -ve weights but not neg cycle.



→ Applied on Directed & undirected.

→ Gives shortest dist from src → dest.



src = 1 & dest = 3

→ we will apply formula ↓

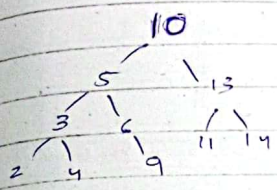
```
if (dist[U] + wt < dist[V])
        dist[V] = dist[U] + wt;
```

(n-1) times on all edges of graph
n = no. of nodes

→ or phir ek dafa or lagenge yehi formula or ab agar koi distance update hogaye then it mean -ve cycle exist. Else no -ve cycle.

CIEL VALUE IN BST:

Ciel means smallest val greater than or equal to key.



key = 8, so ciel = 9

key = 11, ciel = 11 (11 is present)

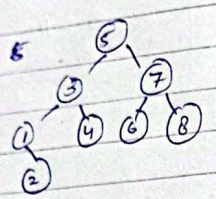
key = 12, ciel = 13

```
int ciel = -1
while (root)
{
    if (root -> data == key) {
        ciel = root -> data;
        return ciel;
    }
    if (key > root -> data) {
        root = root -> right;
    }
    else
    {   ciel = root -> data;
        root = root -> left;
    }
return ciel;
}
```

→ If do BST traversal Right Root left → Descending

k^th Smallest element in BST:

if k=3 (3rd smallest)
so val = 3



1 2 3 4 5 6 7 8

We will traverse inorder to will get a sorted order.
→ Inorder of BST is always sorted.
→ At every step we will Inc. counter and when ctr == k we will reach our ans.

For largest → [(totalnodes - k)+1] smallest
5th largest = ((8-5)+1) smallest
= 3rd smallest

FLATTEN A BT to LinkList:
→ Usi BT ko rearrange krna hai LL mein

```
prev = NULL;
flatten(node) {
    if(node == NULL)
        ret;

    flatten(node → right)
    flatten(node → left)

    node → right = prev;
    node → left = NULL;
    prev = node;
}
```

bilkul right se start krenge or wapsi aate ne change krte jayenge.

Q: We are given any array & integer K we need to find 2 values ~~such that a & b~~ such that a+b = k.

k = 10

0	1	3	4	5	6	8

s 1 1 e

→ do pointers hoenge start & end. done ki values add krenge.
→ If add krne pair k se kari aati hai to end-- or choti aati hai to start++.

0 + 8 = 8 → val less than 10 (start++)
1 + 8 = 9 → " "
3 + 8 = 11 → now greater so end--
3 + 6 = 9 → now less so start++
4 + 6 = 10 ✓

If start > end no val found!.

DATA STRUCTURES

" Way to arrange data in main memory for efficent usage."

eg: Arrays, Linked List, Queues, Stack

ADT (Abstract Datatype)

It is a model to make a datastructure

ADTs are like user defined data types which defines operation on values using function without specifying what is there inside the function & how the operation are performed.

eg: Stack ADT (a stack consist of same type arranged in a sequential order).

Operations:

→ initializ(), Push(), Pop(), isEmpty().

⇒ Here we are only specifying which functions can be used on stack but no providing their inner functionalities

→ ADT is like a black box which hides the inner structure & design of the datatype from user.

→ There are multiple ways to implement ADTs.

- A stack ADT (which itself is a datastructure) can be implemented using arrays & Linked List.

• ~~Why ADT?~~

→ ADT tells us what is to be done and datastructure tells us how to do it.
like Stack ADT uses array & Linked list

→ Which data structure to choose for an ADT?
In reality, different implementation of ADT are compared for time and space efficiency. The one best suited Acco. to the current requirement of the user will be selected.

TYPES OF DATA STRUCTURES

Linear DS

↓

A data structure is linear when all elements are arranged in a linear (sequential) order

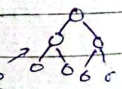
- array
- Queue
- Linked list
- Stack

it has only one sucessor & one predecessor except 1st & last element.

Non-Linear DS

↓

There is no sequential arrangment of the elements

- tree 
- graph

Static Data Structure

↓

In these type of DS, the memory is allocated at Compile time, max size is fixed.

Adv. → Fast access
DisAdv. → Slower insertion & deletion

eg: Array

Dynamic Data Structures

The memory is allocated run tim. Max size is flexible.

Adv. → Faster ins. & del.
Dis → Slower access.

eg: Linked List

"ARRAY IN DS"

→ TRAVERSAL (visit every element of any array at least once).

```
int a[5] = { 1, 2, 4, 6, 8 }
for (int i = 0 ; i<5 ; i++){
        pr—("%d", a[i]);
}
```

→ INSERTION

```
int a [8] = { 3, 7, 9, 20, 30 }
printf(" Enter the pos "); scanf("%d", &pos);
printf(" Enter the number ");
scanf (" %d ", &num);
```

pos = 4th
num = 40

0	1	2	3	4	5	6	7
3	7	9	20	30			

new no.

```
for (int i = length-1 ; i >= pos-1 ; i++) {
        a[i+1] = a[i];
}
a[pos-1] = num;
length++;
```

a[i+1] = a[i]

pos-1 is liye kia kai q k pos = 4th & i = 3.

Best Algo: Agr humein unsorted array mein insertion krni ho to hum jis position par insert krna ka raha usse element ko last mein shift krdenge & new element ko req. position par. This will take constant time.

a[size] = a[pos-1]
a[pos-1] = num.

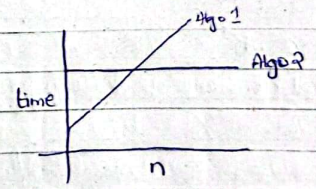
TIME COMPLEXITY

"Jese jese input ka size bardhta jaa rha hai vese vese algorithm k runtime par bha effect pash rha hai"

O(1) → Big O of 1 means an algo takes constant time.
aap input/data ka jo bhi size lelo ye algorithm constant time mein run hoga.

O(n) → Big O of n means it takes linear time. Jitna input/data ka size hoga utna time approximately usko hi time lagega

Example: Algo 1 takes O(n)
Algo 2 take O(1)



n. ko value hai k use upper dono functions ka behaviour change nhi hoga.

ASYMPTOTIC NOTATION

"It is used to compare two algorithms"

5 types of Asymptotic Notations
- Big Oh O
- Big Omega Ω
- Big Theta Θ

time

size of input

- **Big O :** A function $f(n)$ is said to be $O(g(n))$ if & only if there exist a constant (c) and a constant ~~n₀ such that~~ n_0 such that $0 \le f(n) \le c\,g(n)$ where $n \ge n_0$

$f(n) = n^3 + 1$

if we take $O(n^3)$ ab isko condition se verify krenge.

$$0 \le n^3 + 1 \le c \cdot n^3$$

if $n = 2$ & $c = 1$ than condition is not satisfied!

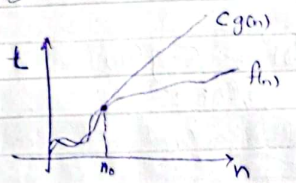
$9 \le 1 \times 8$ ✗

But if $n = 2$ & $c = 2$ than condition is satisfied!

$9 \le 2 \times 8$

$9 \le 16$ ab koi bhi 'n' lelein or $c = 2$ rakhein to condition satisfy hojayegi.

To ab iska matlab ye function $O(n^3)$ hai lekin ye function $O(n^2)$, $O(n^5)$ & so on bhi hai lekin humein iski least value find krni hai jo k is case mein $O(n^3)$ hai.



→ algo ko isse zyada time nhi lagega

→ worst case

→ Least Upper bound (at most)

↳ yaani zyada se zyada kitna time lagega

$f(n) = 2n^2 + n$ $\quad O(?)$

$2n^2 + n \le c \cdot g(n^2)$ → n^2 isliye liya q k $f(n)$ mein dominant term hai

$2n^2 + n \le 3n^2$ → $c = 3$ isliye liya q k barabr mein $2n^2$ pehle se mojood hai to dono apas mein cancel out hojayenge or n ki value less than 0 ajyegi yaani negative.

$2n^2 + n \le 3n^2$

$n \le 3n^2 - 2n^2$

$n \le n^2$

$1 \le n$

$\boxed{n \ge 1}$

↳ n greater than equal to 1 k liye koi bhi values put krein to hamesha eqn satisfy hogi.

- iska matlab ab chahe kitch bhi hojaye mere algo ka time $3n^2$ se zyada nhi lagega.

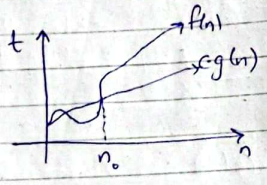
Big Omega (Ω)

→ Best Case → sabse kam time lagega

→ Greatest Lower Bound. (At least)

↳ kam se kam kitna time lega.

Condition : $f(n) \geq c \cdot g(n)$ $\quad n > n_0$



matlab g(n) esa function ho k hamesha n ki ~~kisi~~ n_0 se baad values par chota answer de.

if $f(n) = 2n^2 + n$ $\quad \Omega(g_n)$?

$f(n) \geq c \cdot g(n)$

$2n^2 + n \geq 2 \cdot n^2$ → n^2 q liya (q k $2n^2+n$ se chota

$c = 3$ lete ~~to~~ condition satisfy nhi hoti.

kuch bhi le sakte hai $n, n\log(n)$ n^2 etc lek humein closest chahiye for sagr $f(n)$ hai to hum 0, 1, 2 nhi bhke 4 lenge.)

$\cancel{2}n^2 + n > \cancel{2}n^2$

$\boxed{n \geq 0}$

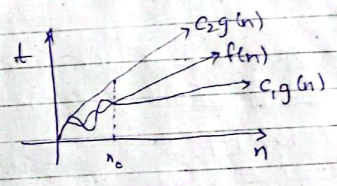
- iska matlab ye hoa k chahe kuch bhi hojaye mere algo ka time $2n^2$ se kam nhi lagega.

Big Θ Theta :

The Θ notation bounds a function from above & below, so it defines exact asymptotic behaviour.

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

$n \geq n_0$



Big
- $f(n)$ is $O(g(n))$ → $a \leq b$
- $f(n)$ is $\Omega(g(n))$ → $a \geq b$
- $f(n)$ is $\Theta(g(n))$ → $a = b$

Small
- $f(n)$ is $o(g(n))$ → $a < b$
- $f(n)$ is $\omega(g(n))$ → $a > b$

a b

ARRAYS

pos = 3

DELETION:

0	1	2	3	4
6	10	15	20	30

```
for (i=pos-1 ; i<size-1; i++){
        a[i]= a[i+1]
}
size -- ;
```

a[2] = a[3]
a[3] = a[4]

i=2 pass
2<5
15=20 ✓
i=3 3<5
20 = 30
i=4 4<5 ✗
30 = --

agar hum size tak change to ye iteration fegul chalegi.

Best Algo: We can simply pick the last element and place it in the position which we want to delete. Applicable to only unsorted array. this will take constant time.

LINKED LIST

Collection of elements of similar data type but it is not stored in contagious location.

→ single element is called Node.

→ ~~Each node 8 bytes hai~~

ek node do parts bate hai ek main element store hoga or dusra pointer hoga jismein next node ka address store hoga. Or last Node ka pointer mein NULL hota hai

So a single node takes 8 byte (for int)

→ Array mein humne ek matlab size allocate karta to change nhi kr sakte lekin linked list mein hum size badha sakte hai.

→ Address of the 1st node is stored in the head pointer

→ DRAWBACK : We cannot directly access any element. We have to access it sequentially. Agr humein last element tak jana hai to starting se traverse krna hoga.

- Accessing an element takes O(n) whereas in array it takes O(1).
- Insertion & deletion is easier than arrays.
- Binary Search not possible in Linked List.
- It is of Dynamic size.

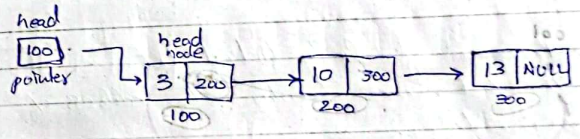
```
struct node {
        int a; -> to store data
        struct node *next; -> to store address of next node.
}
```

" TYPES OF LINKED LIST "

- Singly ll (Linked list)
- Doubly ll
- Circular ll
- Doubly Circular ll

" SINGLY LINKED LIST "

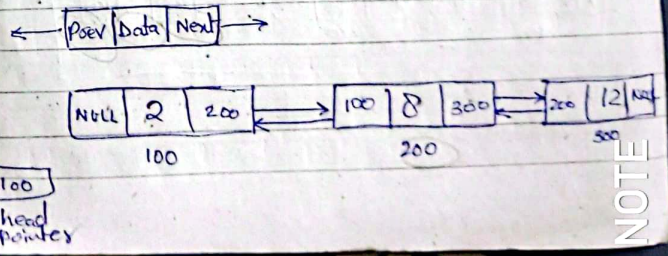
→ Most common form of ll.
→ Each node consist of two part (Data & Link part)
→ Only forward traversal is possible.
→ It is called singly bcz it has only single connection with the proceeding node.



" DOUBLY LINKED LIST "

→ Each node consist of 3 parts. (1 Data part & 2 Address/Link part)
→ It has two pointer
  ↳ One pointer to it's previous node
  ↳ Second pointer to its next node
→ It has double links.

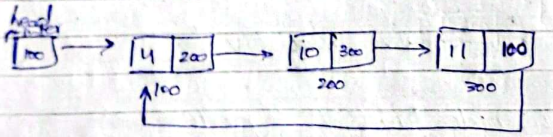
Node



```
struct node {
        int data;
        struct node *next;
        struct node * prev;
}
```

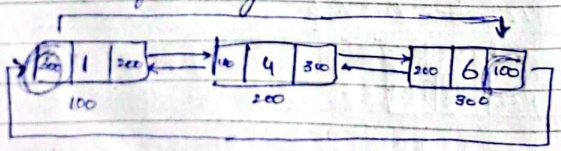
" CIRCULAR LINKED LIST "

→ Variation of Singly linked List
→ The last node contains the address of 1st node.



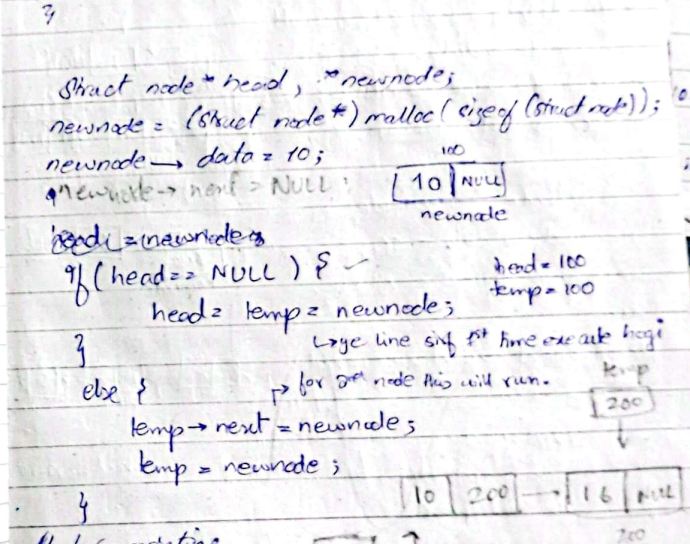
" DOUBLY CIRCULAR LINKED LIST "

→ Variation of Doubly linked List.

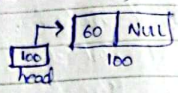


" IMPLEMENTATION OF LINKED LIST "

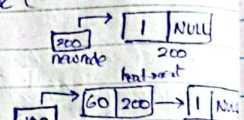
```
struct node {
        int data;
        struct node * next;
}

struct node * head, *newnode;
newnode = (struct node *) malloc( sizeof (struct node));
newnode → data = 10;
newnode → next = NULL;
head = newnode;
if ( head == NULL ) {
        head = temp = newnode;
}
else {
        temp → next = newnode;
        temp = newnode;
}
// for printing
temp = head;
while (temp != 0) {
        printf("%d", temp → data);
        temp = temp → next;
}
```



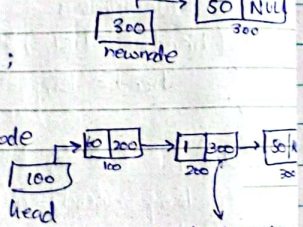
struct node * head = malloc(———)
head → data = 60;
head → ~~t~~next = NULL;



struct node * newnode = malloc(———)
newnode → data = 1;
newnode → next = NULL;
head → next = newnode;



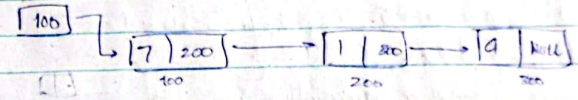
Structnode * newnode = ma(———)
newnode → data = 50;
newnode → next = NULL;
kiya
head → next → next = newnode



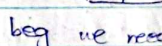
200
↳ 200→next
iska matlab 2nd node ka link part.

"INSERTION IN LINKED LIST"

Insertion At Beginning:



Struct node * head, * newnode;
newnode = malloc(——→) // Creating a newnode which is to be inserted



// for inserting at beg we need to first change the value of first node link part in the newnode's link part.
// then save the address of newnode in head.

newnode → link = head;
head = newnode;

ye hoaaaaaa hai

Insertion At End:

// Here we have to only store the address of newnode in the last node's link part.
head → link → link → link = newnode;

100's link part
200's link part
300's link part

```
struct node * ptr = head;
*ptr = add_at_end(ptr, 88);
ptr = add_at_end(ptr, 90);
ptr = " " (ptr, 80);
```

```
struct node* add_at_end(struct node* , int dat){
struct node* temp = malloc(—)
temp->data = data;
temp->link = NULL;
ptr->link = temp;
return temp;
}
```

O(1)

OR

// another method is to traverse the list till end then insert new node.

```
temp = head;
while (temp!=0) {
        temp = temp->link;
}
```

temp->link != 0

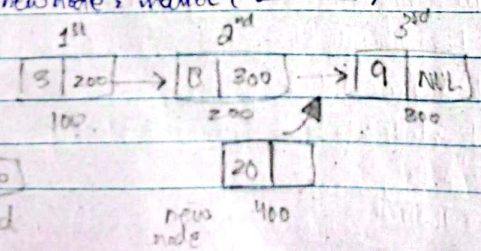
↳ Time Complexity: O(n)

// Now when this loop end our temp pointer will point to the last node. So we simply now save the value of new node in last node's link part

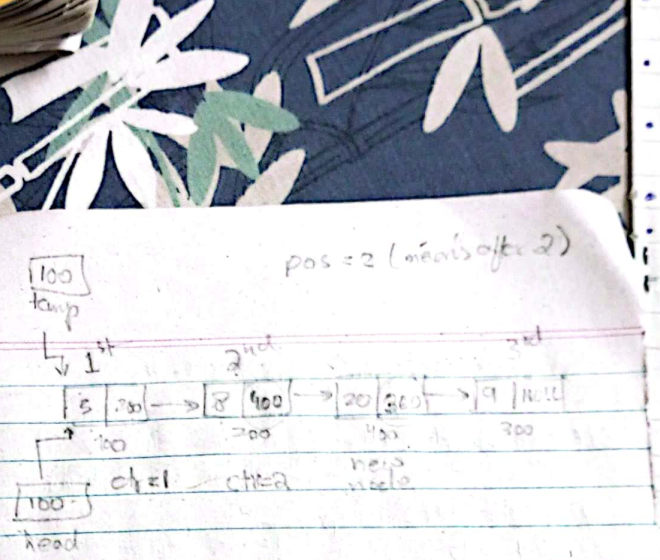
```
temp->link = newnode;
```

Insertion At Given Location :

```
struct node *head, *newnode, *temp;
newnode = malloc(———)
```



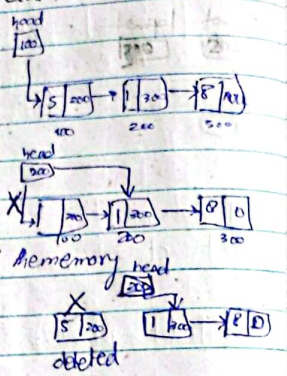
pos = 2 (means after 2)




```
temp = head;
int ctr = ;

while(temp!=0) {

    if(ctr==pos){
        newnode->link = temp->link;
        temp->link = newnode;
        break;
    }
    temp = temp->link;
    ctr++;
}
```

"Deletion In Linked List"

Delete For Beginning:

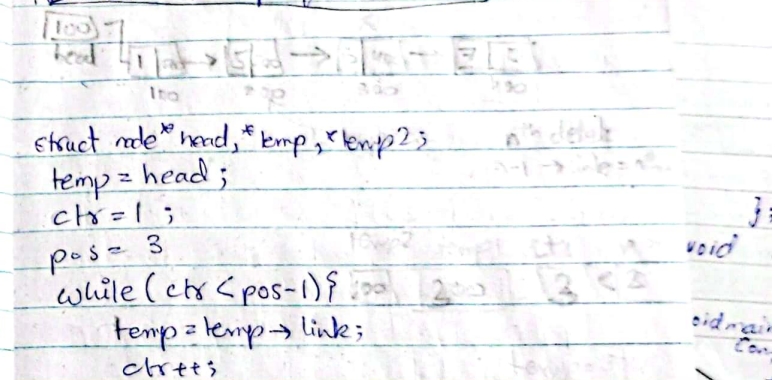



```
struct node* head, temp;
temp = head;
head = head->link; //
free(temp);   //released memory
```

Delete From End:

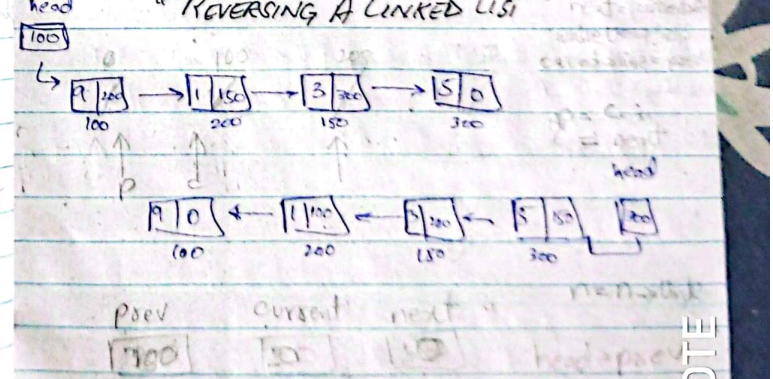
```
struct node *head, *temp1, *temp2;
temp1 = head;
while (temp1->link != 0){
    temp2 = temp1;
    temp1 = temp1->link;
}
if (temp1 = head){   //iska matlab sirf ek node
    head = 0;        hai or usse humne link tod
}                    dia or end mein free temp
                     kr dia
else {
    temp2->link = 0;   //ye 2nd last node
}                      k link part KO
free(temp);            NULL kr dega.
```

Deletion At Specific Position:




```
struct node* head, *temp, *temp2;
temp = head;
ctr = 1;
pos = 3
while (ctr < pos-1){
    temp = temp->link;
    ctr++;
} temp2 = temp->link;
temp->link = temp2->link;
free(temp2);
```

"Reversing A Linked List"

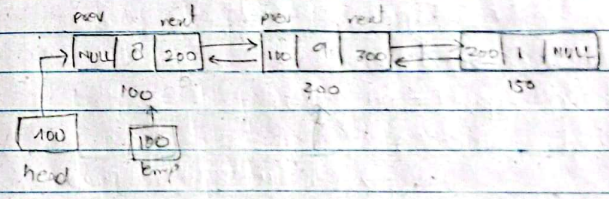


```
struct node *prev, *current, *next, *head;

prev = NULL;
current = head;
next = current -> link;

while (current != 0) {

    current -> link = prev;
    prev = current;
    current = next;
    next = next -> link;
}

head = prev;
```

"DOUBLY LINKED LIST"

```
struct node {
    int data;
    struct node* prev;
    struct node* next;
}
```

"IMPLEMENTATION"




```
struct node * head; *newnode, * temp;
newnode = malloc (——);
newnode -> data = 8;
newnode -> prev = 0;
newnode -> next = 0;
if (head == 0) {
temp = head = newnode;
} else {
temp -> next = newnode;
newnode -> prev = temp;
temp = newnode;
```

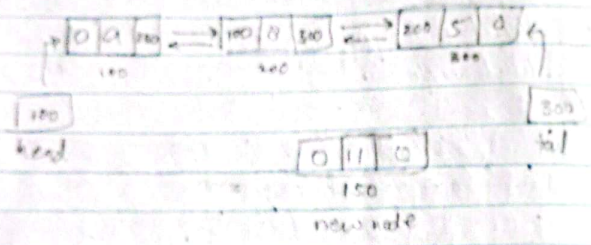
*tail

temp kei jaga tail pointer maintain kerenge jo insertion mein help kregg.

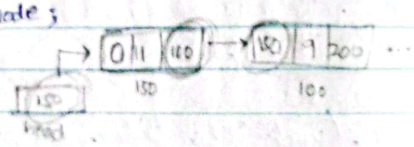
tail stores the address of last node

INSERTION

• INSERTION AT BEGINING :



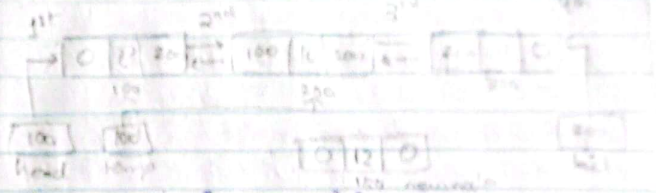

```
newnode->next = head;
head -> prev = newnode;
head = newnode;
```



• INSERTION AT END :

```
tail->next = newnode;
newnode->prev = tail;
tail = newnode;
```

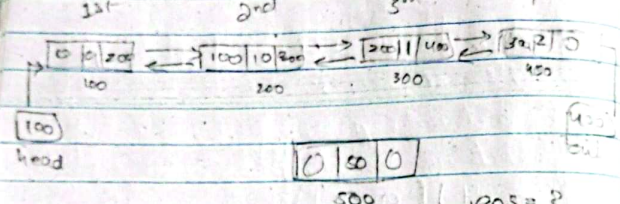
• INSERTION AT POSITION :




```
if(pos > 4 || pos < 1){
        invalid!                    pos = 3
else if (pos == 1){
        // insertion at beg;
}else if (pos == 4){
        // insertion at end;
else {
        int ctr=1; temp=head;
        while (ctr < pos-1){
                temp = temp->next;
                ctr++;
        }
        newnode->prev = temp;
        newnode->next = temp->next;
        temp->next = newnode;
        newnode->next->prev = newnode;
```

means if pos = 2 than we will insert after 2nd node.

INSERTION AT AFTER A POSITION:



pos = ?

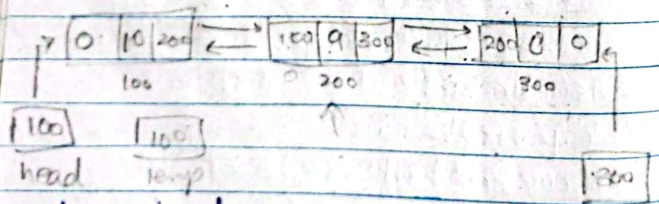
// Only our while loop will change !

// We now traverse till "pos" not "pos-1".

```
while (ctr < pos){

}
```

"DELETION"

DELETION FROM BEGINING:

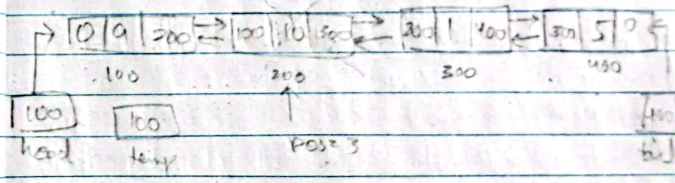



```
temp = head;
head = head->next;   OR head->next->prev = 0;
head->prev = 0;      head = head->next;
free (temp);
```

DELETE FROM END:

```
temp = tail;
tail = tail->prev;   OR tail->prev->next = 0;
tail->next = 0;      tail = tail->prev;
free (temp);
```

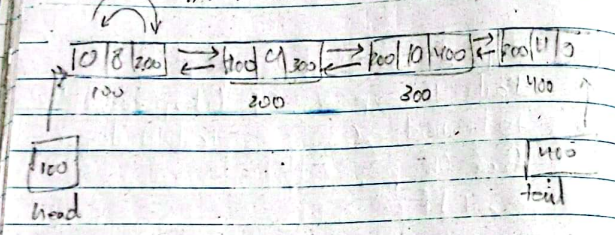
DELETE FROM SPECIFIC POSITION:



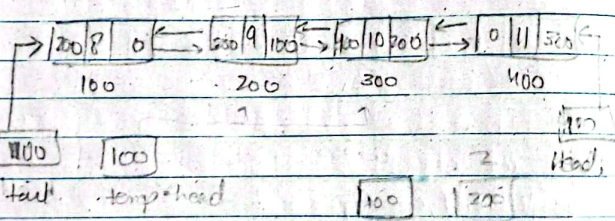

```
temp = head;
int ctr = 1
pos = 3
while (ctr < pos){
    temp = temp->next;
    ctr++;
}
temp->prev->next = temp->next;
temp->next->prev = temp->prev;
free (temp);
```

temp temp2

"REVERSE"



New List




```
temp= head;
while (temp!=0) {
        temp2 = temp->next;
        temp->next= temp->prev;
        temp->prev = temp2;

        temp = temp->prev;
}
temp = head;
head = tail;
tail = temp;
```

"CIRCULAR LINKED LIST"


```
struct node {
        int data;
        struct node* link;
}
struct node * head, *newnode, tail,
newnode = malloc (——);
head = newnode
if (head == 0) {
        head = temp = newnode;
        new node -> next = head;
}else {
        temp->next = newnode;
        temp = temp->next;
        temp->next = head;
}
        temp= head;
do {    printf(——);
        temp= temp->next;
while (temp!=head);
```

eg. while loop lagaya to phle condition tu false hogayegi.

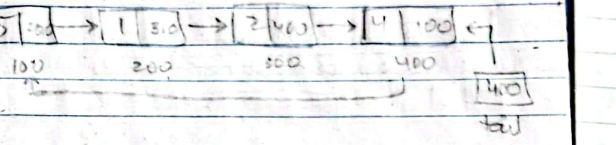
"IMPLEMENTATION"
using only tail

```
if (tail==0){
    tail = newnode;
    tail->next = newnode;
}else{
    newnode->next = tail ~~->next~~;
    tail->next = newnode;
    tail = newnode;
}
```

→ Traversing through tail;

```
temp = tail;
do{
    pr_(__)
    temp = temp->next;
}while(temp != tail->next);
```

"INSERTION"



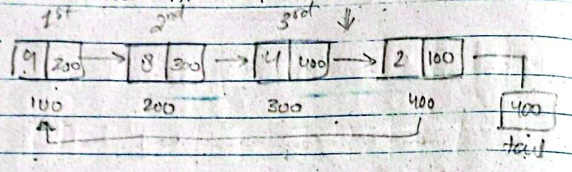
INSERTION AT BEGINING:

```
if (tail==0){
    tail->next = newnode;
    newnode->next = tail;
}else{
    newnode->next = tail->next;
    tail->next = newnode;
}
```

INSERTION AT END:

```
newnode->next = tail->next;
tail->next = newnode;
tail = newnode;
```

INSERTION AT ANY POSITION:



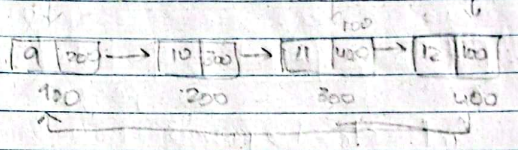
pos=3 means insertion after 3.

```
temp= tail -> next;
int ctr=1;
int pos= 3;

while ( ctr < pos) {
    temp = temp -> next;
    ctr++;
}

new node -> next = temp -> next;
temp -> next = new node;
```

temp	ctr
100	1
200	2
300	3

"DELETION"

DELETION FROM BEGINING:

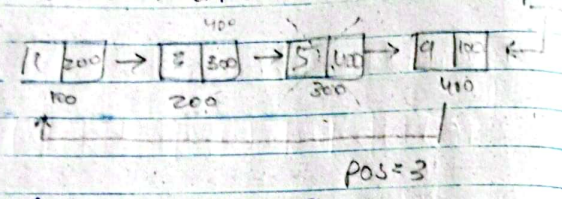



```
temp = tail -> next;
tail -> next = temp -> next;
free (temp);
```

DELETE FROM END:

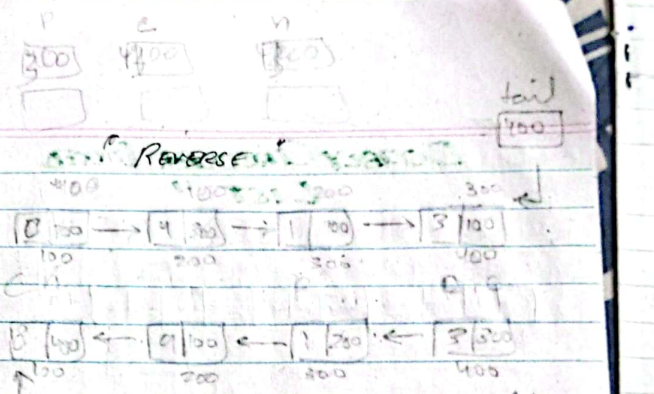
```
temp = tail -> next;
while ( temp -> next != tail)
        temp = temp -> next;

temp -> next = tail -> next;
free (tail);
tail = temp;
```

DELETION FROM A CERTAIN POSITION:

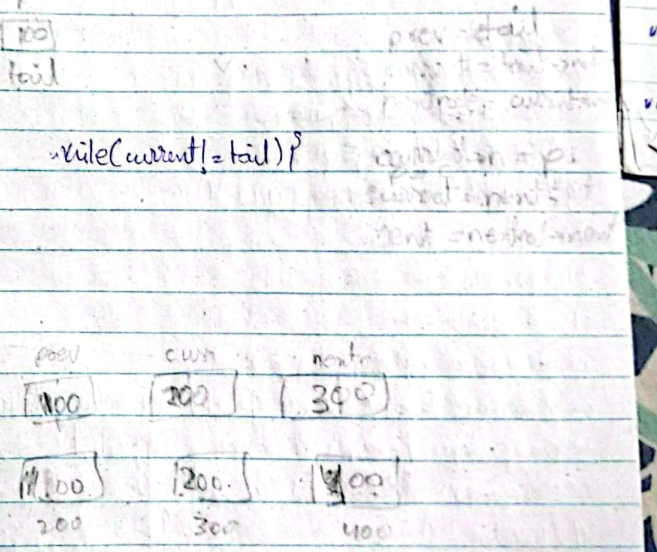



```
int ctr=1, pos=3
temp = tail -> next;

while (ctr < pos-1){
        temp = temp -> next;
        ctr++;
}

temp2 = temp -> next;          //300 is saved
temp -> next = temp2 -> next;  //400 saved to 200
free (temp2);
```

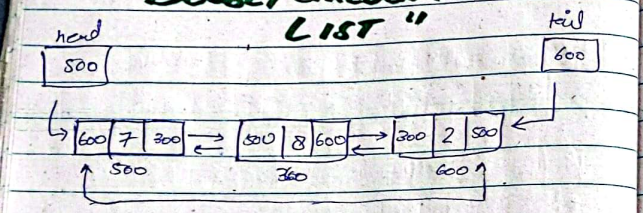
REVERSE




```
while(current != tail){
```



"DOUBLY CIRCULAR LINKED LIST"




```
struct node
{   int data;
    struct node * prev;
    struct node * next;
} *head, *tail;

struct node * newnode;
head = 0;
newnode = malloc(——)
newnode -> data = 7
newnode -> prev = 0;
newnode -> next = 0;
if (head == 0) {
    head = tail = newnode;
    head -> next = head -> prev = head;
else {
    tail -> next = newnode;
    newnode -> prev = tail;
    newnode -> next = head;
    head -> prev = newnode;
}
```

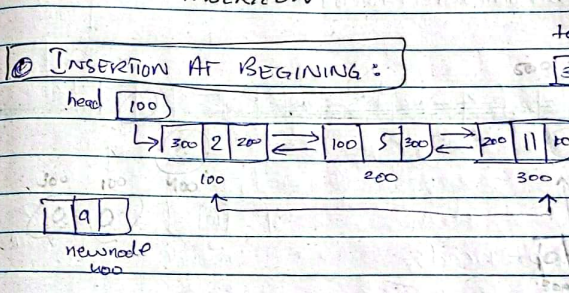
```
while(temp != tail){
    print(——)
    temp = temp -> next
}
```

OR

dowhile

"INSERTION"

INSERTION AT BEGINING:

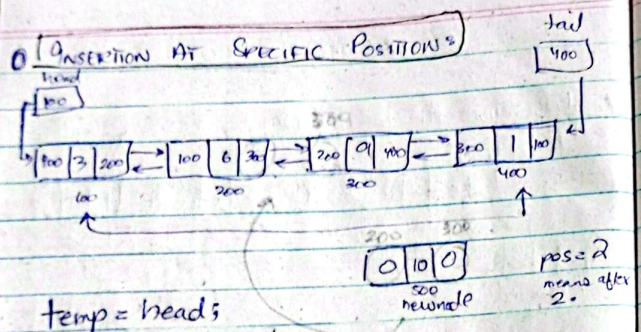



```
head -> prev = newnode;
newnode -> next = head;
newnode -> prev = tail;
head = newnode;
tail -> next = head / newnode;
```

INSERTION AT END:

```
newnode -> prev = tail;
newnode -> next = head;
tail -> next = newnode;
tail = newnode;
head -> prev = newnode / tail;
```

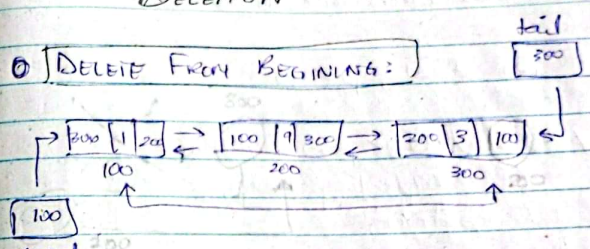
Insertion at Specific Positions:




```
temp = head;
int ctr = 1, pos = 2

while ( ctr < pos ) {
        temp = temp -> next;
        ctr++;
}

newnode -> prev = temp;
newnode -> next = temp -> next;
temp -> next -> prev = newnode;
temp -> next = newnode;
```

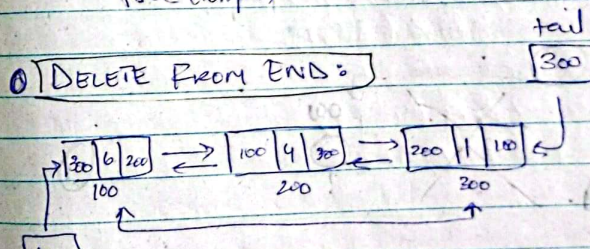
"Deletion"

Delete From Beginning:




```
temp = head;
temp -> next -> prev = tail
head = temp -> next;
tail -> next = head;
free (temp);
```

Delete From End:

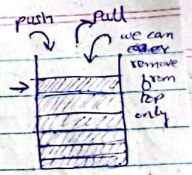



```
temp = tail;
temp -> prev -> next = head;
tail = temp -> prev;
head -> prev = tail;
free (temp);
```

DELETE FROM SPECIFIC S

```
int ctr=1, pos=3

while (ctr<pos) {
        temp = temp->next;
        ctr++;
}
temp->prev->next = temp->next;
temp->next->prev = temp->prev;

if (temp->next == head) {
        tail = temp->prev;
        free (temp);
} else {
        free (temp);
}
```

"STACK"



→ Linear Data Structure.
→ Limited acess is possible.
→ It is ordered list.

RULE:- Push, Pop

- Insertion & Deletion only possible from one end.
- LIFO (Last In First Out).

→ Collection of similar Datatype.
→ Fundamental Operation Push(x), Pop().
→ Other Operation

- Peek() / top() → it will only return the topmost element, but not remove it.
- Is Empty()
- Is Full()

→ Push() & Pop() only takes O(1) time complexity
→ Can implement stack using dynamic or static memory allocation.

APPLICATIONS:-

→ Reverse
→ Undo Mechanism
→ Recursion / Function call
→ many others

IMPLEMENTATION OF STAK
"ARRAYS"

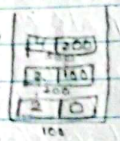
```
#define N 5;
  int stack[N];
  int top= -1;

void Push()
{   int x;
    printf("Enter data");
    scanf("%d", &data);
    if(top== N-1){
        printf("Overflow");
    } else {
        top++;
        Stack[top]= x;
    }
}

void pop(){
    int item;
    if(top==-1){
        print("Underflow");
    else {
        item= stack[top];
        top--;
    }
    printf("%d", item);

void peek(){
    int item;
    if(top==-1){
        print("Empty");
    }
    else {
        printf("%d", stack[top]);
    }
}

void display(){ → print from top to bottom
    int i;
    for(int i=top; i<=0; i--){
        pr_("%d", stack[i]);
    }
```

ismein hum reverse linked list banaye yaani 2nd node ke andar 1st ka address store hoga

IMPLEMENTATION OF STACK
"LINKED LIST"

```
struct node{
    int data;
    struct node *link;
}
struct node * top = 0;
```

top →




```
void push(){
    int x;
    scanf("%d", &x);
    struct node *newnode = malloc(__);
    newnode -> data = x;
    ~~newnode -> link = 0;~~
    newnode -> link = top;
    top = newnode;
}
```

top

```
void pop (){
    struct node* temp = top;
    top = top -> link;
    free (temp);
}
```

```
void display(){
    struct node *temp = top;
    if (top == 0){
        printf(Empty);
    }else{
        while (temp != 0){
            printf("%d", temp->data);
            temp = temp->link;
        }
    }
}
```

```
void peek(){
    if (top == 0){
        printf("Empty");
    }else{
        printf("top element is %d", top->data);
```

→ Stack with linklist will only be implemented when the size of stack is unknown.

"INFIX ~~PREFIX~~ POSTFIX"

"An infix expression is a mathematical or arithmetic expression in which operators are placed between operands."

eg: 4 + 8 , a+b*c , 6×5÷8+2

operands operator

→ These expressions are evaluted by precedence of operators & associativity.

"Prefix also known as polished notations is a mathematical expression in which operators are placed before their operands. They eliminate the need for parenthesis to indicate order of ~~precedence~~ operations

Prefix is used for LISP programming
compiler construction.

"Postfix expression is a mathematical expression in which operators come after operands.

eg: AB + (postfix) = A+B (infix) , ABC/+ = A+B/C

→ The order of evaluation of operator is left to right with no brackets in the expression to change the order.

INFIX

A+B/C*D - E/(F+G) → A+B/C*D - E/(FG+)

A+BC/*D - E/(FG+)

A+BC/D* - E/(FG+)

A+BC/D* - EFG+/

ABC/D*+ - EFG+/

ABC/D*+EFG+/-

yahe par + phele hai to multiply / se phele FG + hoge phir / hai to jo F+G hoga wo / hoga E/ se phir isko hoga use phele uvado se. Second mein BC / hoge usko bond BC ka ans * by D hoga phir + by A.

The expression will be scanned from L to Right & when as we will encounter an operator, we will apply it on last 2 operands.

"ALGORITHM TO CONVER INFIX EXP. TO POSTFIX USING "STACK" "

- Scan the symbols of the expression from L to R & for each symbol do the following:

a) If symbol is an operator/d (8,5,6)
→ Print that symbol onto the screen.

b) If symbol is a left parenthesis
→ Push it on the stack.

c) If symbol is a right parenthesis
→ Pop all the operators from the stack upto the first left parenthesis & print them on the screen.
→ Discard the L & R parenthesis

d) If symbol is an operator
→ If the precedence of the operators in the stack is greater than or equal to the current operator, then
↳ Pop the operator out of the stack & print them on screen, & push the current oper. onto the stack.
→ else
↳ Push the current operator onto the stack

eg: 8*(5^4+2)-6^2/(9*3)

Output: 854^2+-62^93*/-

"EVALUATION"

- Scan the symbols of the given postfix expression from L to R & for each symbol, do the following.

a) If symbol is an operant
→ Push it on the stack

b) If symbol is an operator
→ Pop two symbol out of stack & apply the operator on these symbols.
→ Push the result on to the stack

c) After scanning all the symbols of the postfix exp., pop the remaing symbol out of stack & print on screen. The remaing symbol is the result of the postfix exp.

NESO

" PROGRAM FOR CONVERTIN INTOPOST "

```
char stack[100];
char infix[100], postfix[100];
int top=-1;

int main(){

    cout<< "Enter the infix exp.";
    cin>> infix;
    infixtopostfix (infix);
}

void infixTopostfix (){

    for(int i=0; i != '\0'; i++){
        if(infix[i]>=0 || infix[i] <= 9)
            postfix[j++] = infix[i]; //storing the postfix exp.

    }else if (infix[i]== '(' ){
            top++;
            stack[top] = infix[i];
    }else if (infix[i]== ')' ){
        for(i=top; i>=0; i--){
            if(stack[i]== '(' ){
                top--;
                break;
            }
        else {
            postfix[j++] = stack[top];
            top--;
        }

    else if (infix[i]== '+' || infix[i]== '-' ||
        infix[i]== '*' || infix[i]== '/' || '^' ){

        while (!isEmpty() && precedence (stack[top])>=precedence (infix[i]))
        {
            postfix[j++] = stack[top];
            top--;
        }
        top++;
        stack[top] = infix[i]; } push (infix[i]);
    }
}

while (! isEmpty() ){
    postfix[j++] = stack[top];
    top --;
}

postfix[j] = '\0';
}
```

```
int precedence (char symbol)
{
    switch (symbol)
    {
    // Higher value means greater precedence
    case '^':
            return 3;
    case '/':
    case '*':
            return 2;

    case '+';
    case '-';
            return 1;
    default:
            return 0;
    }
}
```

PROGRAM FOR EVALUATING POST

```
int Evalpo (string post){

    for (int i=0; i<strlen(post); i++){
        char symbol = post[i];
        if (symbol>='0' && symbol<='9')
        {

            push(symbol-'0');
        }
        else
        {   a=pop(); b=pop();
            switch(symbol){
            case '+':
                push(b+a); break;
            case '-':
                push(b-a); break;
            case '*':
                "
            case '/':
                "
            case '^':
                "
            }
        }
    }
    return pop();
```

→ there will be only one left which will be ans

Infix To Pre-fix Conversion

A + B*C ⇒ A + *BC ⇒ +A*BC

→ Phele expression ko reverse krenge.
→ Agr koi operand hoi to print krdenge.
→ Agr operator hoi.
  ↳ If stack is empty push that operator ~~to stack~~
  ↳ else,
    ↳ If the precedence of current operator is less than topmost operator in stack then we will popout the stack operator. Then again check the precedence of current & top most operator in stack, will do until both have same precedence or the current has a greater than topmost.

    ↳ If both have same preceden check the associtivity if it's left to right simply push it into the stack.

    ↳ If current has greater less precedence than also push it.

    ↳ If operator is closing bracket push it.
    ↳ If " " opening b " pop the operators until you reach closing brackets.
  If expression is over than simply popout the operators.

K+L-M*N+(O^P)*W/U/V*T+Q
Q+T*V/U(W*)P^O(+N*M-L+K

Input	Stack	Postfix
Q		Q
+	+	Q
T	+	QT
*	+*	QT
V	+*	QTV
/	+*/	QTV
U		QTVU
/	+*//	QTVU
W	+*//	QTVUW
*	+*//*	QTVUW
)	+*//)	QTVUW
P	+*//)	QTVUWP
^	+*//)^	QT "
O	+*//)^	QTVUWPO
(	+*//*	QTVUWPO^
+	+*//*	QTVUWPO^
N	+*+	QTVUWPO^*//
*	++*	QTVUWPO^*//*N
M	++*	NM
	++-	NM
	++-	NM*
	+-+	"
	-+	" " K

ab.: QTVUWPON * // * NM * LK + - + +

ab final answer isko reverse karke likhenge.

+ + - + KL * MN * // * NOPW UVTQ

⇒ PREFIX TO POST:

Reverse the prefix expression.

① If an operand push ~~an~~ in stack.

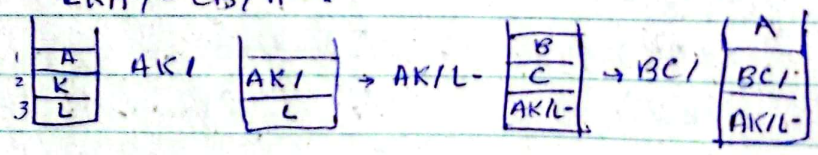
② If an operator pop the last two operands then operand1 + operand2 + operator

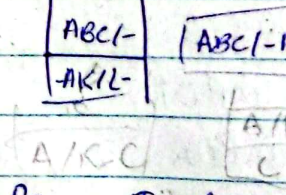
④ then again push this expression into stack

⑤ When expression is finish pop the last element in stack which will be our final postfix exp

* - A/BC - /AKL

LKA/ - CB/A - *





A/KC

Postfix To Prefix.

⇒ operator + operand2 + oper1

⇒ PREFIX TO INFIX

Reverse the exp.

same as above but jab operator aye to oper1 + oper2 + operator

sabko combine karke push karenge.