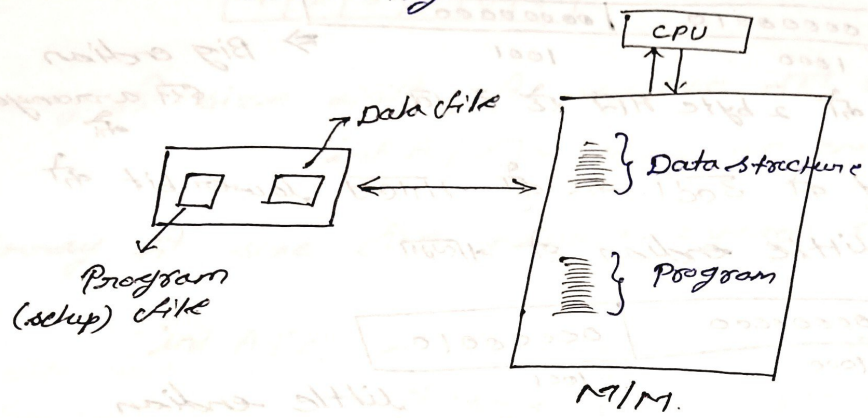
Data Structure

Data structure can be defined as arrangement of collection of data (in main memory/RAM) so that they can utilise efficiently



1.) Arrangement of data
2) operation on data

DATA STRUCTURE

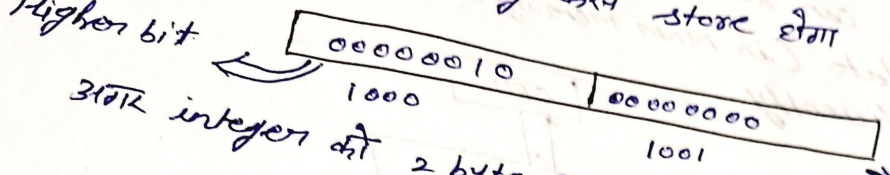
- Primitive DS
  - int
  - char
  - float
  - double
  - pointer
- Non-primitive DS
  - Linear DS
    - Array
    - stack
    - queue
    - linked list
  - Non-linear DS
    - Graph
    - Tree

operation on DS

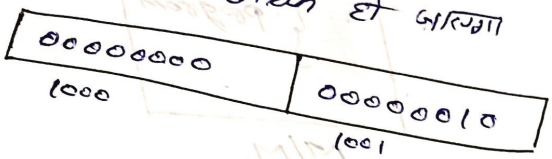
1) Traversing
2) Searching
3) Insertion
4) Deletion
5) Sorting
6) Merging

X → 512 (1000) int X = 512

Memory में actually कैसे store होगा



अगर integer को 2 byte मान रहे हैं तब ⇒ Big endian
arrangement को उल्टा कर दी मतलब lower bit को पहले तब
little endian हो जाएगा



little endian

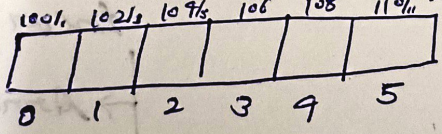
इस arrangement को

Array

An array is a collection of data items, all of the same type, accessed using a common name

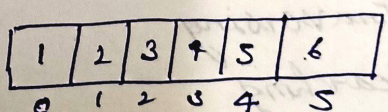
```
Datatype ArrayName [size];
int A[6];
```
} Array Declaration

index हमेशा 0 से start होता है.

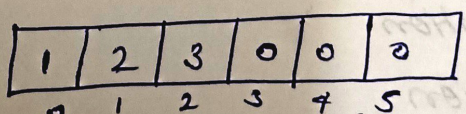


Array Initialisation

int A[6] = {1,2,3,4,5,6}; A

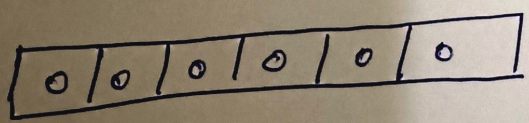


int A[6] = {1,2,3}; A



बाकी बचे हुए element zero से initialise होंगे Garbage value नहीं store होगा।

int A[6] = {0}; A



int A[7] = {1, 2, 3, 4, 5, 6}; जैसे ही A array को initialize कर सकते Array का size 6 है और store होगा।

1	2	3	4	5	6

* Array की size कभी भी negative नहीं हो सकती।

int A[-6] ✗ throw an error

Array की size हमेशा positive integer होगी।

int A[6]; ⇒ एक array size 6 का बन गया उसमें garbage value store है।

```
for(int i=0; i<6; i++){
    Printf("%d", A[i]);
        or
    Printf("%d", i[A]);
        or
    Printf("%d", *(A+i));
}
```

तीनों method same है।

Important

Little-endian and Big-endian :-

Endianness is a term that describes the order in which a sequence of bytes is stored in computer memory.

Big-endian is an order in which the "big end" (most significant value in the sequence) is stored first, at the lowest storage address

Little-endian is an order in which the "little end" (least significant value in the sequence) is stored first

A hexadecimal number A45DF7BC is stored from memory address 5007H in little endian format. The byte stored in memory location 5007H, 5008H, 5009H and 500AH will be.

Sol.

BC 5007H
F7 5008H
5D 5009H
A4 500AH

Q If the numerical value of a 2-byte unsigned integer on a little endian computer is 255 more than that on a big endian computer, which of the following choices represents the unsigned integer on a little endian computer?

A) 0X0001

B) 0X6665 ✓

C) 0X0100

D) 0X4243

Sol बस जहाँ bit different है बस वहीं देख लो उसका difference

2048 1024 512 256 128 64 32 16 8 4 2 1

0000 0000 0000 0001 → little endian ⇒ 1

0000 0001 0000 0000 ⇒ 256

option A ✗

0110 0110 0110 0101 → little endian
512 1 → 513

0110 0101 0110 0110 → Big endian
256 2 → 258

$$\begin{array}{r} 513 \\ -258 \\ \hline 255 \end{array}$$

So, option B ✓

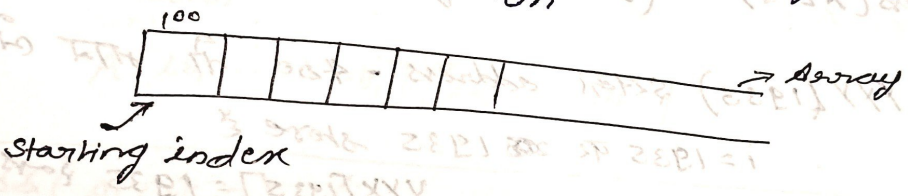
0000 0001 0000 0000 → 256
0000 0000 0000 0001 → 1 } 25 =

So, option C ✓

0100 0010 0100 0011 → little endian
0100 0011 0100 0010 → 256

little endian होता है
So, option D ✗

* Array :- Random access
  ↳ Possible due to contiguous memory allocation



*
$$\text{Address}(A[i]) = (i - \text{starting index}) \times w + \text{Base Address of an Array}$$
(w → size of an element)

starting index को lower bound भी बोलते हैं और last index को upper bound

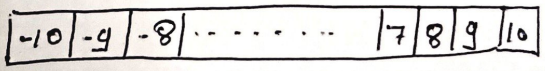
$$\text{Address}(A[i]) = (i - lb) \times w + BA$$
(lb ⇓ lower bound; BA → Base Address)

Question में Base address mention नहीं है तो zero को लेते हैं same applicable for starting index

Important point

XXX(-10 ..... 10) इसका length निकालना है
यहाँ पर size नहीं

-10 .... 10 तक element present है array में



तो length = 10 - (-10) + 1 = 21
⇓
last element - First element + 1

Q Consider the linear array XXX(-10:...10), YYY(1935...1985), ZZZ(35)

i) No. of element in each array?

ii) Base(YYY) = 400 & w = 4 word per memory cell for YYY then address of YYY[1942], YYY[1977] & YYY[1988]

Sol" i) length XXX = 10-(-10)+1 = 21

length YYY = 1985-1935+1 = 51

length ZZZ = 35

ii) address(A[i]) = (i - starting index) × w + BA

YYY[1935) इसका address = 400 तो मान लेते हैं

i = 1935 पर 1935 store है.

YYY[1935] = 1935 {assumption}

ये सही मानना है तो (i - starting index) क्या देता है कितना cell move करना पड़ा Base address के cell से।

∴ YYY[1942] = (1942 - 1935) × 4 + 400

= 428

YYY[1977] = (1977-1935) × 4 + 400

= 568

YYY[1988] = (1988 - 1935) × 4 + 400

= 212 + 400

= 612 ✗

⇒ 1988 element ही नहीं है array का

2-D Array

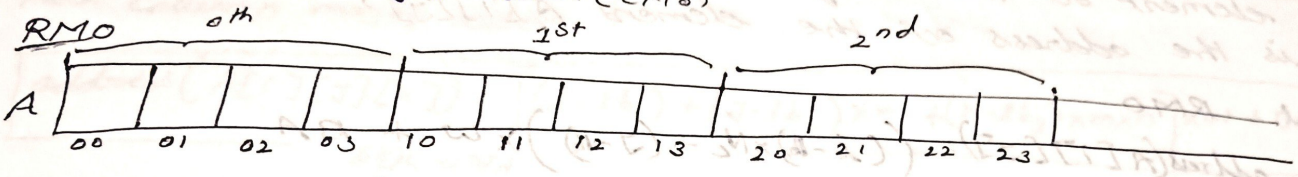
int A[3][4]
↓ Row ↓ column

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

Memory में 2d कैसे रहता है

1.) Row mayor order (RMO)
2.) Column mayor order (CMO)

RMO



N_R :- No. of Row

N_C : No. of Column

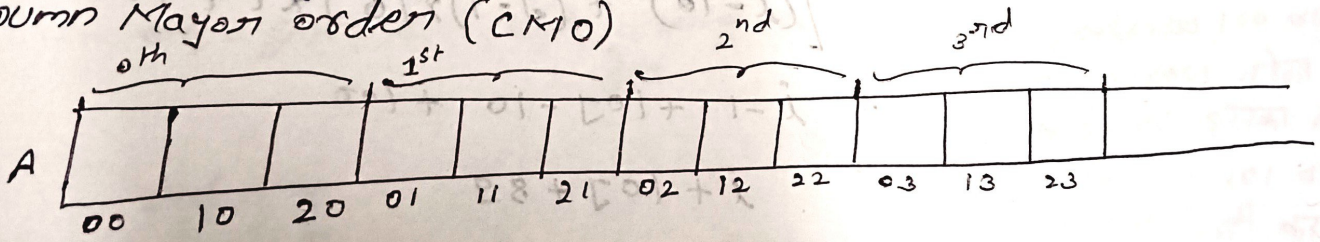
$$A[lb_1 \ldots\ldots ub_1][lb_2 \ldots\ldots ub_2]$$

$$\boxed{Address(A[i][j]) = ((i - lb_1) \times N_C + (j - lb_2)) \times w + BA}$$

N_C ↓ No. of Column; BA ↓ Base address

↓ For Row mayor order

Column Mayor order (CMO)



$$A[lb_1 \ldots ub_1][lb_2 \ldots ub_2]$$

$$\boxed{Address(A[i][j]) = ((j - lb_2) \times N_R + (i - lb_1)) \times w + BA}$$

↓ For Column mayor order

@ Let A be a two-dimensional array stored as follows

A: array[1.....10][1.....15] of integer (GATE-1998)

Assuming that each integer takes one memory location. The array is stored in row major order and the first element of the array is stored at location 100, what is the address of the element A[i][j]?

Sol RMO

$$address(A[i][j]) = ((i-lb)\times N_c + (j-lb))\times w + BA$$

one memory location = 1 byte

$$= [(i-1)\times 15 + (j-1)]\times 1 + 100$$

$$= 15i + j - 16 + 100$$

$$= 15i + j + 84$$

From Colowmn mayor

$$Address(A[i][j]) = [(i-lb_1) + (j-lb_1)\times N_R]\times w + BA$$

$$= [(i-10) + (j-1)\times 10]\times 1 + 100$$

$$= i - 1 + 10j - 10 + 100$$

$$= i + 10j + 89$$

2-D Array

⇓

इसको बोल सकते है collection of 2-D array

A[ ][ ][ ]

2D array

कितने 2D array है वह ये बतायेगा।

इसको reverse भी मान सकते है।

$A[lb_1 \dots ub_1][lb_2 \dots ub_2][lb_3 \dots ub_3]$

(m = ub_1 side, n = ub_2 side, p = ub_3 side)

For Row major order

$$address(A[i][j][k]) = ((i-lb_1)\times n\times p + (j-lb_2)\times p + (k-lb_3))\times w + BA$$

(row में इधर से चलते है)

For Column major order

$$address(A[i][j][k]) = [(i-lb_1) + (j-lb_2)\times m + (k-lb_3)\times m\times n]\times w + BA$$

(इधर चलते है column में)

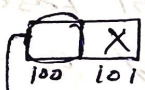
Pointers and Arrays

`int *q;` ⇒ ऐसे, pointer को declare करते है

`*` ⇓ De-referencing operator

```
int x = 10;
Char *q;
*q = &x;
```

} Error show नही करेगा but wrong answer देगा



q से q बस address 100 वाले को access करेगा because इसका type Char है। 101 वाले को access नही कर पाएगा।

`int *q`

q ⇓ address है

→ address के अंदर जाओ

→ कितना byte read करना है

Previous example में int को 2 byte माने है तो

`Char *q;` ⇒ इसका मतलब address q के अंदर जा के 1 byte read करो। Char 1 byte का होता है।

* Variable का और pointer का data type same रखते है। नही तो uncompatible का issue होगा।

array name ⇒ Base address of array

*a = a[0]

दोनो एक ही चीज है

%p ⇒ address print करने का format specifier है।

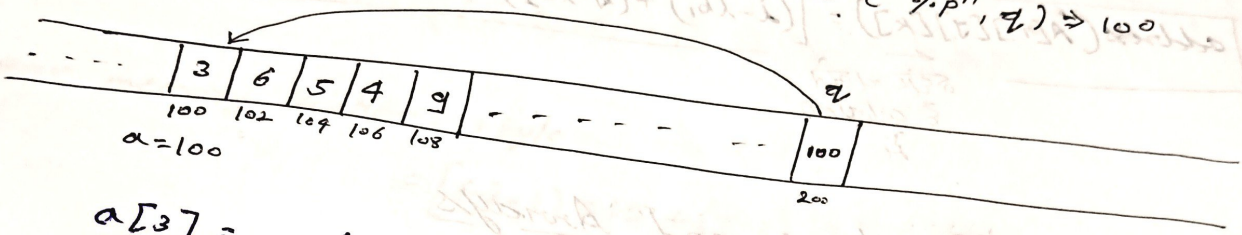
int a[5] = {3, 6, 5, 4, 9}

int *q;

q = a;

printf("%p", a) ⇒ 100

printf("%p", q) ⇒ 100



a[3] = *(a + 3*2)

= *(106)

* Array का base address कभी change नहीं होता इसलिए array पर operation perform नहीं करते

a = a+1; ✗ a = a*4; ✗

but pointer पर कर सकते है और इसलिए pointer की जरूरत पड़ी

q = q+1; इसका मतलब है q से 1 block आगे, इसका address अब q में store होगा

block का size = 2 byte है

q = 100+1*2

= 102

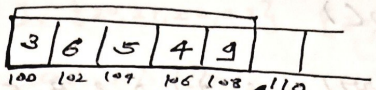
102
q

* Pointer की value को array को assign नहीं कर सकते because Base address of pointer change हो जाएगा but pointer को base address of array assign कर सकते है.

a = q; ✗ q = a; ✓

** $a[3] = q[3] = *(\&a[3])$ q = pointer है

$a+1 = 100 + 1*2$
$= 102$



** $\&a+1 = 110$

इसका मतलब है पूरा array को cross कर दो फिर जो next element है वो।

$\&a+1 = 110$
$\&a+2 = 100 + 2*10$
$= 120$

* simple सा बात है array की base address update नहीं होना चाहिए $a+1$ ✓ $a = a+1$ ✗
$q = a+1$; ये statement सही है।

$$\&a+x = a + x * \text{size of array}$$

**
```
int main(){
    int a[] = {1,2,3,4,5};
    Printf("%p\n", &a);
    Printf("%p\n", &a+1);
    printf("%p\n", &a+2);
}
```
} example है!!

output: 0x7ffc16fa4c60] 20
0x7ffc16fa4c74]
0x7ffc16fa4c88]

Q. output = ?

```
#include <stdio.h>
int main()
{
    int arr[] = {1,2,3,4,5,6,7,8,9,0,1,2,5};
    int *p = arr+4;
    Printf("%d", P[1]);
    return 0;
}
```

sol. arr+4 = a[4] = 5 की address है।

P[1] = *(P+1) = *(108+2)
= *(110)
= 6

1	2	3	4	5	6
100	102	104	106	108	110

output = 6

Q

```
#include <stdio.h>
int main()
{
    int a[] = {2,4,6,8,10};
    int i, sum=0, *b = a+4;
    for(i=0; i<5; i++)
        sum = sum + (*b-i) - *(b-i);
    Printf("%d", sum);
    return 0;
}
```

Gate-2019

output = 10

100	102	104	106	108
2	4	6	8	10
0	1	2	3	4

a+4 = a[4] = 10

*b = 10

*(b-0) = 10

Sol,

sum = (10-0) + (10-1) + (10-2) + (10-3) + (10-4) -
{10+8+6+4+2}

= 10 + 9 + 8 + 7 + 6 - 10 - 8 - 6 - 4 - 2

16 - 6 = 10

Pointer & 2D array

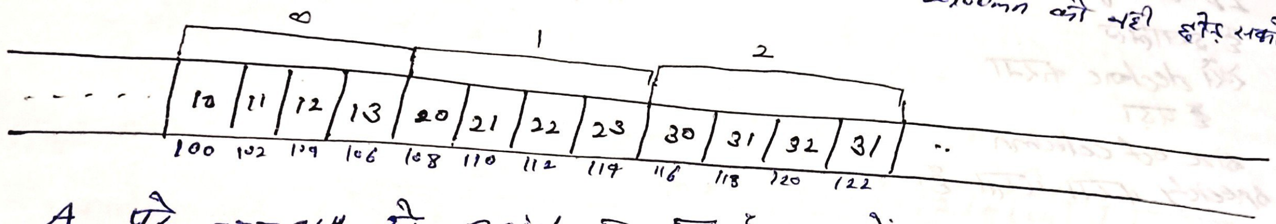
int A[3][4] = { {10, 11, 12, 13}, {20, 21, 22, 23}, {30, 31, 32, 33} }

⇓

इसको ऐसे भी लिख सकते हैं int A[ ][4]

↳ ये अपने आप set हो जाएगा but column को नहीं छोड़ सकते।

RMO

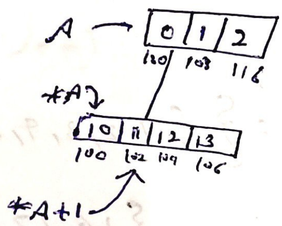


A पूरे array को point कर रहा है A में store है address 0, 1, 2 इन तीनो block का

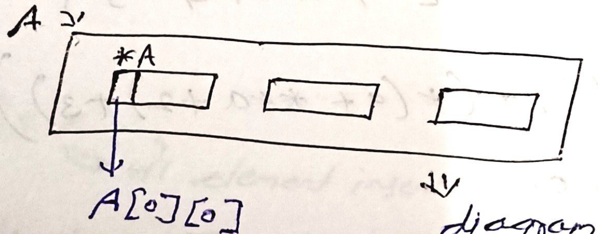
A = 100

*A = 100

& A[0][0] = 100



* A का मतलब A में जो store है उसका ~~address~~ value तो A में 3 address store है तो पहला वाला address point कर देगा।



⇓

diagram to understand 2D array

A+1 = 108

*A+1 = 102

$A[2][3] = *(*(A+1) + 3)$

**(P+1) = 20

int *P = A; ✗ 2D-array के case में ऐसे pointer नहीं declare कर सकते।

* int(*P)[4]   P is pointer to an array of 4 integer

int *P[4]   P is an array of 4 integer pointer

int(*P)[4] = A;

P = 100

P+1 = 108

P+2 = 116

*P+1 = 102

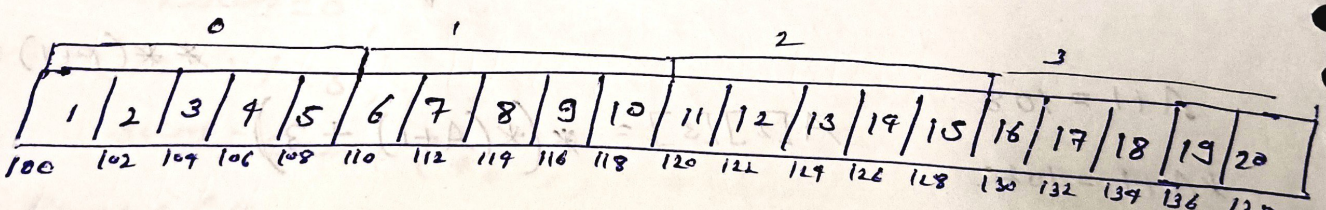
* P[1][2] = *(P[1]+2) = *(*(P+1)+2)

int(*P)[] ⇒ यहाँ bracket लगाने की जरूरत इसलिए पड़ी because [] is ki priority थोड़ा ई पहले से evaluate हो जाएगा * से.

2D-array ई इसलिए ऐसे declare करना ई पड़ा size of column specify करना पड़ता है।

Q
```
#include <stdio.h>
int main()
{
  int a[4][5] = { {1,2,3,4,5}, {6,7,8,9,10}, {11,12,13,14,15},
                  {16,17,18,19,20} }
  Printf("%d", *(*(a+**a+2)+3))
  return 0;
}
```

Sol.



*(a+**a+2) = ~~*(100+1+2)~~ : ~~*(104) = 3~~

~~*6~~ *(a+3) = 130 ⇒ a[3]

Now ***(a[3]+3) = 19**

*(130+3*2) = *(130+6)
= *(136)
= 19

Array operation

Traversing:-

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

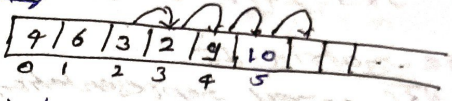
⇒ & लगाना important है

Insertion:

$0 \le index < n$ → size of array / no. of element in array

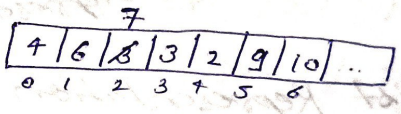
Insertion at particular "index"

int A[20] n=4



index = 5
insert 10
A[5] = 10
n = 5

index = 2
insert 7
इसके लिए A[2] से आगे element को right shift करना होगा.




```
for(int i=n; i>index; i--)
{
    A[i] = A[i-1];
}
n++;
A[index] = x;
```

⇒ जो element insert करना है

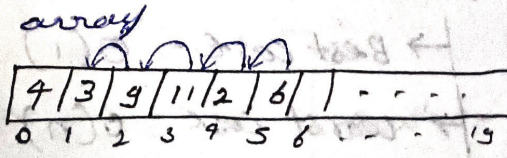
Time Complexity:- Insertion — Best case O(1)
— worst case O(n)

Deletion:

index पहली बात insertion और deletion के लिए valid होने चाहिए index check कर लो पहले

$0 \le index < n$ → size of array / no. of element in array

int A[20]; n = 6




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

Time Complexity:
→ Best case O(1)
→ worst case O(n)
(n-1) shift

Sparse Matrix

ऐसा matrix जिसके maximum element zero हो।

Array Representation

	0	1	2	3	4	5	6	7	8
row	0	1	1	2	4	4	5	6	6
col	4	2	6	0	1	5	7	2	4
value	5	4	6	1	8	3	2	10	7

$3 \times 9 = 27 \times 2 = 54$ byte

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

$7 \times 8 = 56 \times 2 = 112$ byte

7×8

integer को 2 byte मानते हैं।

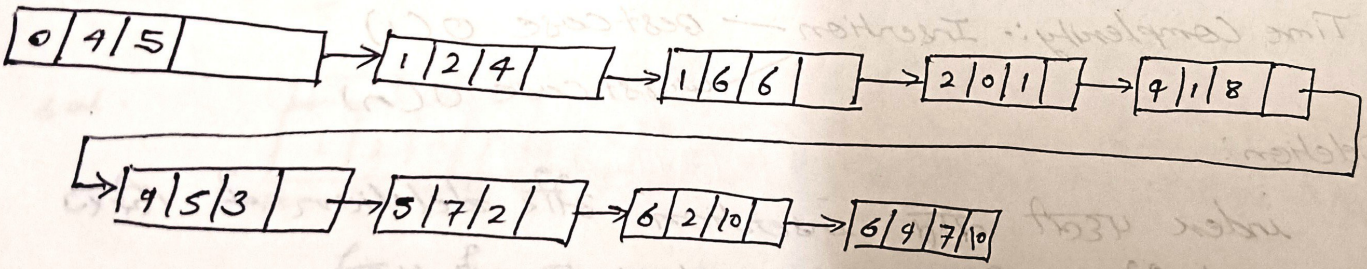
sparse matrix को इस NXE array representation method में store करते हैं।

Sparse Matrix

- Array representation
- Linked representation

Linked List Representation

row	col	value	next Address



→ Linear search

- Best case - $O(1)$
- Worst case - $O(n)$
- Average case - $O(n)$

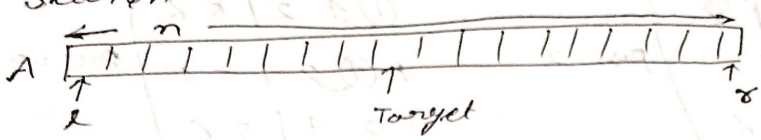
Average case निकालने के लिए सारे case का time निकाल लो divide No. of case.

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

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

→ Recursive relation & linear search का

→ Binary search :- → Half Interval search → Binary search का दूसरा नाम Logarithmic search भी बोलते है।



Algorithm :-

```
int BS(A, n, Target){
    int l=0, r=n-1, mid;
while(l<=r){
    mid = (l+r)/2;
    if (A[mid] == Target)
        return mid;
    else if (A[mid] < Target)
        l = mid+1;
    else
        r = mid-1;
}
return -1;
}
```

$n \Rightarrow n/2 \Rightarrow n/2^2 \Rightarrow \dots \dots \frac{n}{2^k}$

$\frac{n}{2^k} = 1$

$n = 2^k$

$2^k = n$

$k = \log_2 n$

$O(\log_2 n)$ Time Complexity

→ for(i=n; i>=1; i=i/2)

→ for(i=1; i<=n; i=i*2)

Time complexity $O(\log_2 n)$

Binary search or Half Interval search or Logarithmic search

Binary search Efficient होता है but array sorted होनी चाहिए।

Sorting

→ Bubble Sort

$n = 5$

(4)	1	5	6	2	
1	(4)	5	6	2	
1	4	(5)	6	2	
1	4	5	(6)	2	
1	4	5	2	(6)	Pass = 1

1st pass में सबसे largest number end में चला जाएगा।।

① 4 5 2 | 6
1 ④ 5 2 | 6
1 4 ⑤ 2 | 6
1 4 2 5 | 6
Pass = 2

① 4 2 | 5 6
1 ④ 2 | 5 6
1 2 4 | 5 6
Pass = 3

① 2 | 4 5 6
1 ② | 4 5 6
Pass = 4

① 2 4 5 6

n = No. of element
then **total no. of pass = n-1**

```
Bubble Sort(A, n) {
    for (i=0; i<n-1; i++)
    {
        for (j=0; j<n-1-i; j++)
        {
            if (A[j] > A[j+1])
            {
                temp = A[j];
                A[j] = A[j+1];
                A[j+1] = temp;
            }
        }
    }
}
```

Bubble sort simple
First element को
पकड़ो और
compare करते जाओ
swap करते जाओ
First iteration
में max. element
last position
पर
फिर 3 वाली दौड़
दो बाकी पर
element पर
same operation

* Bubble sort की time complexity worst तब होगी जब descending order में array होगा।

तब

$(n-1)+(n-1) + (n-2)+(n-2) + \cdots\cdots 2+2+1+1$

$= (n-1)(n-\cdots)$

$= n^2 - n$

Pass = 1 पर
(n-1) swap
(n-1) comparision

Pass = 2 पर
(n-2) swap
(n-2) comparision

worst case = n^2-n → इतने comparision और swapping होंगे

Time complexity → worst case = $O(n^2)$

Best case में क्या होगा कि swapping नहीं होगी comparision तो होगा ही

Array → sorted in descending order → worst case → n^2-n comparision & swapping → Time complexity $O(n^2)$

→ sorted in ascending order → Best case → $\frac{n^2-n}{2}$ swapping → Time complexity $O(n^2)$

$n-1 + n-2 + n-3 + \cdots 2+1$

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

→ code में थोड़ा modification कर दें तो best case में Time complexity $O(n)$ हो जाएगी only $(n-1)$ comparision करने होंगे.

अगर किसी iteration में swapping नहीं हो रही है तब loop से break कर जाओ।

Modified Bubble sort Algorithm:-

```
Bubble sort(A, n) {
    for(int i=0; i<n-1; i++) {
        int flag=0;
        for(int j=0; j<n-i-1; j++) {
            if (A[j] > A[j+1])
            {
                temp = A[j];
                A[j] = A[j+1];
                A[j+1] = A[j];
                flag=1;
            }
        if (flag==0)
            break;
    }
}
```

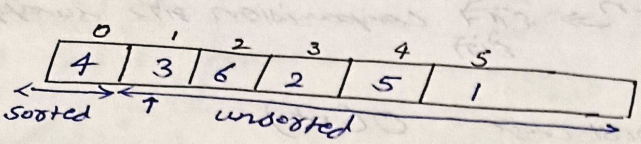
* कोई Bubble sort दीजे तो modified ही लेना है।

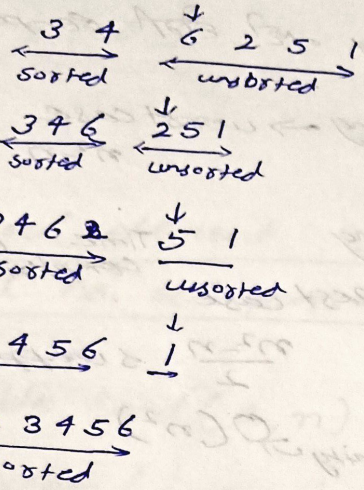
Time complexity :- $O(n^2)$

space complexity $O(1)$

Time complexity
- Best case $O(n)$
- worst case $O(n^2)$
- Average case $O(n^2)$

Insertion Sort





हर element को pick कर रहे है उसको उसकी exact position पर रख देंगे।

Algorithm:-

```
for(int j=1; j<n; j++){
    key = A[j];
    i = j-1;
    while(i>=0 && A[i]>key)
    {
        A[i+1] = A[i]
        i = i-1;
    }
    A[i+1] = key;
}
```

→ Time complexity

↳ worst case

जब array sorted है descending order

5, 4, 3, 2, 1

⇓

No. of comparison = 10

No. of swapping = 10

$n, n-1, \dots\dots 1$

No. of comparision ← 1 + 1 + 2 + 2 + 3 + 3 + ------ $n-1 + n-1 = n(n-1)$ → No. of swapping

$= n^2 - n$

j=1 j=2 j=3

Time complexity : $O(n^2)$

Best case

⇒ जब array sorted है ascending order में

⇓

(n-1) comparision

तब Time complexity $O(n)$

लगेंगे बस

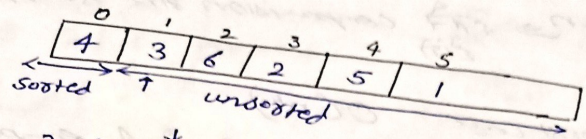
Total (n-1) comparisons ⇐ { j=1 ⇒ 1 comparision, j=2 ⇒ 1 comparision, ⋮, j=n-1 ⇒ 1 comparision

↳ Best case - $O(n)$

↳ worst case - $O(n^2)$

No swapping ⇐

→ Insertion Sort



3 4 | 6 2 5 1
sorted | unsorted

3 4 6 | 2 5 1
sorted | unsorted

2 3 4 6 | 5 1
sorted | unsorted

2 3 4 5 6 | 1

1 2 3 4 5 6
sorted

हर element को pick कर रहे हैं उसको उसकी exact position पर रख देंगे।

Algorithm:-

```
for(int j=1; j<n; j++){
    key = A[j];
    i = j-1;
    while(i>=0 && A[i]>key)
    {
        A[i+1] = A[i]
        i = i-1;
    }
    A[i+1] = key;
}
```

→ Time complexity
- worst case
  जब array sorted है descending order
  5, 4, 3, 2, 1
  ⇓
  No. of comparison = 10
  No. of swapping = 10

n, n-1, ----- 1.

No. of comparision ← 1+1 + 2+2 + 3+3 + ----- n-1 + n-1 = n(n-1) → No. of swapping
j=1, j=2, j=3

$= n^2 - n$

Time complexity : $O(n^2)$

Best case
⇒ जब array sorted है ascending order में
⇓
(n-1) comparision लगेंगे बस

Time complexity $O(n)$

- Best case - $O(n)$
- worst case - $O(n^2)$

No swapping ⇐ Total (n-1) comparision ⇐
- j=1 ⇒ 1 comparision
- j=2 ⇒ 1 comparision
- ⋮
- j=n-1 ⇒ 1 comparision

Q The usual $\Theta(n^2)$ implementation of insertion sort to sort an array uses linear search to identify the position where an element is to be inserted into the already sorted part of the array. If instead, we use binary search to identify the position the worst case running time will

a) remain $\Theta(n^2)$
b) become $\Theta(n(\log n)^2)$
c) become $\Theta(n \log n)$
d) become $\Theta(n)$

sorting
/ \
Comparision ↓↓ 1 element का index find करने के लिए comparision nlogn n के लिए nlogn

swapping ↓↓ element को उसकी जगह insert करना है array में तो insertion के at particular index का Time तो वही रहेगा $\Theta(n^2)$ ← सारे element को तो swap करना पड़ेगा।

worst case = $\Theta(n \log n) + \Theta(n^2) \Rightarrow \Theta(n^2)$

suppose $i = 5$ पर insert करना है तो $j = 6$ पर element को swap करना होगा जगह बनाने के लिए

→ selection sort
↳ just reverse of Bubble Sort

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

4 3 6 2 5 1
↑ ↑
i j

min = i = 0 ⇒ i = 0 से minimum कोई j मिला तो min को update कर देंगे ⇒

j = 3 जब होगा तब min = j फिर compare करेंगे

→ unsorted array में से minimum element निकाल रहे है हर बार

1 3 6 2 5 4] pass 1
↑ ↑
i j

1 2 6 3 5 4] pass 2
sorted unsorted

1 2 3 6 5 4] pass 3

1 2 3 4 5 6] pass 4

1 2 3 4 5 6] pass 5

code

```
for(int i=0; j<n-1; i++) {
    int min=i;
    for(j=i+1; j<n; j++) {
        if(A[j] < A[min])
            min=j;
    }
    swap(A[min], A[i]);
}
```

Bubble sort से यह अलग बस इसलिए है because यहाँ पर comparision तो (n-1) ही for index=n but swapping 1 होगा।

worst case: descending order में हो array

comparison $n-1 + n-2 + n-3 + \cdots + 2 + 1$

Swapping $\underbrace{1 + 1 + 1 + \cdots + 1 + 1}_{n}$

So, Time complexity: $n + n-1 + n-2 + \cdots + 2 + 1$

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

$O(n^2)$

Best case: Ascending order में array हो

यहाँ भी swapping 1 ही होगी और comparision उतना ही होगा।

So, selection sort का Best case, worst case, average case सबमें Time complexity $O(n^2)$

* code में swapping कम कर सकते हैं but इससे Time complexity पर कोई change नहीं आएगा।

Modified code (No. of swapping minimum होगा।)

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

→ First iteration में अगर swapping नहीं हुई क्या यहाँ भी Bubble sort की तरह Break कर सकते हैं Insertion sort में.

No 1 4 5 3 6

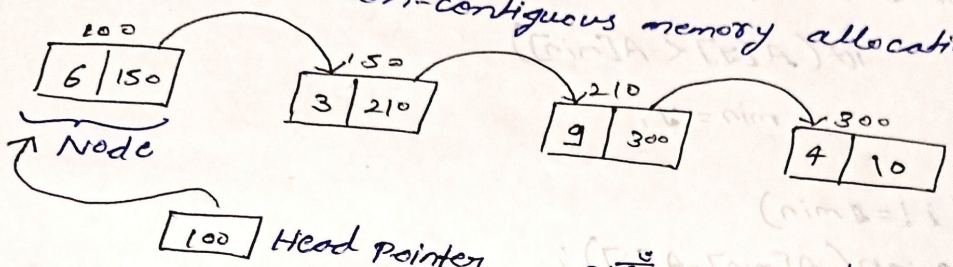
swapping नहीं हो रही इसके बस ये confirm होगा कि 1st element सबसे minimum है।

Linked List

Runtime में grow & shrink कर सकते हैं, Array में ऐसा नहीं कर सकते हैं।

size

Linked list ⇒ non-contiguous memory allocation



यहाँ पर Random access possible नहीं है Array में possible है Random access बोले तो A[3] इस तरह का।

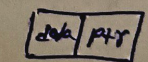
→ Linked list में sequential access होता है।

* Head pointer पर increment decrement operation नहीं perform कर सकते

Structure
⇒ user defined data type

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

⇒ pointer का data type है because ये एक Node को point कर रहा है जिसका type खुद में struct Node है

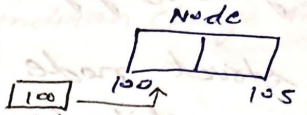


* Linked list को self-Referential data structure भी बोलते हैं

Head pointer linked list का base address store करता है।

→ malloc()
⇒ return करता void pointer मतलब ऐसा pointer जिसका कोई data type ना हो।

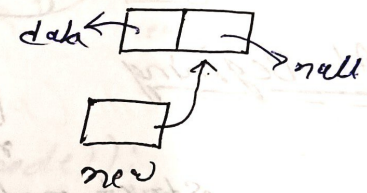
```
malloc(sizeof(struct node));
//
```



100 return करेगा address of Node (starting address)

इसलिए ~~linked list~~ को type cast कर देते हैं मतलब malloc को address दे रहा है एक struct node type का address दे रहा है।

```
new = (struct node*) malloc(sizeof(struct node));
scanf("%d", &new->data);
new->next = null
```



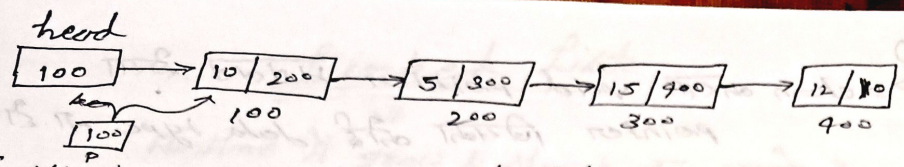
* malloc function के लिए header file होती है #include<stdlib.h>

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

First node का address store किए रहेगा।
newly created node का address है इसके पास।
Node को link करती है एक दूसरे से see else part of code

```
struct node *head, *new, *temp;
new = (struct node*) malloc(sizeof(struct node));
scanf("%d", &new->data);
new->next = NULL;
if (head == NULL)
    head = temp = new;
else
    { temp->next = new;
      temp = temp->next;
    }
```

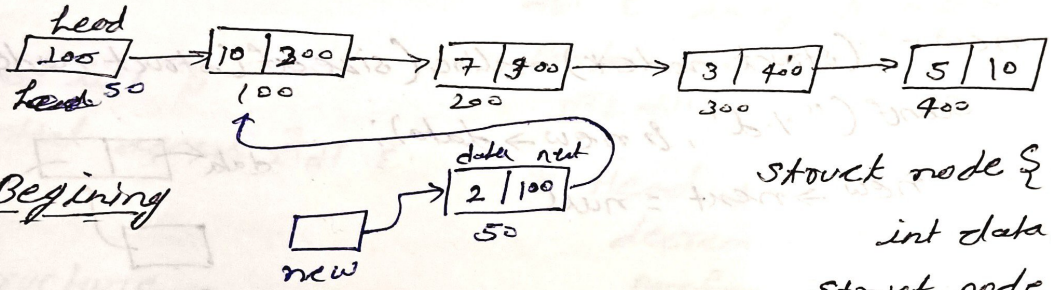
इस code को loop में चला देंगे linked list बन जाएगी।



Traversing a singly linked list :-

```
struct node *p = head;
while (p != NULL) {
    Printf("%d", p->data);
    p = p->next;
}
```

→ Insertion of a Node in a Linked list



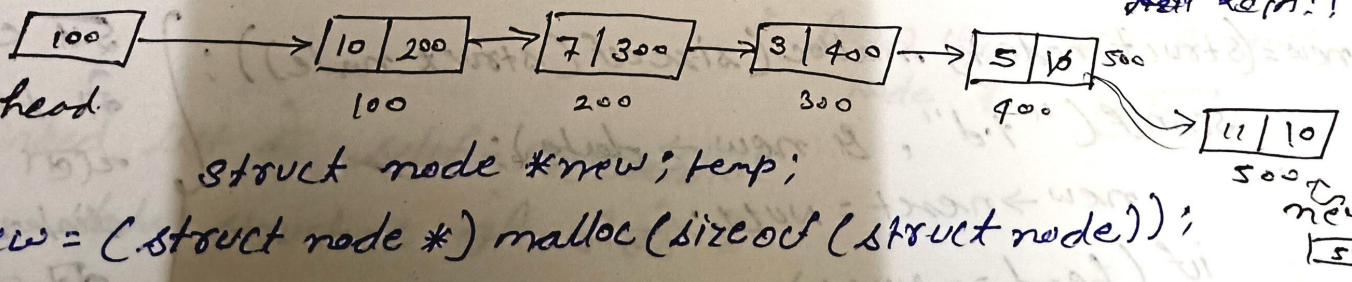
At Beginning

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

```
struct node *new;
new = (struct node *) malloc(sizeof(struct node));
scanf("%d", &new->data);
new->next = head;
head = new;
```

] If order important पहले ही head में new store होगे तो पुराना linked list का base address खो देगे!!

At last




```
struct node *new; temp;
new = (struct node *) malloc(sizeof(struct node));
scanf("%d", &new->data);
new->next = NULL;
temp = head;
```

~~while (temp != NULL) {~~
~~temp = temp -> next;~~
~~}~~

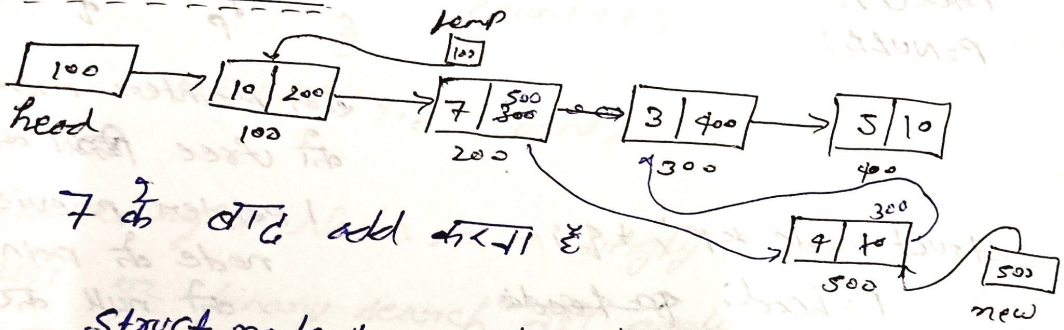
इससे तो linked list खत्म हो जायेगी तब रुकेगा

इसका मतलब while के अंदर वाला condition change करना होगा।

while (temp -> next != NULL) {
temp = temp -> next;
}

temp -> next = new;

At specific position



7 के बाद add करना है

struct node * new, temp;
new = (struct node *) malloc (sizeof (struct node));
new -> next = null;
temp = new;
while (temp -> data != 7) temp = temp -> next;
new -> next = temp -> next;
temp -> next = new;

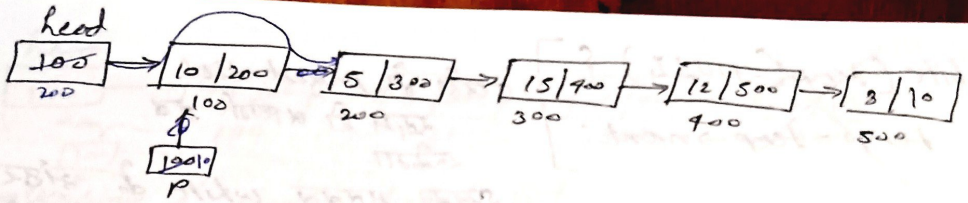
→ Deletion of a Node in a linked list

Free ( ); ⇒ यह function memory release कर देता है।

1 element ही है node में तब deletion कैसे करेंगे

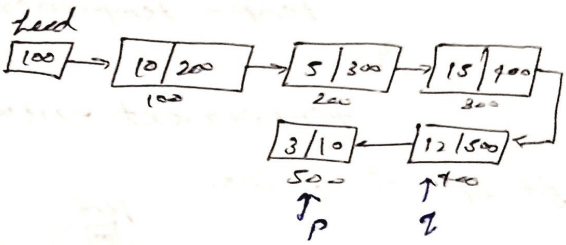
if (head -> next = NULL) {
Free (head);
head = NULL;
}

if (head = NULL) list is empty



At beginning

```
struct node *p;
p = head;
head = p->next;
Free(p);
p = NULL;
```



At end

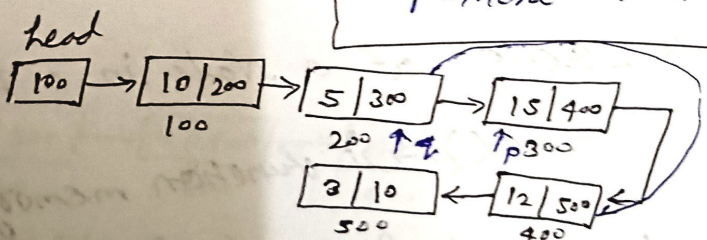
```
struct node *p, *q;
p = head;
while (p->next != NULL) {
    q = p;
    p = p->next;
}
q->next = NULL;
Free(p);
p = NULL;
```

जब pointer memory को free करेगा, 1 pointer previous node के pointer को null कर देगा।

Memory wastage नहीं बचाना है तो

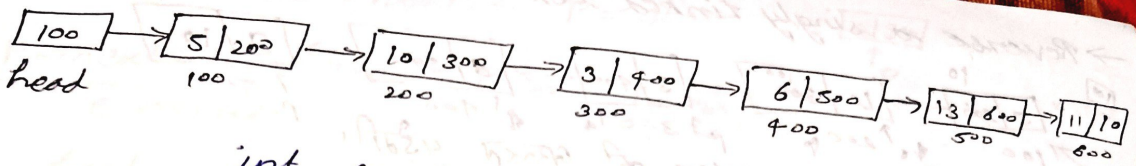
```
while (p->next->next != NULL)
{
    p = p->next;
}
p->next = null;
```

At specified position:




```
Struct node *p, *q;
p = head
while (p->data != 15) {
    q = p;
    p = p->next;
}
q->next = p->next;
Free(p);
p = NULL;
```

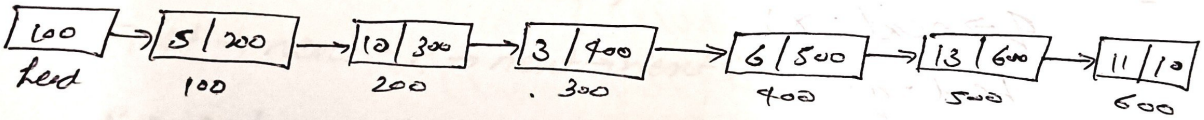
→ Length of singly linked list




```
int count = 0;
struct node *temp;
temp = head;
while (temp != NULL) {
    temp = temp->next;
    count++;
}
printf("%d", count);
```

→ Searching a Node in a singly Linked List

यहाँ पर binary search नहीं apply कर सकते mid नहीं calculate कर पाएंगे।



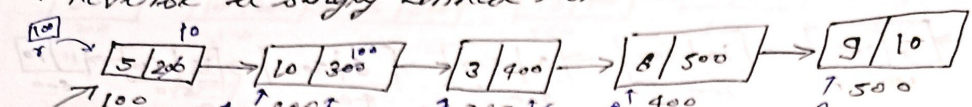
key = 6 को search करना है

```
int flag = 0;
struct node *t;
t = head;
while (t != NULL) {
    if (t->data == key) {
        flag = 1;
        break;
    }
    t = t->next;
}
if (flag == 1) printf("key found");
else printf("key is not found");
```

Time complexity:
O(n)
without break;

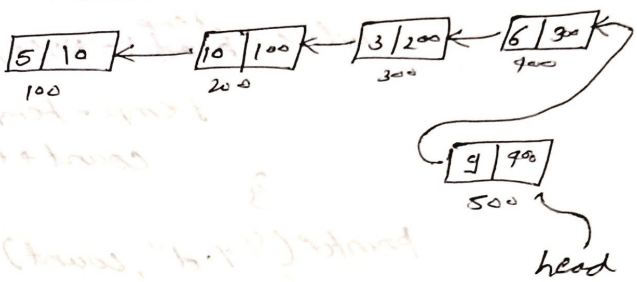
with break
- worst case - O(n)
- Best case - O(1)

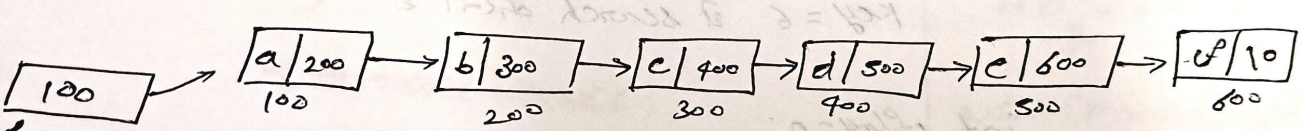
→ Reverse a singly linked list



यहाँ पर 3 pointer की जरूरत पड़ेगी।

```
struct node *p, *q, *r;
    p = head->next;
    q = head;
    r = NULL;
while(p!=NULL) {
    q->next = r;
    r = q;
    q = p;
    p = p->next;
}
    q->next = r;
    head = q;
```






```
struct node *p;
p = head->next->next;
p->next->next->next = p;
    head->next = p->next;
Printf("%c", head->next->next->next->next->data);
```

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

output:- d

p = head → next → next; ⇒ p = 300

p → next → next → next = p → e | 300

e deleted node

c | 400 ← p

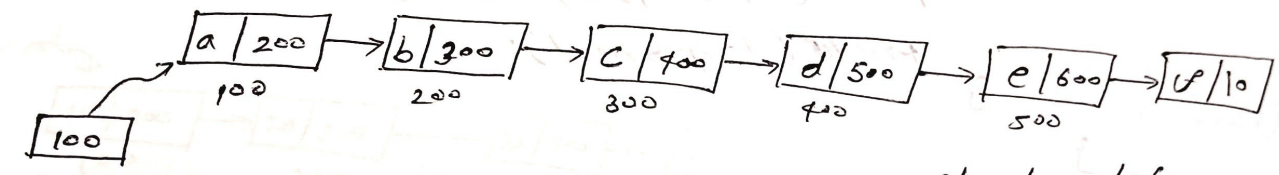
head → next = p → next ; a | 400

head → next → next → next → next → data

d | 500 | e | 600 | c | 400 | d | 500

300

(a)



struct node {
char data;
struct node *next;
};

struct node *p;

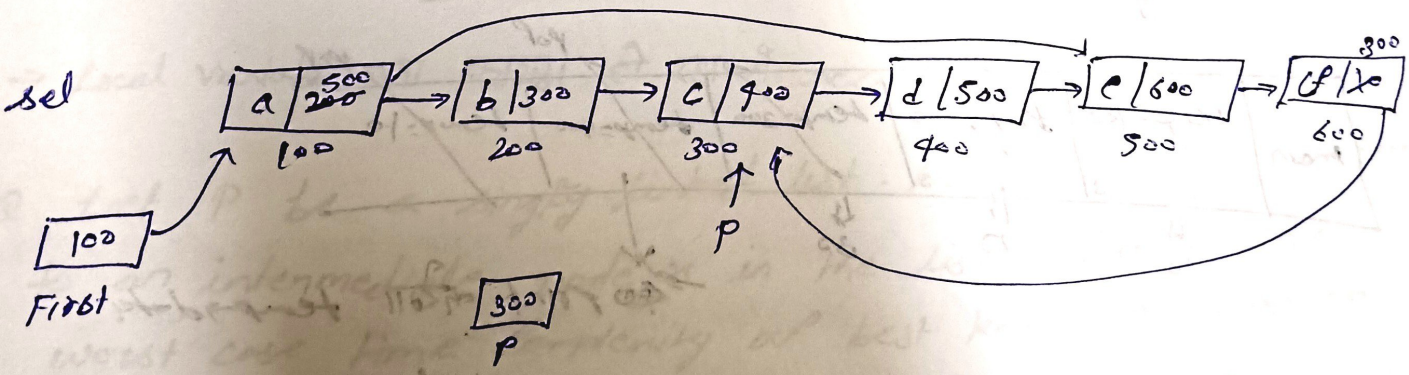
p = First → next → next;

First → next = p → next → next;

p → next → next → next → next = p;

printf("%c", First → next → next → next → data);

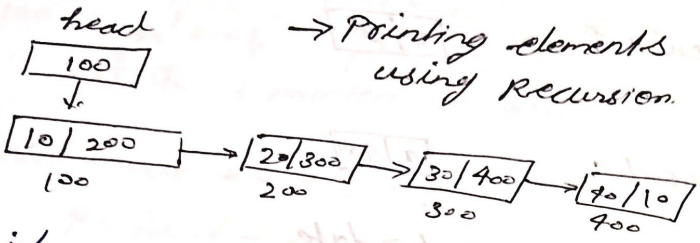
sol



output: C

→ जब भी function call होता है तब stack data structure use में आता है।

→ Printing elements of single linked list using Recursion.

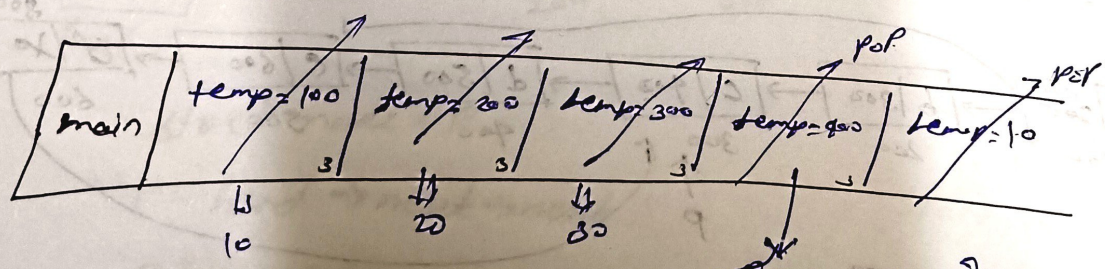



```
void print (struct node *temp) {
    if (temp != NULL) {
        printf("%d", temp->data);
        print(temp->next);
    }
}
```

अगर reverse order में print करना है तो

```
void print (struct node *temp) {
    if (temp != NULL) {
        print(temp->next);
        printf("%d", temp->data);
    }
}
```

print(head);



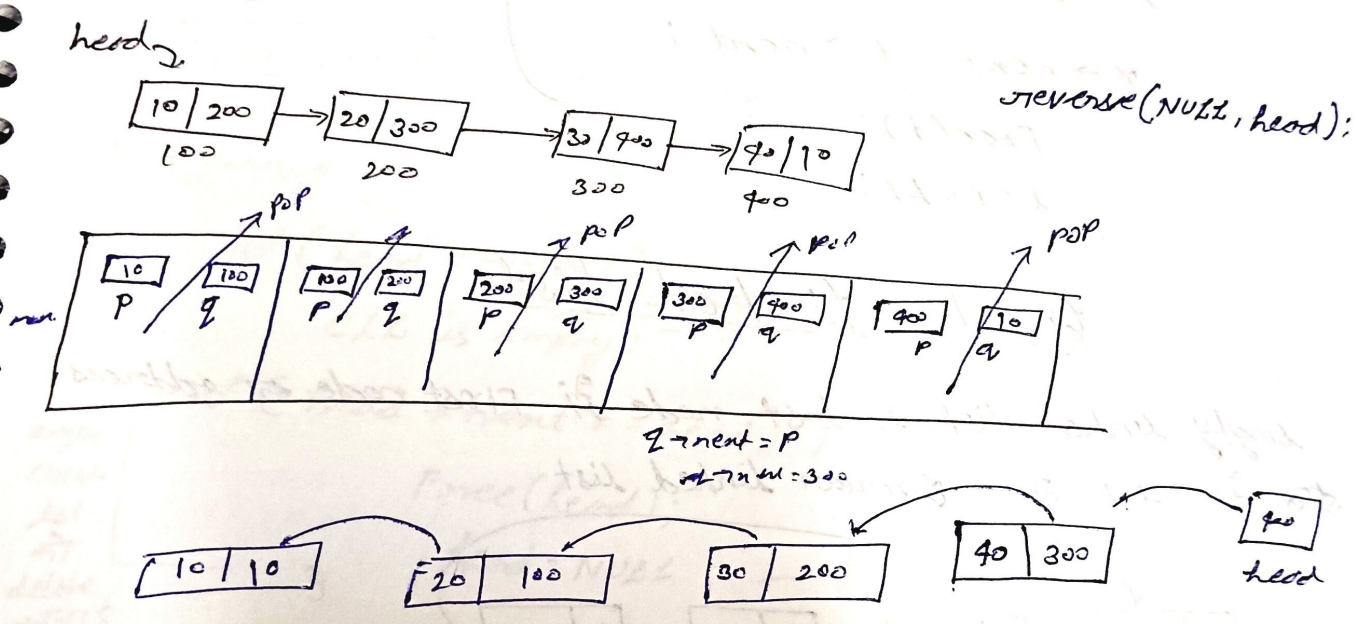
40 30 20 10

Stack First IN LAST OUT

Stack में FILO होता है इसलिए ऐसे print हुआ।

→ Reverse a single linked list (recursive method)

```
void reverse(struct node *p, struct node *q)
{
    if (q != NULL)
    {
        reverse(q, q->next);
        q->next = p;
    }
    else
    {
        head = p;
    }
}
```

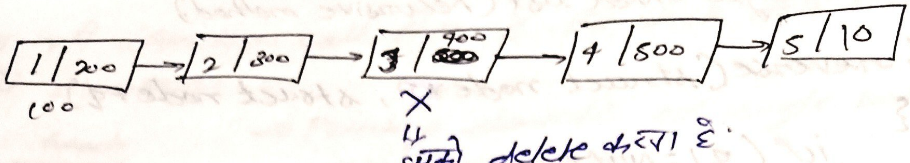


→ Local variable का फायदा हो रहा है recursive call में

Q Let P be a singly linked list. Let Q be the pointer to an intermediate node x in the list. What is the worst case time complexity of best known algorithm to delete the node x from the list.

sol^n O(1) ⇓ copy data from the next node to the node to be deleted & delete the next node

A simple solution is to traverse the linked list until you find the node you want to delete, O(n-2) ⇒ O(n)



इसको delete करना है।

जिस node को delete करना है उसमें next node को copy कर लो हो गया delete और next node को free कर दो

pointer = t = x→next;
जिसको delete करना था

x→data = t→data;

x→next = t→next;

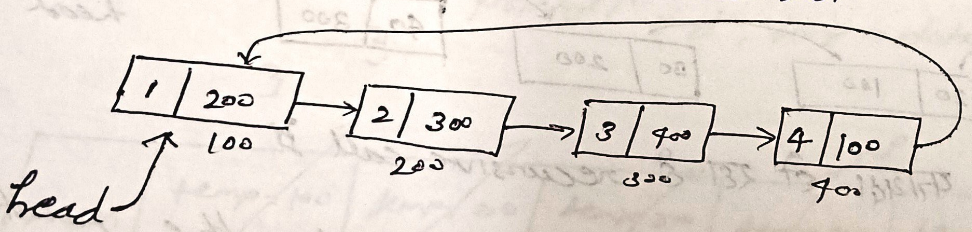
Free(t);

t = NULL;

इसका Time O(1)

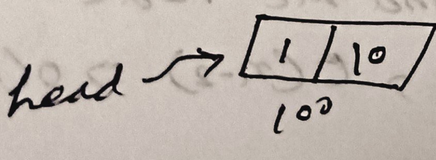
Circular Linked List

singly linked list के last node में First node का address डाल दो बन गया circular linked list

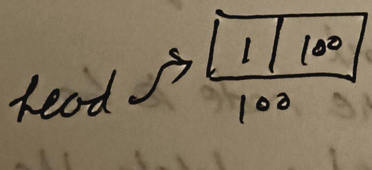


Circular linked list में current node का previous node को access कर सकते। 1 pointer का ही use करके घूम के आ जाएंगे पर singly linked list में ऐसा नहीं कर सकते।

singly linked list with one node

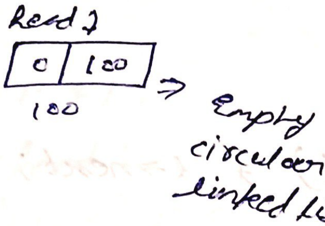


Circular linked list with one node



extra information:

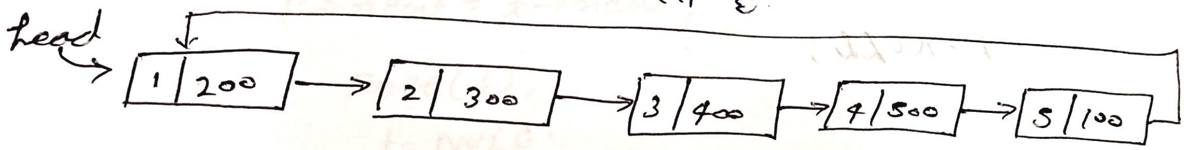
head = '10' ⇓ NULL एक Circular linked list का representation थोड़ा change हो जाता है एक dummy node attach हो जाता है जो count और First node का address store करता है head node बोलते है इसे original circular linked list में नहीं लेते।



ये commonly used notation नहीं है।

→ Deletion (circular linked list)

singly linked list जैसा है



At beginning

```
if (head == NULL)
    CLL is empty;
if (head->next == head) {
    free(head);
    head = NULL;
}
else {
    struct node * temp;
    temp = head;
    while (temp->next != head) temp = temp->next;
    temp->next = head->next;
    head = head->next;
}
```

ये common code है सारे deletion के लिए

single circular list को delete करना है

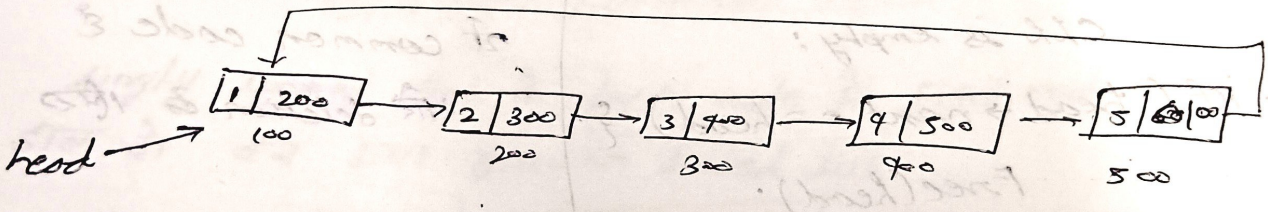
but इस code से memory का wastage हो जाएगा because [1|200] ये वाला node delete नहीं किया।

| 1 | 200 | इसको delete की करना है तो

```
struct node *t, *p;
    t = head;
while (t->next != head)  t = t->next;
    p = head;
head = head->next;
t->next = head;
    free(p);
    p = NULL;
```

At end

पहले जो common code है वो लिखेंगे फिर else part में At the end वाला code

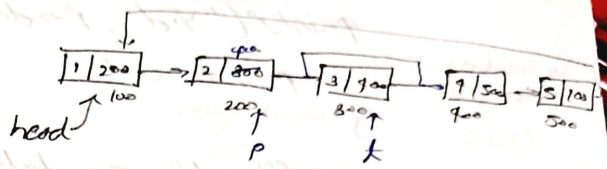



```
struct node *p, *t;
p = NULL ; t = head;
while (t->next != head) {
    p = t;
    t = t->next;
}
p->next = head;
free(t);
t = NULL;
```

→ At particular position

suppose data = 3 वाला node delete करना है

```
struct node *p, *t;
p = NULL; t = head;
while (t -> data != 3) {
    p = t;
    t = t -> next;
}
p -> next = t -> next;
free(t);
t = NULL;
```



→ Circular Linked List Creation

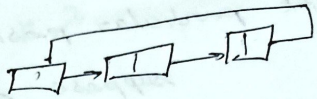
```
struct node {
    int data;
    struct node *next;
};
struct node *head, *new, *temp;
int main() {
    int choice;
    do {
        new = (struct node *) malloc (sizeof (struct node));
        printf("Enter the values");
        scanf("%d", &new->data);
        new -> next = NULL;
        if (head == NULL) head = temp = new;
        else {
            temp -> next = new;
            temp = temp -> next;
        }
        temp -> next = head;
        printf("Do you want to add one more node");
        scanf("%d", &choice)
    while(choice == 1);
}
```

→ First node का address store कर रहा है

→ newly created node का location

→ link कर रहा है सारे node को

→ Circular linked list Traversal



```
struct node *p = head;
while (p->next != head) {
    Printf("%d", p->data);
    p = p->next;
}
Printf("%d", p->data);
```

यहाँ पर p != head लिखते हैं तो 1st iteration में ही false हो जाएगा।

⇒ last वाला node नहीं print होगा इसलिए इस printf का use पड़ा

* suppose 1 ही printf use करना है तो do while loop use करेंगे।

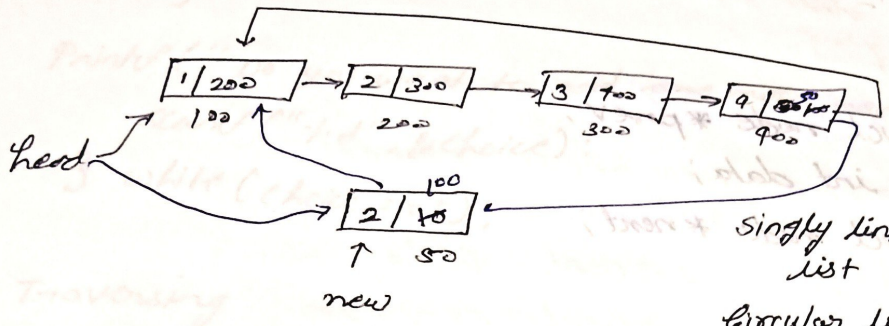
```
struct node *p = head;
do {
    if (p->data
    Printf("%d", p->data);
    p = p->next;
} while (p->next != head);
        p != head
```

→ Insertion in circular linked list

अगर Circular linked list empty है तब कैसे insert करेंगे:

```
if (head == NULL)
{
    head = new;
    new->next = head;
}
```

अब हम suppose कर रहे हैं कि linked list में किसी element को insert करना है।



Singly linked list — Insertion at beginning O(1)

Circular linked list — O(n)

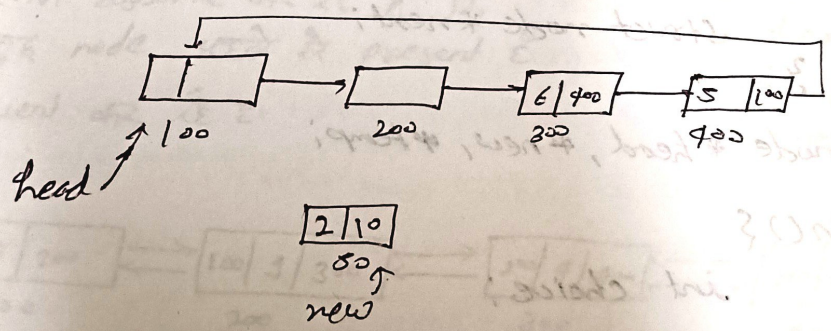
At beginning

```
struct node *p = head;

while (p->next != head) p=p->next;

p->next = new;
new->next = head;
head = new;
```

At last




```
struct node *p = hed;
while (p->next != head) p = p->next;
p->next = new;
new->next = head;
```

At particular position

```
struct node *p = head
while (p->data != 6) p = p->next;
new->next = p->next;
p->next = new;
```

Doubly Linked List

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

→ Creation of Doubly linked list

```
struct node {
        struct node *prev;
          int data;
        struct node *next;
    };
struct node *head, *new, *temp;
int main(){
        int choice;
    do{
    new = (struct node *) malloc (sizeof (struct node));
    Printf("Enter the values");
    Scanf("%d", &new->data);
        new->prev = NULL;
        new->next = NULL;
  if (head == NULL){
        head = temp = new;
    }
else{
        temp->next = new;
```

```
        new->prev = temp;
        temp = temp->next;
    }
    Printf("Do you want to add some more node");
    scanf("%d", &choice);
} while (choice == 1);
```

→ Traversing

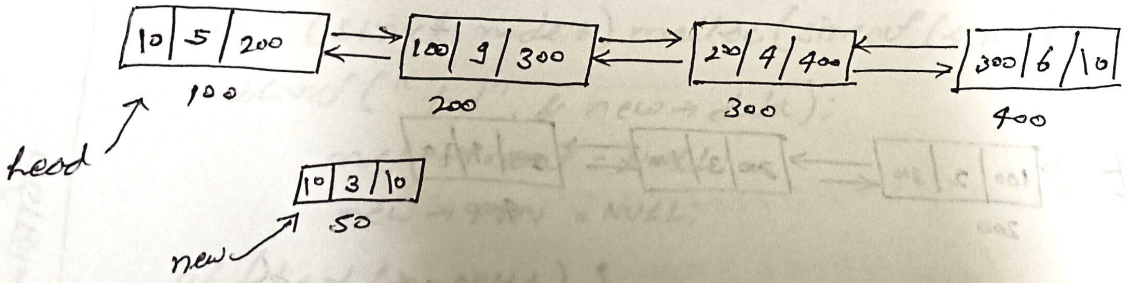
```
struct node *p = head;
while (p != NULL) {
    printf("%d", p->data);
    p = p->next;
}
```

→ Insertion

हम लोग assume कर रहे हैं कि कुछ node पहले से present है और insert कर रहे हैं।

```
if (head == NULL)
    head = new;
```

जब head NULL हो तब ऐसे करते हैं



At beginning

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

At last

```
struct node *t;
t = head;
while (t->next != NULL) {
    t = t->next; }
t->next = new;
new->prev = t;
```

At particular node

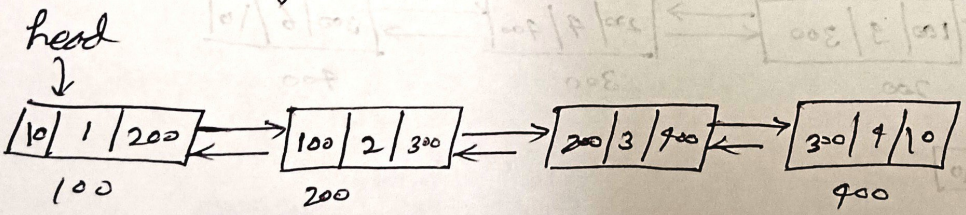

```
struct node *t;
t = head;
while (t->data != 4) t = t->next;

new->next = t->next;
new->prev = t;
t->next = new;
new->next->prev = new;
```

→ Deletion from double linked list

deletion ई चाहे कहाँ से करना ई न code common होगा।

```
if (head == NULL)
    DLL is empty;
if (head->next = NULL)
{ free (head);
    head = NULL;
}
else {
    इस part में सबका code लिखेंगे।
}
```



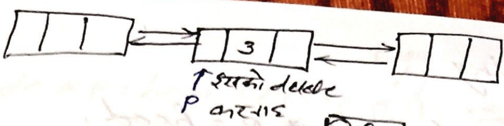
At beginning

```
struct node *P = head;
head = head->next;
head->prev = NULL;
free (P);
P = NULL;
```

At end

```
struct node *P = head;
while (P->next != head)
    P = P->next;
P->prev->next = NULL;
free (P);
P = NULL;
```

At particular node

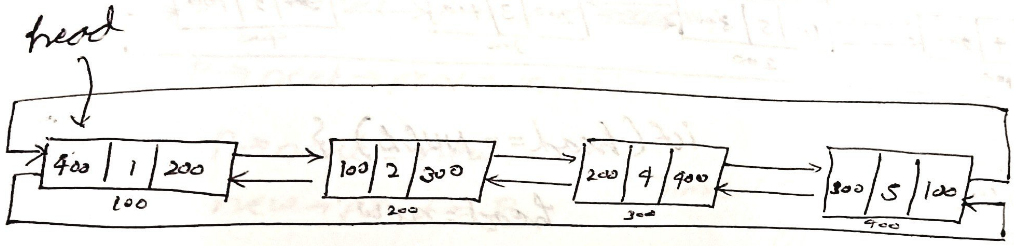



```
struct node *p;
    p=head;

while(p->data!=3) p=p->next;

        p->prev->next = p->next;
        p->next->prev = p->prev;
            free(p);
            p=NULL;
```

Circular Doubly Linked List



→ creation

```
struct node *temp, *prev, *next;
~~new =(struct *node)~~

new= (struct node*) malloc(sizeof(struct node));
    scanf("%d", &new->data);
        new->next = NULL;
        new->prev = NULL;
    if (head == NULL) {
            head = temp = new;
            head->next = new;
            head->prev = new;
    }
else {
        temp->next = new;
        new->prev = temp;
        new->next = head;          temp=temp->next;
        head->prev = new;     }.
```

इसको loop में चलाएंगे

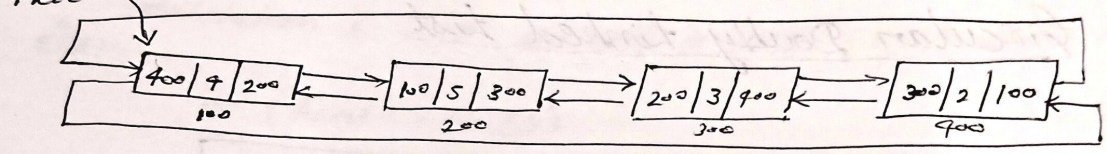
→ Traversing

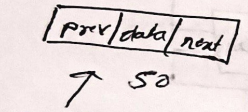
```
struct node * p = head;
while (p->next != head) {
    printf("%d", p->data);
    p = p->next;
}
printf("%d", p->data);
```

OR

```
struct node * p = head;
do {
    printf("%d", p->data);
    p = p->next;
} while (p != NULL);
```

→ Insertion






```
if (head == NULL) {
    head = new;
    head->next = head;
    head->prev = head;
}
else {
    यहां लिखना Code जिसमें 3 नोड।
}
```

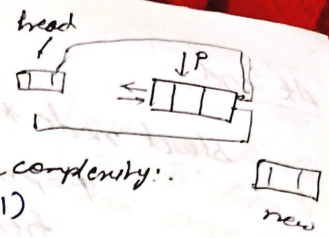
At beginning

```
struct node *p = head->prev;
head->prev = new;
new->next = head;
new->prev = p;
p->next = new;
head = new;
```

Time Complexity :- O(1)

At end

```
struct node *p = head->prev;
p->next = new;
new->prev = p
new->next = head
head->prev = new;
```

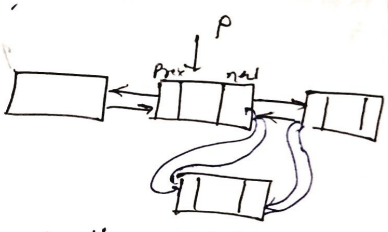


Time complexity:
O(1)

At particular position

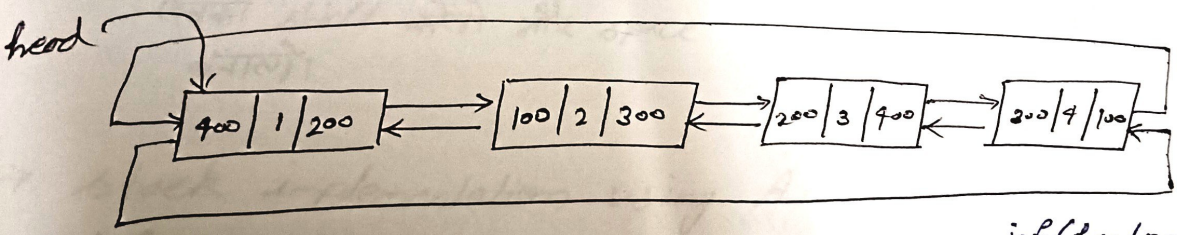
data = 3 के बाद insert करना है।

```
struct node *p;
while (p->data != 3) p = p->next;
new->next = p->next ~~->prev~~;
p->next->prev = new;
p->next = new;
new->prev = p;
```



Time complexity
O(n)

→ Deletion

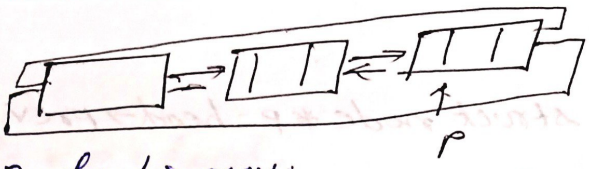


At beginning

```
struct node *p, *t;
p = head->prev;
t = head;
head = head->next;
head->prev = p;
p->next = head;
Free(t);
t = NULL;
```

```
if (head == NULL)
    CDLL is empty
if (head->next == head)
{
    Free(head);
    head = NULL;
}
else {
    यहाँ पर code
    लिखेंगे
}
```

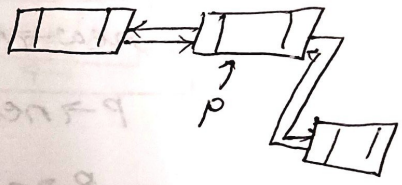
At end




```
struct node *p = head->prev;
p->prev->next = head;
head->prev = p->prev;
Free(p);
P=NULL;
```

At particular position

```
struct node * p = head;
while(p->data != 3) p = p->next;
p->prev->next = p->next;
p->next->prev = p->prev;
Free(p);
P=NULL
```



Stack

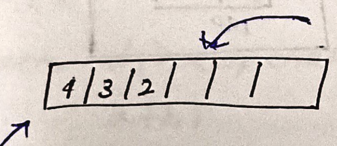
Physical data structure क्या हो है < Linked List, Array] Memory में क्या यही हो form में data store होता है।

1) Push (x) :- top पे element को push कर देगा (x को)
2.) Pop () :- top of stack के element को delete कर देगा।
3.) Peek () :- top of stack के element को दे देगा।
4.) Is full () :- stack full है तो True देगा।
5) Is Empty :- stack Empty है तो False देगा।

LIFO, FILO] दोनो एक ही है।

* Recursion में stack का use होता है।
* Balance Parenthesis
* Parsing
* Infix to postfix
* Text Editor

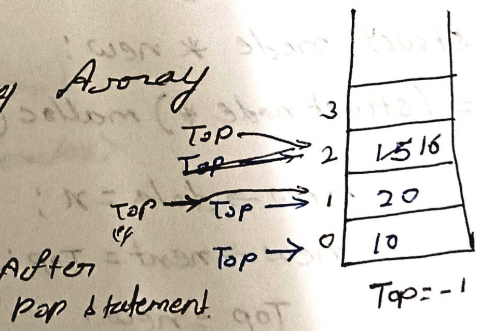
} Application of stack

→ 

but इधर से push करना है तो O(1) time में हो जायगा इसलिए stack को जब implementation करेंगे तब right hand side को top बनायेंगे

यहाँ पर push करना होगा तब सबको shift करेंगे और space बनायेंगे।

→ stack implementation using Array

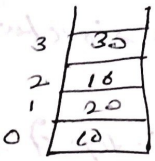


push(10);
push(20);
push(10);
Pop();
Pop(16);

```
#define <stdio.h>
#define size 5
 int stack[size];
  int Top = -1;
void push(intx) {
      if (Top = size-1) Printf("stack overflow");
      else {
            Top = Top+1;
            stack[Top] = x;
      }
```

```
void pop() {
      if (Top == -1)
         Printf("stack underflow");
      else {
         Top = Top-1;
      }
}
```

pop में सिर्फ top को change कर देते हैं actual में element को delete नहीं करते।

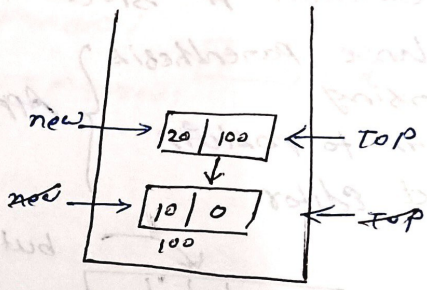


→ Traversal in stack

```
for(int i = top ; i >= 0; i--)
    {
      printf("%d", stack[i]);
    }
```

→ Stack implementation using linked list

linked list में insertion at beginning में सबसे कम time लगता है O(1)
∴ Front को ही top बनाएंगे।




```
#include <stdio.h>
struct node {
        int data;
        struct node *next;
    };
struct node * Top = NULL;
void push (int x) {
    struct node *new;
new= (struct node *) malloc (sizeof(struct node));
        new->data = x;
        new->next = Top;
            Top = new;
}
void pop() {
        struct node *t;
    if (Top = NULL) printf("stack underflow");
    else {
            t = Top;
          Top = Top->next;
            Free(t);
            t=NULL;
} }
```

* linked list में कभी overflow की condition नहीं आएगी।

push - O(1)
pop - O(1)

→ Traversal

```
struct node *P;
    P = Top;
while (P != NULL)
{   printf("%d", P->data);
        P = P->next;
}
```

Infix, Prefix & Postfix expression

operator Precedence Table

operator	Precedence	Associativity
exponent ← ^ (2^3, 3^7 etc.)	1	Right to Left
*, /	2	Left to Right
+, -	3	Left to Right

1.) INFIX

<operand$_1$> <operator> <operand$_2$>

a+b;

2.) PREFIX (Polish Notation)

<operator> <operand$_1$> <operand$_2$>

+ a b

3.) POSTFIX (Reverse Polish Notation)

<operand$_1$> <operand$_2$> <operator>

a b +

* operand का order किसी में भी change नहीं होता चाहे INFIX, PREFIX, POSTFIX हो।

Q Postfix: 234*+

करते हैं क्या है कि जैसे ही L → R जाने पर पहला operator मिले उनके previous दो operator पर वो operation लगा दो and sequence का ख्याल रखते हुए

2 3 4 * + ⇒ 2 3*4+ ⇒ 2 12 + = 2+12 = 14

L → R

3, 4 में पहला operand 3 है।

Prefix: +2*34 Right to left evaluate करते है

3,4 में पहला operand है

+2*34 ⇒ +2 3*4

= +2 12

= 2+12 = 14

INFIX: 2+3*4 → पूरा scan किया पहला highest precedence का मिला *

2+12 → फिर scan किया highest precedence का मिला +

14

* Prefix, postfix में एक बार ही scan करते है Left to Right whereas Infix में more than one scan करना पड़ता है।

Scan = No. of operator

+2*34 ← यह पूरा scan में evaluate हो जाएगा

Pow (↑) के:

2↑3↑2 इसको Right to left evaluate करते है

2↑9

512

Q Infix.

2+3*4/6 - 2↑1↑2*2/2 = 2

R→L ↑ ये मिला

2↑1↑2 → 1^2

2↑1 = 2

2+3*4/6 - 2*2/2 → *

2+12/6 - 2*2/2 → */

2+2 - 2*2/2 → *

2+2 - 2/2 → /

2 + 2 - 2 +

2 + 0

2

Postfix

2 3 4 * 6 / 2 + 2 1 2 ↑ ↑ 2 * 2 / - 3*4 = 12

2 1 2 6 / + 2 1 2 ↑ ↑ 2 * 2 / - 12/6 = 2

2 2 + 2 1 2 ↑ ↑ 2 * 2 / - 2+2 = 4

4 2 1 2 ↑ ↑ 2 * 2 / - 1↑2 = 1

4 2 1 ↑ 2 * 2 / - 2↑1 = 2

4 2 2 * 2 / - 2*2 = 4

4 4 2 / - 4/2 = 2

4 2 - 4-2 = 2

2

Prefix

- + 2 / * 3 4 6 / * ↑ 2 ↑ 1 2 2 2 1↑2 = 1

- + 2 / * 3 4 6 / * ↑ 2 1 2 2 2↑1 = 2

- + 2 / * 3 4 6 / * 2 2 2 2*2 = 4

- + 2 / * 3 4 6 / 4 2 4/2 = 2

- + 2 / * 3 4 6 2 3*4 = 12

- + 2 / 1 2 6 2 12/6 = 2

- + 2 2 2 2+2 = 4

- 4 2 4-2 = 2

2

→ Infix → postfix conversion & Infix to prefix

Infix: a+b*c (scan करेंगे highest priority वाले को पहले करेंगे)

a+[*bc]

+a*bc ← prefix

a+b*c (scan करेंगे highest priority वाले को पहले करेंगे)

a+[bc*]

abc*+ → postfix

** operand का order change नहीं होना चाहिए infix, postfix, prefix में।

Infix: (A+B) * C/D (bracket highest priority)

[+AB] * C/D

[*+ABC] / D

/*+ABCD ⇒ prefix.

[AB+] * C/D

[AB+C*] / D

AB+C*D/ ⇒ postfix

Infix: a+b * c/d↑e↑f *a-b/c

↑ ⇒ priority सबसे ज्यादा है

a+b*c/d↑[e↑f] *a-b/c

a+b*c/[↑d↑e↑f] *a-b/c

a+[*bc] / [↑d↑e↑f] *a-b/c

a+[/*bc↑d↑e↑f] *a-b/c

a+[*/*bc↑d↑e↑f a]-b/c

a+[ ] -/[/bc]

[+a*/*bc↑d↑e↑f a]-[/bc]

[-+a*/*bc↑d↑e↑f a/bc] Prefix

a+b*c/d↑e↑f*a-b/c

↑ high priority

a+b*c/d↑ [ef↑]*a-b/c

a+b*c/[def↑↑]*a-b/c

a+[bc*]/[def↑↑/]*a-b/c

a+[bc*def↑↑/]*a-b/c

a+[bc*def↑↑/a*]-[bc/]

[abc*def↑↑/a*+]-[bc/]

| abc*def↑↑/a*+bc/- | postfix

| ab*cdef↑↑/a*+bc/- |

Infix to postfix conversion Algorithm

H = high precedence

L = low precedence

E = Equal precedence

opt = कोई भी operator हो

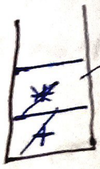
	i/p	TOS	operation
1	H	L	PUSH
2	L	H	POP
3	E	E	PUSH (R-L)
4	E	E	POP (L-R)
5	(	opt	PUSH
6	opt	(	PUSH
7	(	(	PUSH

→ Equal precedence है तो associativity अगर R→L हुई तो push करो

1) a+b*c

अब * मिला * का i/p precedence ज्यादा है + ⇓ TOP

postfix
abc*+



operator stack यहाँ बस operator डालेंगे

stack empty है तो push कर लो operator और operand को point कर लो जैसे मिलता जायेगा फिर pop कर लो stack के element को

L→R जाता है

2.) a*b+c

↑ + जैसे ही आया है low है precedence तो pop कर देंगे।

postfix
ab*c+

a + b − c

pop कर देंगे

postfix
ab+c−

4) a ↑ b ↑ c → push कर देंगे

postfix
abc↑↑

5)

a * (b+c)

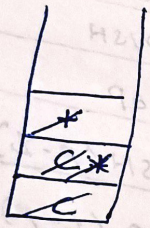
postfix.
abc+*

जैसे ही हमें close parenthesis मिलेगा तभी element को pop कर देंगे जब तक open parenthesis न मिल जाए.

** postfix में कहीं भी parenthesis नहीं लिखते.

6)

((a+b) * c)

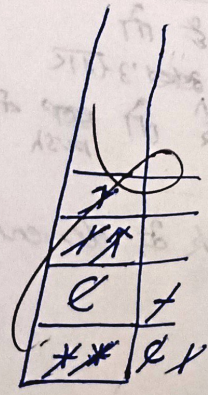


operator stack

postfix.
ab+c*

Q

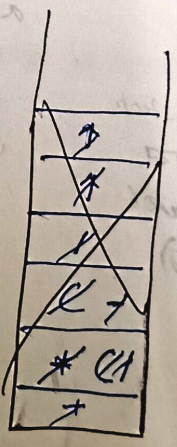
a + b * (c / d ↑ e ↑ f) * (a − b) / c



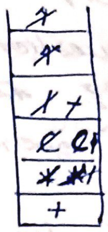
operator stack

postfix
~~a b * c d / e f ↑ ↑ * a b − c / +~~

postfix
~~a b c d e f ↑ ↑ / * a b − c / +~~



Q. a+b*(c/d↑e↑f)*(a-b)/c



Postfix

abcdef↑↑/*ab-c$*/+

गलती क्या कर रहा था कि

जब equal precedence का था और pop कर रहे थे तब pointer को बढ़ा दे रहा था but pop करने के बाद भी pointer उसी point पर point करेगा।

Infix to postfix conversion का time complexity - O(n)

Postfix को solve करने का time complexity: O(n)

Total time complexity: O(n) + O(n)
= 2 O(n)
= O(n)

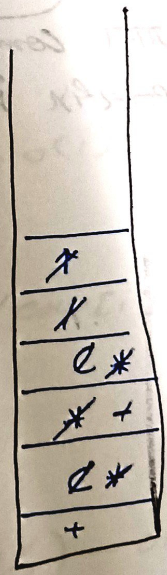
Q. A+(B*C-(D/E↑F)*G)*H

इसको tabular form में भी करते है

symbol | stack | postfix expression

जो ज्यादा मन देख के सीधे भी कर सकते है

same चीज है



Postfix

ABC*DEF↑/G*-H*+

Postfix Expression Algorithm (evaluation)

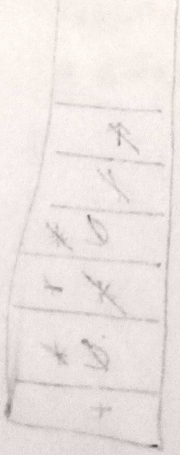
जो previous questions में postfix expression को evaluate कर रहे थे उसी का code हम यहाँ लिख रहे हैं इसमें हमें कोई operator मिल रहा था उसके just 2 operand पर operator लगा देते।

```
Postfix Evaluation(exp) {
    create a operand stack s  — O(1)
    for int i = 1 to n. {
        if(exp[i] is operand) push(exp[i]) — O(1)
        else {
            OP2 = pop()  — O(1)
            OP1 = pop()  — O(1)
            X = perform(exp[i], OP1, OP2) — O(1)
            push(x) — O(1)
        }
    }
}
```

Top of stack OP_2 बनेगा।

Time Complexity = $n \times O(1)$
= $n O(n)$

→ Infix expression को postfix expression में convert कर लेते हैं हमारा compiler ये काम करता है।
Prefix में बहुत कम करते हैं।



Infix to Prefix conversion Algorithm

Infix: $A+(B*C-(D/E\uparrow F)*G)*H$

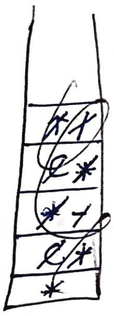
Rule simple है. postfix वाला
High priority वाला आए तो push कर लो Low priority आया तो pop कर लो
(है तो हमेशा push
) मिले तो pop करो जब तक open bracket न मिल जाए
Equal precedence है R→L push L→R pop

1st इसका reversal निकाल दो

(→)
) → (

$H*(G*(F\uparrow E/D)-C*B)+A$

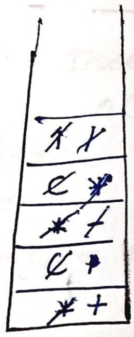
अब इसका postfix निकाल देंगे



Postfix

~~H G F E ↑ D / * C B * - A * * A * +~~

H G F E ↑ D / * C B * - * A + → इसका Reversal निकाल लो prefix हो जाएगा



| + A * - * B C * / D ↑ E F G H | ⇒ prefix

⇒ Prefix कैसे evaluate करते है पीछे किए हुए है हमलोग अब उसी का algorithm है.

* postfix सबसे ज्यादा use करते है

```
Prefix Evaluation(exp) {
    create a operand stack S — O(1)
    for i = n to 1 {
    if (exp[i] is operand) push(exp[i]) — O(1)
    else {
        OP1 = pop(). — O(1)
        OP2 = pop() — O(1)
        X = perform(exp[i], OP1, OP2) — O(1)
            push(x)
        }
}
```

So, Time complexity is $= n \times O(1)$
$= O(n)$

→ precedence table देकर पूछ सकता है solve करो

+	1	L → R
/	2	L – R
↑	3	R → L
–	4	R – L
*	5	L → R

$5 + 4 * 3 \uparrow 2$ → +

$9 * 3 \uparrow 2$ → ↑ $\quad 3 \uparrow 2 = 9$

$9 * 9$

81

For given precedence table output is 81

$5 + 4 * 3 \uparrow 2$ convert it into postfix using above precedence table

[5 4 +] * 3 ↑ 2

[5 4 +] * [3 2 ↑]

5 4 + 3 2 ↑ *

$5 * 6 + 2 - 12 / 4$ → +

$5 * 8 - 12 / 4$ → /

$5 * 8 - 3$ → –

$5 * 5$

25

5 * 6 + 2 – 12 / 4

5 * [6 2 +] – 12 / 4

5 * [6 2 +] – [12 4 /]

5 * [6 2 + 12 4 / –]

5 6 2 + 12 4 / – *

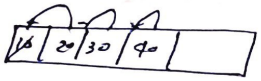
Queue

Queue insertion 1 end से होगा और deletion दूसरे end से.

Queue works on FIFO ⇒ First in First out

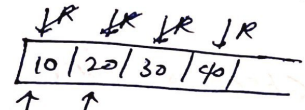
	Insertion	deletion
	O(1)	O(n)

यहाँ पर space waste हो रहा है।



जब delete करेंगे तो वो space खाली होगा और सबको shift करना पड़ेगा।

deletion को O(1) बनाना है तो दो variable ले लेंगे F, R जब delete करेंगे तो F को एक increment कर देंगे.



एक element ही होगा तो F & R उसी को point करेगा।

Now, Insertion O(1) deletion O(1) ⇒ यहाँ पर time complexity कम है पर space बरबाद ही हो गया।

Application:-
Printer
Customer care service इसको resole करने के लिए circular queue का use करते है।

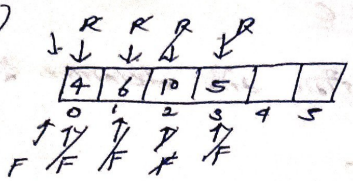
Queue implementation using Array

```
#include <stdio.h>
#define size 5
   int Q[size]
   int F = -1;
   int R = -1;
enqueue (int x) {
   if (R == size-1)
      Printf("Q is FULL");
   else {
         R = R+1;
         Q[R] = x;
   if (F == -1)
      }  F = 0;
}
```

```
Dequeue() {
   if(F == -1 && R == -1)
      Printf("Q is empty");
   else if ( F == R) {
      Printf("%d", Q[F]);
         R = -1;
         F = -1;
   }
   else {
      printf("%d", Q[F]);
         F = F+1;
   }
}
```

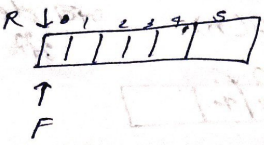
size = 6

```
enqueue(4);
enqueue(6);
enqueue(10);
dequeue();
enqueue(5);
dequeue();
dequeue();
dequeue();
```



enqueue – O(1)
dequeue – O(1)

⇒ After all these operation.



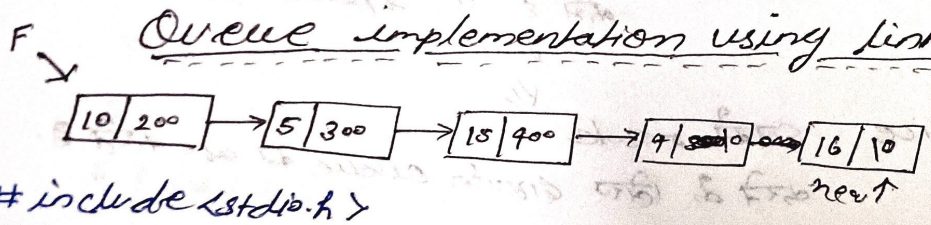
ऐसा दिखेगा queue

→ Traversal in queue

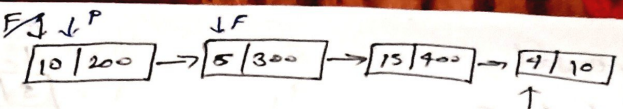
```
For(i=F; i<=R; i++) {
    Printf("%d", Q[i]);
}
```

⇒ check कर लेंगे कि element present है या नहीं। queue empty ही हो तब garbage value print कर देगा।

Queue implementation using linked list




```
#include<stdio.h>
struct node {
    int data;
    struct node *next;
};
struct node *F, *R;
void enqueue(int y) {
    struct node *new;
new = (struct node *) malloc(sizeof(struct node));
    new->data = y;
    new->next = NULL;
if (F==NULL) F=R=new;
else {
    R->next = new;
    R = new;
}
}
```



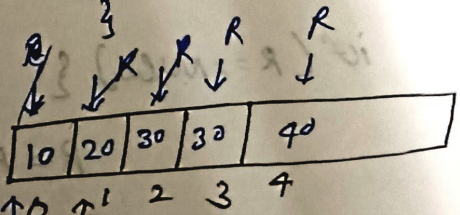

```
int dequeue(){
    int x=-1;
    struct node *p;   => ~~node~~ node को delete करने के लिए अस्थायी रूप से
if (F==NULL)
    Printf("Q is empty");
else{
        p=F;
        F=F->next;
        x=p->data;
        Free(p);
        p=NULL;
    }
    return x;
}
```

Circular Queue ⇒ Queue ही है normal पर उस update और condition बदलना पड़ता है

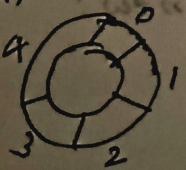
```
#include<stdio.h>
# define size 5
  int Q[size];
  int F=-1, R=-1;
void enqueue(int x){
    if((R+1) mod size == F)
        Printf("CQ is FULL");
    elseif (F==-1 && R==-1){
        F=R=0;
        Q[R]=x;
    }
else
    {
        R=(R+1) mod size;
        Q[R]=x;
    }
}
```

Time complexity
Insertion - O(1)
deletion - O(1)

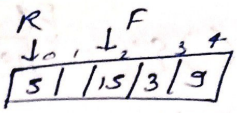
```
Deque(){
    if (F==-1 && R==-1)
        printf("CQ is Empty");
    else if (F==R)
        F=R=-1;
    else
        F=(F+1) mode size;
}
```



dequeue();



→ Traversal



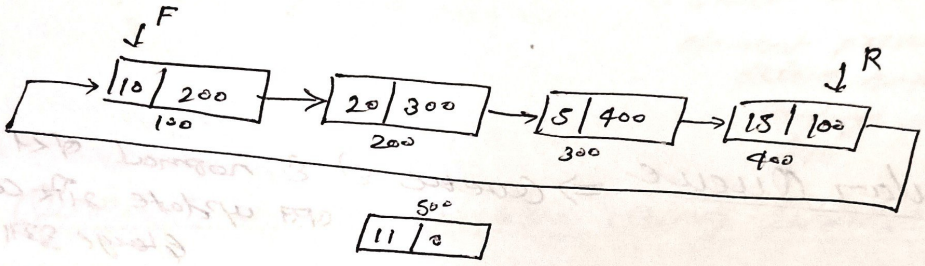
j = 1 to 0

15, 3, 9, 5

```
int i = F;
while (i != R)
{
    printf("%d", Q[i]);
        i = (i+1)%qsize;
}
printf("%d", Q[i]);
```

→ Circular Queue using linked list

linked list में circular linked list होता है तो उसी का use करेंगे।




```
struct node {
        int data;
    struct node *next;
};
struct node *F, *R;
enqueue(int x) {
        struct node *new;
    new = (struct node *) malloc (sizeof (struct node));
            new → data = x;
            new → next = NULL;
    if (R == NULL) {
            R = F = new;
    }
    else {
            R → next = new;
```

```
        R->next = new;
        new->next = F;
        R = next;
    }

Dequeue() {
    struct node *P;
        P = F;
    if (F == NULL) printf("Q is empty");
    else if (F == R) {
            F = R = NULL;
            free(P);
            P = NULL;
        }
    else {
            F = F->next;
            R->next = F;
            free(P);
            P = NULL;
        }
}
```