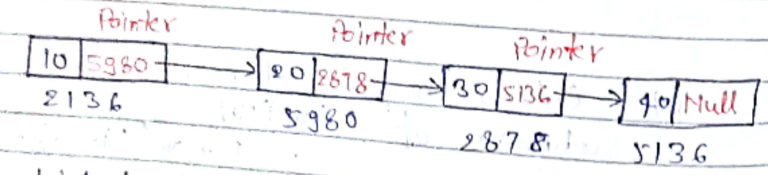
Linked List

Linked List is a linear data structure like array.

Note: In case of array all the elements are stored at successive memory locations i.e. linear order is maintained implicitely.

But in the case of linked list, elements are not stored at successive memory locations. They are stored at random positions.

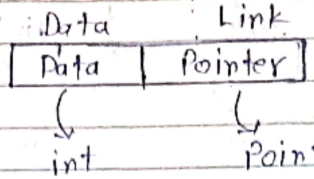
Ex:



Note: Linked List is not a collection of data. It is a collection of elements called nodes.
Each node have 2 fields at least —
1. Data information
2. Pointer, that contain address of next node.

node

How to implement a node in Linked List:

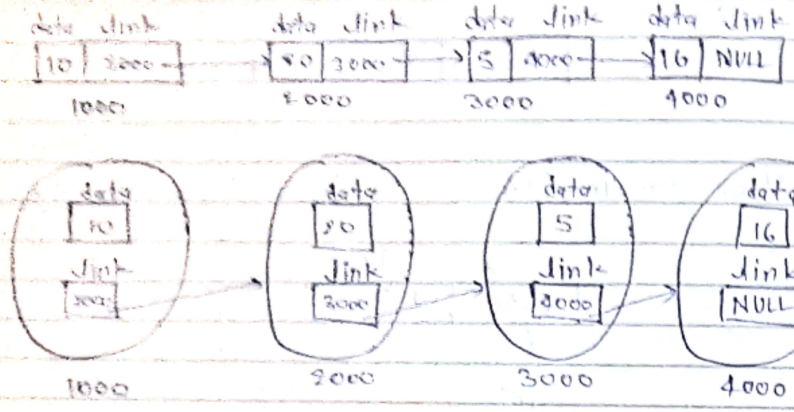


Note: अगर ये दोनों ही field एक ही type की होती तो इसे हम 2 size का एक array लेकर easily implement कर सकते थे। But अभी इन दोनों ही fields (i.e. Data field, address/pointer field) का type अलग - अलग है।

So different data types के group को structure की help से implement किया जाता है।

So it will be implemented as —

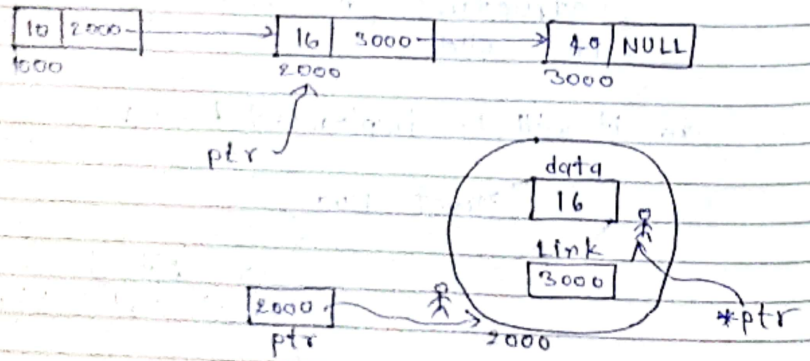
```
Struct Node
{
    int data;
    struct Node *Link;
};
```



Here every node is a structure and link of a structure is pointing to the next structure. Hence link is called as pointer to structure.

So link is pointer to structure.

Ex:

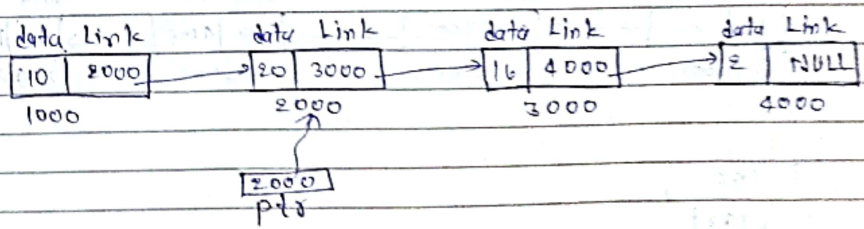


struct Node *ptr;

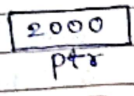
(1)	(*ptr).data	o/p → 16
(2)	(*ptr).Link	o/p → 3000
	OR	
(3)	ptr → data	o/p → 16
(4)	ptr → Link	o/p → 3000

Hence, we have two ways to access the members of a structure as discussed above.

How to go to the next node in linked list?



Here initially ptr is pointing to the node 2. That means ptr value is 2000 (i.e. address of node 2)



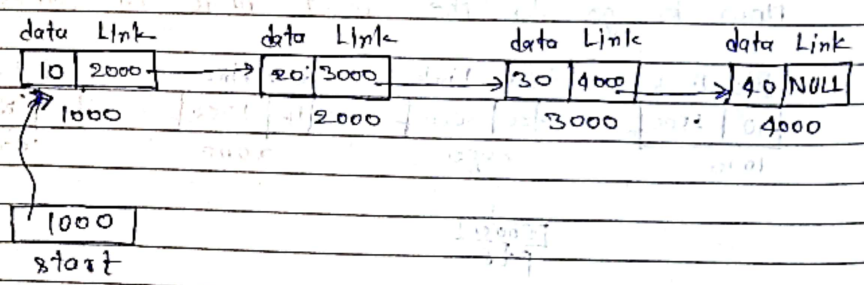
so to go to the next node (i.e. node 3) we have to assign node 3 address 3000 to ptr.

and it will be assigned by writting following code —

```
ptr = ptr → Link;
```

Here Ptr → Link gives 3000 because in node 2 Link feild has the value 3000.

* start / head / front pointer:



(Pointer to first node)

* it holds address of first node in linked list.

Note: start is not a node. It is a pointer to a node. That means it will hold the address of a node.

so, declearation of start pointer should be like —

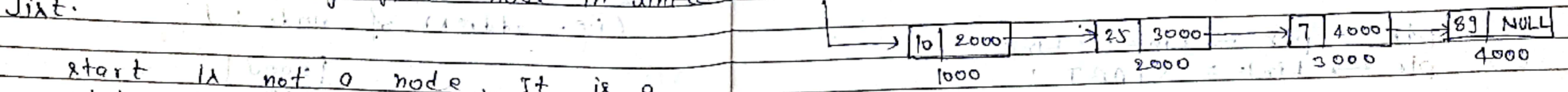
```
struct Node * start;
```

What if Linked list is empty?

If linked list is empty that means there is no first node. and hence there will not be any valid address of first or any node. so start pointer will contain NULL.

That means linked list is empty.

How to access the elements of a linked list.

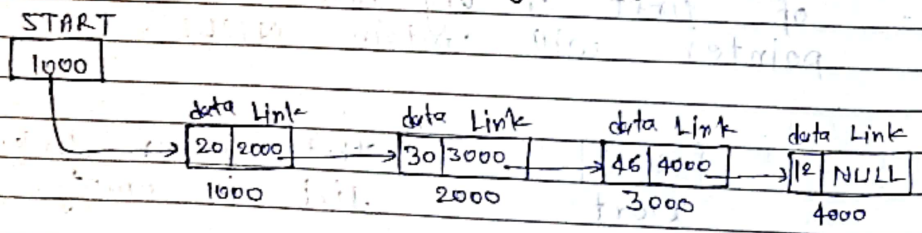


START ≡ 1000 (Pointer to 1st Node)
/Pointer to structure

START → data ≡ 10
START → Link ≡ 2000 (Pointer to 2nd node/structure)

START → Link → data ≡ 20
START → Link → Link ≡ 3000 (Pointer to 3rd node)

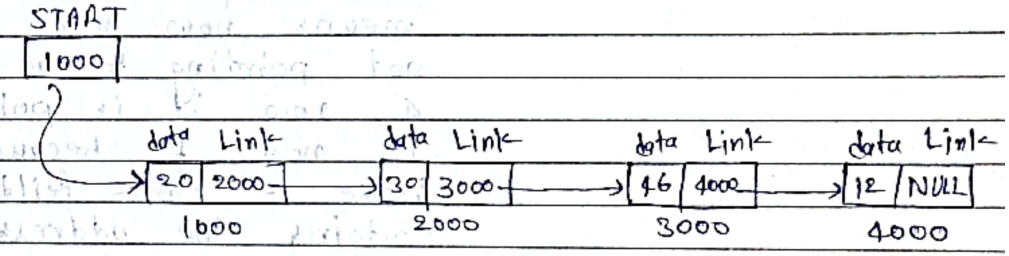
Ques: Consider the following array –



What will be the output after executing following code –

```
struct Node *ptr;
ptr = START;
① ptr = ptr → Link → Link;
② ptr → Link = START;
③ ptr = ptr → Link → Link;
④ ptr = ptr → Link;
⑤ printf("%d", ptr → data);
```

Sol:



struct Node *ptr; means ptr is a pointer variable which is pointing to a node.

ptr = START; means ptr is 1000 because START is 1000. so now ptr will also point to the first node.

ptr = ptr → Link → Link; Here ptr → Link means 2000 and ptr → Link → Link means 3000. so ptr is now 3000.

ptr → Link = START ; Now START value which is 1000 will be assigned to ptr → Link. That means in node 3, we will go to the Link feild and will write there 1000 in the place of 4000. That means now node 3 is not pointing to the node 4. now it is pointing to node 1 because node 3 Link feild contains the address of node 1.

ptr = ptr → Link → Link ; Here ptr → Link means 1000 and ptr → Link → Link means 2000. so now ptr is 2000.

ptr = ptr → Link ; Here ptr → Link means 3000. so now ptr is again 3000.

printf ("%d", ptr → data) Now it will print the data at the address ptr (i.e 3000)

Hence

Note: In Linked List we will always avoide the situation like —

NULL →

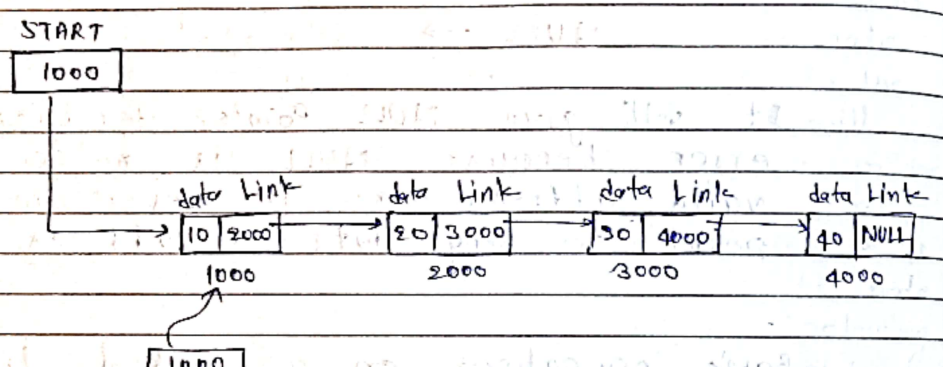
It will give NULL Pointer dereferencing error. because NULL is not a valid address so it never can point to any data or address.

Basic operations on a linked list:

There are various operations we can perform on a linked list as —

1. **Traversal.** That means हर element पर एक बार जाना (या Access करना) (या हर element को print करना)

1. Traversal:




```
struct Node *ptr;
ptr = START;                    //ptr = 1000

printf("%d", ptr -> data);      o/p -> 10

ptr = ptr -> Link               //ptr = 2000

printf("%d", ptr -> data);      o/p -> 20

ptr = ptr -> Link;              //ptr = 3000

printf("%d", ptr -> data);      o/p -> 30

ptr = ptr -> Link;              //ptr = 4000
printf("%d", ptr -> data);      o/p -> 40

ptr = ptr -> Link;              //ptr = NULL
```

Now ptr value is NULL

Note: ये steps हमें तब तक करना है, जब तक ptr = NULL न हो जाये।

So this traversal machanism can be implemented using while loop as —

```
struct Node *ptr;
ptr = START;

while(ptr != NULL)
{
printf("%d", ptr -> data);
ptr = ptr -> Link;
}
```

Note: Here START pointer is global that means it can be accessed from anywhere or any function in the program.

So the above code is actually like —

```
struct Node
{
    int data;
    struct Node * Link;
};
    * START = NULL;

void Traverse ( )
{
    ---
    ---
    ---
}
```

Ques: WAP to count no. of nodes in a linked list:

Sol:

```
int count_Node ( )
{
    int count = 0;
    struct Node *ptr = START;
    while (ptr != NULL)
    {
        count ++;
        ptr = ptr -> Link;
    }
    return count;
}
```

Ques: Given a linked list. print last node data, if exists otherwise just return (i.e. don't do any thing).

Sol: Note: Given a linked list ~~मतलब start~~ START pointer is given That means address of first node is given. (or known to us)

```
struct Node * ptr = START;
if (ptr == NULL)
    return;
while (ptr -> Link != NULL)
    ptr = ptr -> Link;
printf ("%d", ptr -> data);
```

Ques: Given a linked list, print second last node data. otherwise just return.

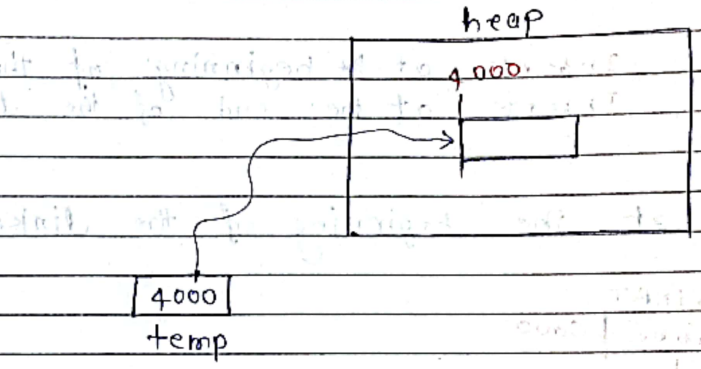
sol:
```
struct Node *ptr = START;
if (ptr == NULL || ptr -> link == NULL)
    return;
while (ptr -> Link -> Link != NULL)
    ptr = ptr -> Link;
printf ("%d", ptr -> data);
```
(ptr == NULL : for 0 node; ptr -> link == NULL : for 1 node)

2. Insertion in a linked list:

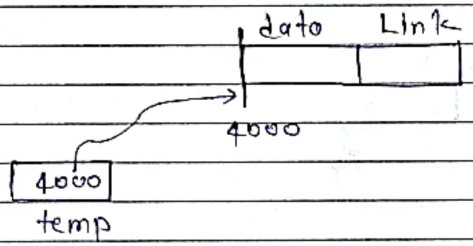
To insert a node into a linked list first of all we have to create that node with the help of malloc() as—

step 1 ① Creating a node:

```
struct Node *temp;
temp = malloc(sizeof(struct Node));
```



step 2 ② Put the data into the node:



Here temp : Pointer to a node / or Pointer to a structure

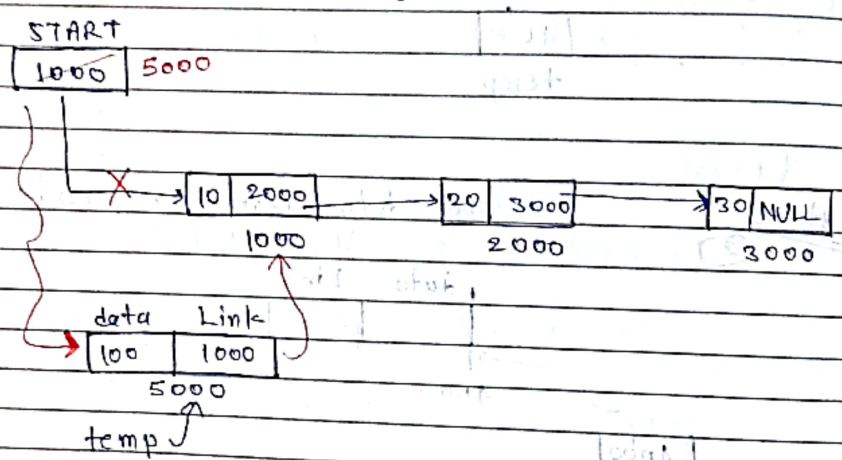
Now
```
temp -> data = x;
temp -> Link = NULL;
```

step 3 ③ Insert this node to the linked list:

we can insert this node to the linked list —

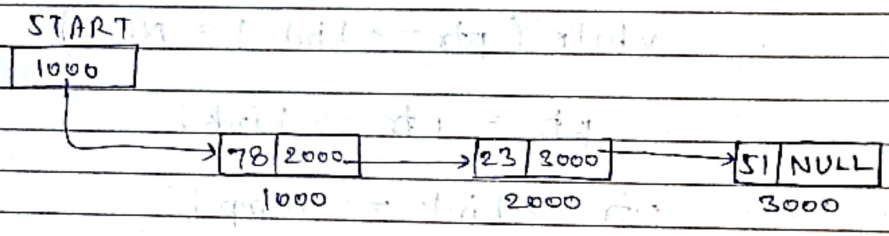
A. Insert at the beginning of the linked list.
B. Insert at the end of the linked list.

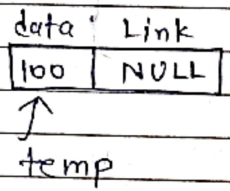
A. at the beginning of the linked list:




```
void Insert (int x)
{
    struct Node * temp;
    temp = malloc (sizeof (struct Node));
    if (temp != NULL)
    {
        temp → data = x;
        temp → Link = START;
        START = temp;
    }
}
void main ( )
{
    Insert (100);
}
```

B. At the ending of the linked list:






```
void Insert (int x)
{
    struct Node * ptr = START;
    struct Node * temp;
    temp = malloc (sizeof (struct Node));
    if (temp == NULL)
```

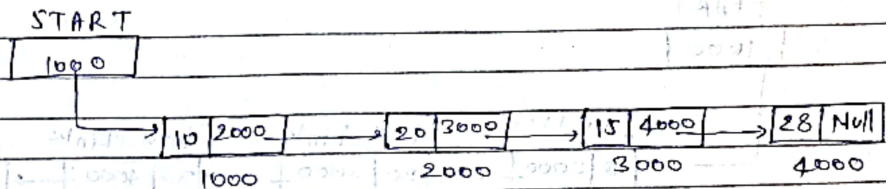
```
temp -> data = x;
temp -> Link = NULL;

if (START == NULL)
{
  START = temp;
  return;
}

while (ptr -> Link != NULL)
  ptr = ptr -> Link;
ptr -> Link = temp;
}
```

Note: Here in all the implementations we have assumed that START is a global variable.

3. Search a key in the Linked list:

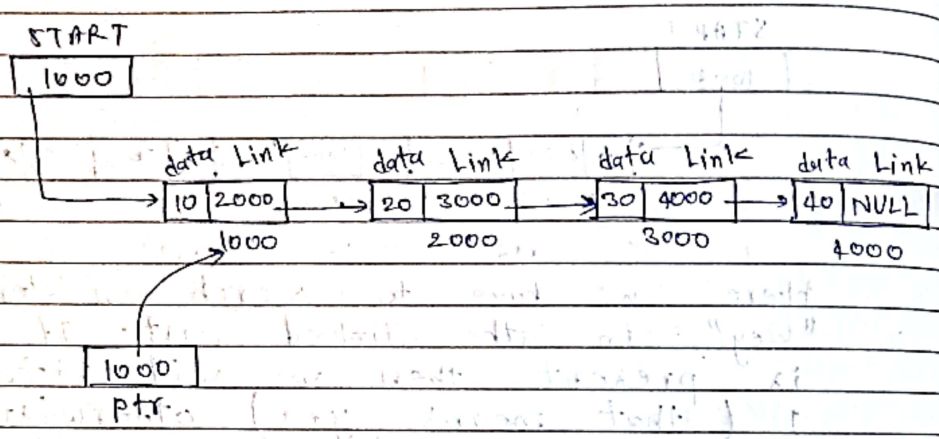


Here we have to search an element "key" in the linked list. If it is present then we will return 1 (that means yes) otherwise 0 (that means no) will be returned.

```
void search (int key) {
struct Node * ptr = START;

while (ptr != NULL)
{
  if (ptr -> data == key)
    return 1;
  ptr = ptr -> Link;
}
return 0;
}
```

4. Deletion in a linked list:



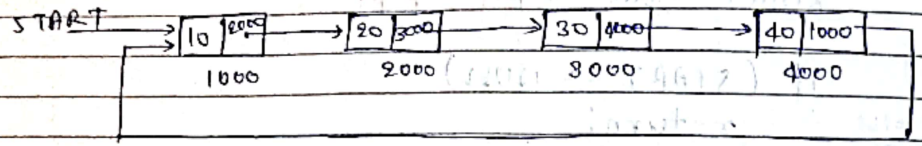
A. Delete the First Node:

```
struct Node * ptr = START;
if (START == NULL)
    return;
START = START -> Link;
free (ptr);
```

B. Delete the Last Node:

```
struct Node *ptr, *pre;
if (START == NULL)
    return;
else if ((START) -> Link == NULL)
{
    ptr = START;
    START = NULL;
    free (ptr);
}
else
{
    pre = START
    ptr = START -> Link;
    while (ptr -> Link != NULL)
    {
        pre = ptr
        ptr = ptr -> Link;
    }
    pre -> Link = NULL;
    free (ptr);
}
}
```

Circular Linked List:



In circular Linked List NULL pointer of last node is replaced by address of first node.

Advantage:

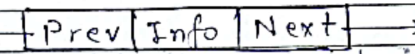
- We can traverse all nodes starting from any given node
- Nodes can be accessed easily.
- Deletion of a node is much easier.

Disadvantage:

- We may enter in an infinite loop.
- HEAD / START node is required to indicate the START or END of the circular linked list.
- Back traversing is not possible.

Doubly Linked List:

Each node in a doubly linked list contains 3 fields —



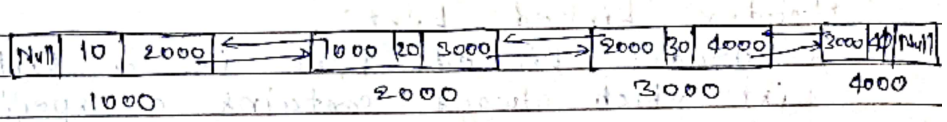
- Info feild contains the data.
- prev feild is a pointer that contains the address of previous node.
- Next feild is a pointer that contains the address of Next node.

Advantage:

It provides Bi-directional Traversing.

Types of Doubly Linked List:

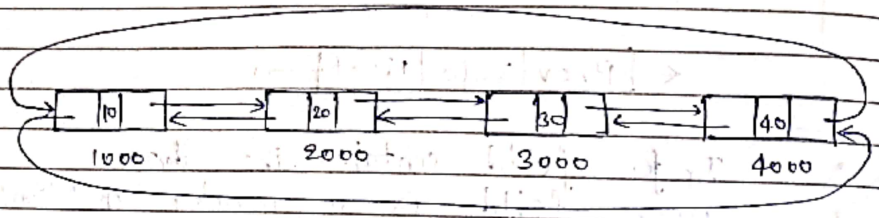
1. Linear Doubly Linked List:



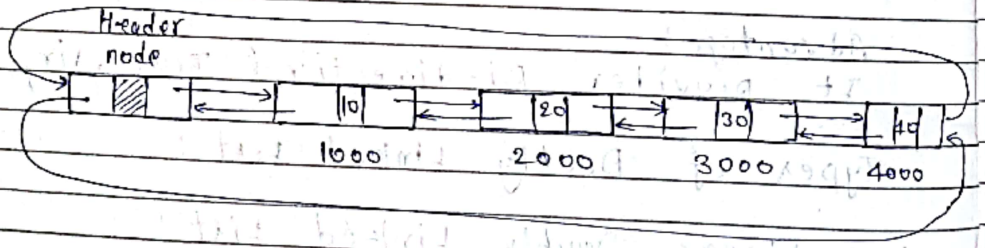
i.e.

2. Circular Doubly Linked List :

A. without a Header :



B. With a Header :

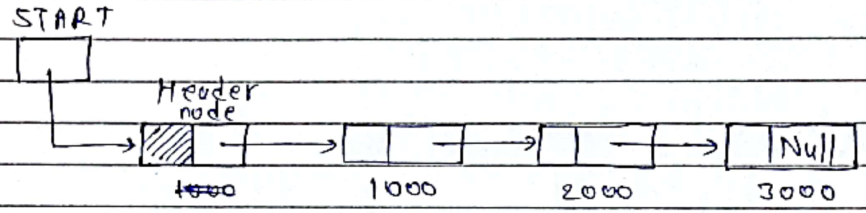


Header Linked List:
A Header Linked List is a Linked List which always contains a special node, called the Header node, at the beginning of the list.

The following are two kinds of Header lists.

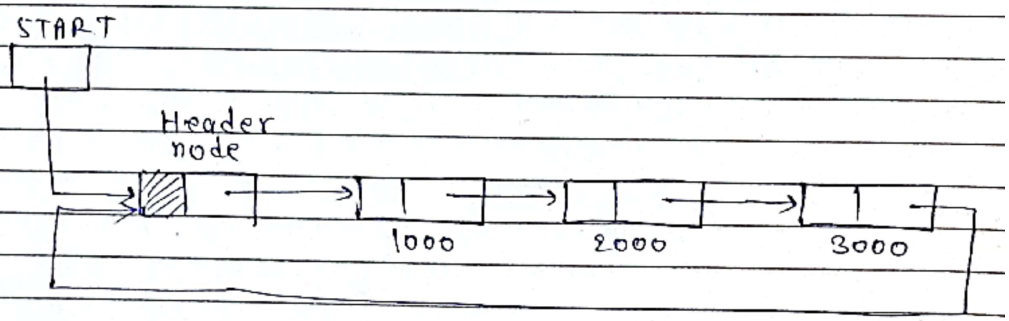
A. A Grounded Header List:

Here the last node contains the Null pointer.



B. Circular Header List:

Last node points back to the Header Node.



Date : Page :

Q: What is Data Structure?

Ans: How to organise the data in such a way that we can retrieve/manipulate it efficiently in future.

Q: What are the various types of Data Structure.

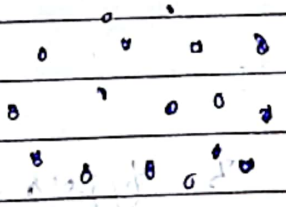
Ans: There are two types of data structure →

1. Linear Data Structure
2. Non-Linear Data Structure

Linear Data Structure: (at most 2 neighbours)

. . . . . .

Non Linear Data Structure: (any no. of neighbours)



Data Structure

Linear Data Structure

- Array
- Linked List
- Stack
- Queue

Non-Linear Data Struc.

- Tree
- Graph

So, we will cover the following 7 major topics in entire Data Structure –

1. Array
2. Linked List
3. Stack
4. Queue
5. Tree ** (most Imp)
6. Graph
7. Hashing

Array

There are generally 3 types of array.

1. Single Dimentional Array.
2. Double Dimentional Array.
3. Multi Dimentional Array.

Note: In all the following examples we are going to consider integer size as 4 bytes.

① Single Dimentional Array: (1 row, n columns)

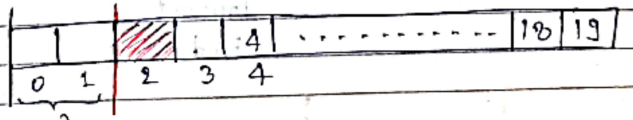
⇒ A[7]

it is an array of 7 size, that means there are 7 elements in the array and indexing starts from 0 to 6.

Note: Generally array index starts from 0 but it not mendatory theoretically and starting index can be anything in Data Structure.

Ques: Lets we have an array A[20]. Each block size is 4 bytes. then find the address of A[2]. Given that base address is 1000.

Sol: int A[20]



Elements filled before A[2] = 0 to 1
= last - First + 1
= 1 - 0 + 1
= 2 element

So, we will cover the following 7 major topics in entire Data Structure :-

1. Array
2. Linked List
3. Stack
4. Queue
5. Tree ** (most Imp)
6. Graph
7. Hashing

Array

There are generally 3 types of array.

1. Single Dimentional Array.
2. Double Dimentional Array.
3. Multi Dimentional Array.

Note : In all the following examples we are going to consider integer size as 4 bytes.

① Single Dimentional Array : (1 row, n columns)

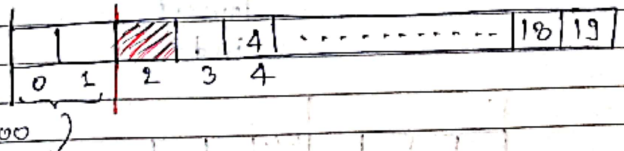
⇒ A[7]

it is an array of 7 size, that means there are 7 elements in the array and indexing starts from 0 to 6.

Note : Generally array index starts from 0 but it not mendatory theoretically. and starting index can be anything in Data Structure.

Ques : Lets we have an array A[20]. Each block size is 4 bytes. then find the address of A[2]. Given that base address is 1000.

Sol : int A[20]



Elements filled before A[2] = 0 to 1
= last - First + 1
= 1 - 0 + 1
= 2 elements

Each element size is 4 byte. (given)

Hence, memory already filled before A[2]
= 2 × 4 bytes
= 8 bytes

Hence address of A[2] is = base address + 8
= 1000 + 8 byte
= 1008 byte

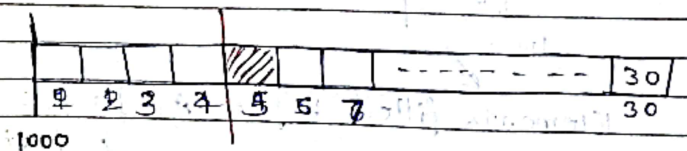
Ans

Ques: Let we have an array A[1 ..... 30]. Base address is 1000 and size of each element is 4 bytes. Find the address of A[5]

Sol: Given A[1 ..... 30]

1 → lowest index

30 → highest index.



Elements filled before A[5] is = 1 to 4

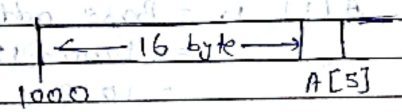
= last − First + 1
= 4 − 1 + 1
= 4 elements.

Memory already filled before A[5] = 4 × 4
= 16 byte

That means 16 bytes memory are already filled before A[5].

Hence Address of A[5] = Base address + 16
= 1000 + 16
= 1016

Ans

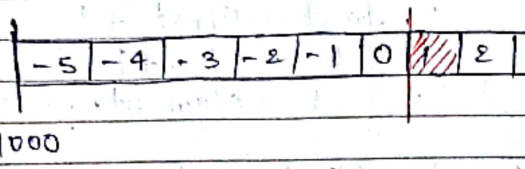


Ques: Let's we have an array A[-5 .... 5]
w = 2 bytes, Base address = 1000
Find address of A[1].

Sol: A[-5 ---- 5]

-5 → lowest index

5 → highest index



Elements filled before A[1] is = -5 to 0
= last - first + 1
= 0 - (-5) + 1
= 6 elements.

memory already filled before A[1] = 6 x 2
= 12 bytes

Hence address of A[1] is = Base address + 12
= 1000 + 12
= 1012.

② Double Dimentional Array : (2-D Array)

(m rows, n columns)

Ex: int A[2][3]

0 1 0 1 2

	0	1	2
0	A_{00}	A_{01}	A_{02}
1	A_{10}	A_{11}	A_{12}

Note: The Biggest Advantage of Array is its random accessing nature. That means we can directly go into the memory and can access data stored in any location.

Ex: printf ("%d", A[1][1]);

Ques: How we can store a 2-D array in memory ?

Ans: There are 2 ways to store a 2-D array in memory —

1. Row wise (Row Major order)
2. Column wise (Column Major order)

Date: Page:

Note: A[2][3]

Row with Index 0 →	0	A_{00}	A_{01}	A_{02}
Row with index 1 →	1	A_{10}	A_{11}	A_{12}

So row with index 0 = 3 elements
row with index 1 = 3 elements

So in A[2][3], First Dimention 2 का every element/index 3 elements को represent करता है।

i.e. A[2][3]
↳ every index in 1st dimention represents 3 elements.

(A) Representation in Row major order:

A_{00}	A_{01}	A_{02}	A_{10}	A_{11}	A_{12}
← A_0 Row →			← A_1 Row →		

Date: Page:

(B) Representation in Column major Order:

A_{00}	A_{10}	A_{01}	A_{11}	A_{02}	A_{12}
← col 1 →		← col 2 →		← col 3 →	

Ques: Represent the following ~~Ind~~ array in row major order and column major order.

A[3][4]

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

Sol: 1. in Row major order:

A_{00}	A_{01}	A_{02}	A_{03}	A_{10}	A_{11}	A_{12}	A_{13}	A_{20}	A_{21}	A_{22}	A_{23}
← 0th index →				← 1th index →				← 2nd index →			
← A_0 Row →				← A_1 Row →				← A_2 Row →			

2. in Column major order:

A_{00}	A_{10}	A_{20}	A_{01}	A_{11}	A_{21}	A_{02}	A_{12}	A_{22}	A_{03}	A_{13}	A_{23}
← col 1 →			← col 2 →			← col 3 →			← col 4 →		

Date: Page:

Ques: Given an array A[3][4] Base address is 1000. If each element size is 4 bytes then find the address of A_{22}. Elements are stored in Row major order.

Sol: How many rows are filled already before row with index 2. = index 0 row and index 1 row.

= last - First + 1
= 1 - 0 + 1
= 2 rows.

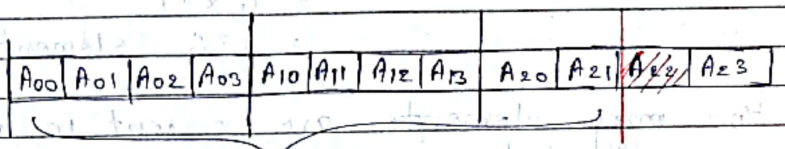
since 1 row contains 4 elements
so 2 row contains 4×2
= 8 elements.

Now how many elements are already filled in index 2 row before A_{22}.
= 0 and 1 columns.
= last - first + 1
= 1 - 0 + 1
= 2 columns/elements.

so total elements filled before A_{22}
= 2 rows + 2 elements
= 2 (4) + 2
= 8 + 2
= 10 elements.

Date: Page:

Hence total memory already filled before A_{22}
= 10 × 4
= 40 bytes.



Hence Address of A_{22} = Base address + 40
= 1000 + 40
= 1040

Ques: Given an array A[-5 ... 5][-10 ... 10]
w = 2 bytes, B.A = 1000

add(A[1][1]) = ?

Sol: How many rows are already filled before row with index 1 = index -5 row to index 0 row
= last - first + 1
= 0 - (-5) + 1
= 6 rows

Each row contains how many elements = -10 to 10
= last - fi

= 10 - (-10) + 1
= 21

So 6 rows will contain how many elements
= 6 X 21
= 126 elements.

How many elements are present in index row with index 1 before element A[1][1]
= -10 to 0
= last - first + 1
= 0 - (-10) + 1
= 11 elements

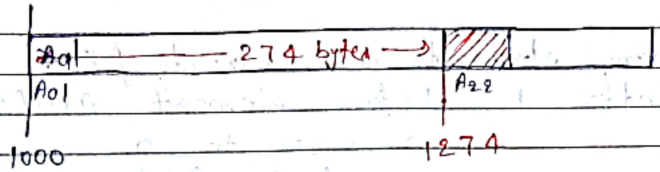
So total elements present already filled before A_{11} is = 126 + 11
= 137 elements.

Since each element takes 2 bytes
Hence 137 elements will take = 137 X 2
= 274 bytes.

So memory already filled before element A_{22}
= 274 bytes.

Hence address of A_{22} = Base Address + 274
= 1000 + 274
= 1274

Ans



Ques: Let's we have an array A[-10..10][-20..20]
w = 2 bytes . B.A = 1000
add (A[1][1]) = ?

Sol: How many rows are already filled before row with index 1 = row with index -10 to row with index 0

= last - first + 1
= 0 - (-10) + 1
= 11 rows

Each row contains how many elements
= index -20 to index 20
= last - first + 1
= 20 - (-20) + 1
= 41 elements.

So, 11 rows will contain how many elements
= 11 X 41
= 451 elements.

How many elements are already filled in row with index 1 before element A[1][1]

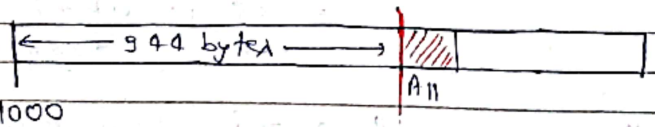
= -20 to 0
= last - first + 1
= 0 - (-20) + 1
= 21 elements.

So total elements already filled before element A_{11} is = 451 + 21
= 472 elements

Hence total memory already filled before element A_{11} is = 472 × 2
= 944 bytes.

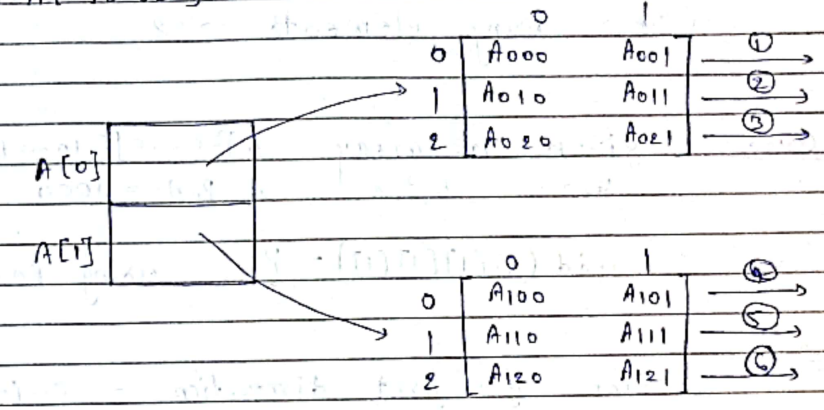
So Address of element A[1][1]
= Base address + 944
= 1000 + 944
= 1944

Ans



3. Three Dimentional Array : (3-D Array)

Ex : A[2][3][2]



in row major order we will store the elements of above matrix (array) in the following way -

A000	A001	A010	A011	A020	A021	A100	A101	A110	A111	A120	A121

Note : in A[2][3][2]

Each number in first dimention represents how many elements = 3 × 2
= 6 elements

Hence size of A[0] = size of A[1] = 3 × 2

in A[2][3][2]

Each number in 2nd dimention represents how many elements = 2

Ques: given an array A[-5 ... 5][-10 ... 10][-5 ... 5]
w = 2 bytes, B.A. = 1000

add (A[1][1][1]) = ? using Row Major Order

Sol:

size of first dimention = 5-(-5)+1 = 11
" " 2nd " = 10-(-10)+1 = 21
" " 3rd " = 5-(-5)+1 = 11

Every index/number in 1st Dimention represents how many elements = 21 X 11
= 231 elements

Every index/number in 2nd Dimention represents how many elements = 11 elements

How many elements are already filled before element A[1][1][1]
= A[-5] matrix of size 21X11 +
= A[-4] " " " " +
= A[-3] " " " " +
A[-2] " " " " +
A[-1] " " " " +
A[0] " " " " +

row with index -10 in A[1] matrix +
row with index -9 " " " +
row with index -8 " " " +

row with index 0 " " " +

elements index from column -5 to column 0 in A[1] matrix and row with index 1.

= (21 X 11) X 6 + (11) X 11 + 6
= 1386 + 121 + 6
= 1513

since each element contains 2 bytes
Hence 1513 " " = 1513 X 2
= 3026 bytes

so address of A[1][1][1] = Base address + 3026
= 1000 + 3026
= 4026
= 4026 Ans

Column major Order:

Ex: A[2][3]

	0	1	2
0	A_{00}	A_{01}	A_{02}
1	A_{10}	A_{11}	A_{12}

A_{00}	A_{10}	A_{01}	A_{11}	A_{02}	A_{12}
col. with index 0		col. with index 1		col. with index 2	

Ques: A[-5 -- 5] [-10 -- 10]

w = 2 bytes, B. Address = 1000

add (A[1][1]) = ? consider column major order.

Sol:

	-10	-9	-8	...	1	...	10
-5							
-4							
-3							
1					•		
4							
5							

How many columns filled already before A[1][1]

= -10 index col to 0 index col

= last - first + 1

= 0 - (-10) + 1

= 11 columns.

Each column have how many elements.

= -5 index to 5 index

= last - first + 1

= 5 - (-5) + 1

= 11 elements.

How many rows filled before A[1][1]

= index -5 to index 0

= last - first + 1

= 0 - (-5) + 1

= 6

So total How many elements filled before element A[1][1]

= 11 x 11 + 6

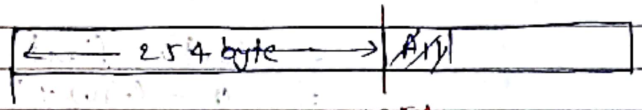
= 127.

Size of each element = 2 bytes.

So size of 127 elements = 127 X2

= 254 bytes.

That means 254 bytes are already filled before element A[1][1].



Hence address of element A[1][1]
= Base address + 254
= 1000 + 254
= 1254.

Ans

Ques: A[-5...5][-10...10][-5...5]
is = 2 bytes
Base Address = 1000
add (A[1][1][1]) = ? Consider — column major order.

Sol: Given
A[-5...5][-10...10][-5...5]

⇒ A[11][21][11]

in A[-5...5][10...10][-5...5]

every no. will represent = 21 × 11 element

every no. will represent 11 elements

Now in A[-5...5][-10...10][-5...5] we have to find the address of A[1][1][1]

so how many elements will be filled already in memory before A[1][1][1] is —

A[1][1][1]

① -5 to 0
= 0 − (-5) + 1
= 6

② -10 to 0
0 − (-10) + 1
= 11

-5 to 0
0 − (-5) + 1
6

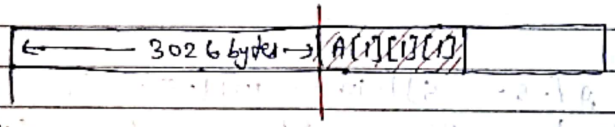
so 11 × 1
11 × 1 = (11)

so 6 × 21 × 11
= 1386

Hence total elements already filled
= 6 × 21 × 11 + 11 × 11 + 6 = 1513

since size of each element = 2 bytes
so size of 1513 " = 2 X 1513
= 3026 bytes.

so total memory already filled before element A[1][1][1] is —



Hence address of A[1][1][1] = Base Add + 3026
= 1000 + 3026
= 4026

Ans

Triangular Arrays:

① Lower Triangular Array:

A[5][5]

	0	1	2	3	4
0	A_{00}	0	0	0	0
1	A_{10}	A_{11}	0	0	0
2	A_{20}	A_{21}	A_{22}	0	0
3	A_{30}	A_{31}	A_{32}	A_{33}	0
4	A_{40}	A_{41}	A_{42}	A_{43}	A_{44}

② Upper Triangular Array:

A[5][5]

	0	1	2	3	4
0	A_{00}	A_{01}	A_{02}	A_{03}	A_{04}
1	0	A_{11}	A_{12}	A_{13}	A_{14}
2	0	0	A_{22}	A_{23}	A_{24}
3	0	0	0	A_{33}	A_{34}
4	0	0	0	0	A_{44}

How to store upper / lower triangular arrays in memory:
while storing such type of arrays in memory efficiently then no need to store zeros in memory.

Ex: Lets store the above lower triangular matrix in memory in row major order.

so it will be stored as —

A_{00}	A_{10}	A_{11}	A_{20}	A_{21}	A_{22}	A_{30}	A_{31}	A_{32}	A_{33}	A_{40}	A_{41}	A_{42}	A_{43}	A_{44}

so lets try to check (find) the address of any element (A43 for example)

A_{43}.

So how many elements are filled before the element A[4][3] (i.e. A43)

A 4 3 → 0 to 2

↓

0 to 3

last - first + 1

3 - 0 + 1

= 4 rows

last - first + 1

2 - 0 + 1

= 3 elements

So 3 elements filled before A43

So 4 rows filled before A43

and in these 4 rows

1st row contains = 1 element

2nd " " = 2 "

3rd " " = 3 "

4th " " = 4 "

So total elements filled in these 4 rows =

= 1 + 2 + 3 + 4

= 10 elements

Hence total elements filled before A43

= 10 + 3

= 13 elements.

since each element contains = 2 bytes

so 13 " " = 13 × 2

= 26 bytes.

Hence address of A43 = Base Address + 26

= 1000 + 26

= 1026

Ans

Major operations performed on any linear Data Structure like Array, linked list:

① Traversal: Processing each element in the list.

② Search: Finding the location of an element.

③ Insertion: Adding a new element to the list.

④ Deletion: Removing an element from the list.

⑤ Sorting: Arranging the elements in some type of order (i.e. increasing/decreasing)

⑥ Merging: combining two lists into a single list.

Formula for finding address of any element A[i][j] in a two-dimentional array A[m][n] :

① In Row major order:

$$\text{Address of } A[i][j] = B + w\left(n(i-L_1)+(j-L_2)\right)$$

② In column major order:

$$\text{Address of } A[i][j] = B + w\left((i-L_1)+m(j-L_2)\right)$$

Here
- B = Base Address
- w = Element size
- m = upper bound of rows
- n = upper bound of columns
- L_1 = lower bound of rows
- L_2 = lower bound of columns
- m = no. of rows
- n = no. of columns.

Ques: How is physically memory allocated for a two-dimentional array? if each element of an array D[20][50] requires 4 bytes of storage, Base address of data is 2000. Determine the location of D[10][10], when the array is stored as -
i). Row major order
ii). Column major order.

sol: i). Row major order :

address of D[10][10] $= B + w\left(n(i-L_1)+(j-L_2)\right)$

$= 2000 + 4\left(50(10-0)+(10-0)\right)$

$= 2000 + 4(510)$

$= 4040$

ii): Column major order :

address of D[10][10]

$= B + w\left((i-L_1)+m(j-L_2)\right)$

$= 2000 + 4\left((10-0)+20(10-0)\right)$

$= 2000 + 840$

$= 2840$