:
:
:

data-type member ;
};

Advantages of structure :-
- It can hold variables of different data types.
- We can create objects containing different types of attributes.
- It allows us to re-use the data layout across programs.
- It is used to implement other data structure like linked list, queues, trees and graphs.

Program :-
how to use structure in program →

```
#include <stdio.h>
#include <conio.h>
Void main ()
{
struct employee
{
int id ;
float salary ;
int mobile ;
};
```

```
struct employee e1, e2, e3;
printf ("\n Enter ids, salary & mobile no. \n");
scanf ("%d %f %d", &e1.id, &e1.salary, &e1.mobile);
scanf ("%d %f %d", &e2.id, &e2.salary, &e2.mobile);
printf ("%d %f %d", &e3.id, &e3.salary, &e3.mobile);
printf ("\n Entered result");
printf ("\n %d %f %d", e1.id, e1.salary, e1.mobile);
printf ("\n %d %f %d", e2.id, e2.salary, e2.mobile);
printf ("\n %d %f %d", e3.id, e3.salary, e3.mobile);
getch ();
}
```

output →

guess the output

And write it here....

Array :- Arrays are defined as collection of similar type of data items stored at contigous memory locations.

Array is the simplest data structure where each data element can be randomly accessed by using its index number.

Array declaration :-

```
int arr[10] ; char arr[10] ; float arr[5]
```

Program without Array :-

```
#include <stdio.h>
void main ()
{
 int marks-1 = 56 ; marks-2 = 78, marks-3 = 89 ;
 float avg = (marks-1 + marks-2 + marks-3)/3 ;
 print (avg) ;
}
```

Program by using Array :-

```
#include <stdio.h>
void main
 {
 int marks [3] = { 56, 78, 89} ;
 int i ;
 float avg ;
 for (i=0 ; i<3 ; i++)
```

```
    {
    avg = avg + marks[i];
    }
  printf(avg);
}
```

Complexity of Array operations :-

1). Time complexity :-

Algorithm	Average case	worst case
Access	O(1)	O(1)
search	O(n)	O(n)
insertion	O(n)	O(n)
Deletion	O(n)	O(n)

2). Space complexity :-

In Array space complexity for worst case is O(n)

Memory Allocation of the Array :-

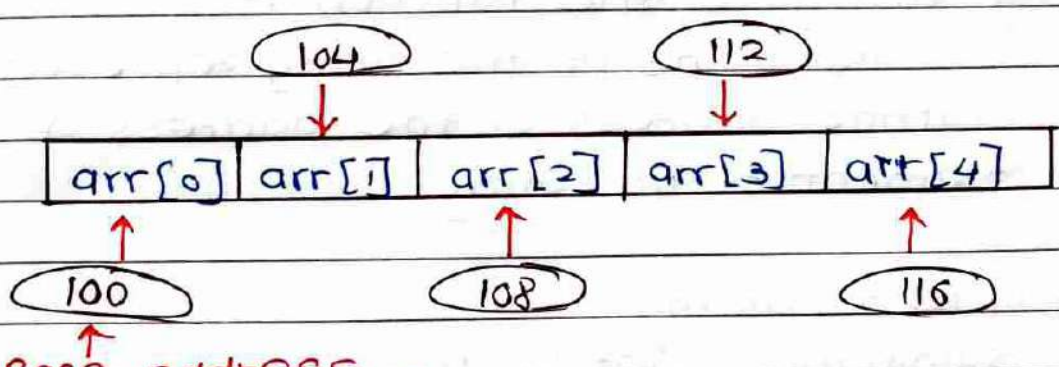
Each element in Array represented by indexing Indexing of array can be defined in three ways:

1. 0 (Zero Based indexing) :-

The first element of the array will be arr[0].

2. 1 (one-based indexing) :-
The first element of array will be arr[1].

3. n (n-based indexing) :-
The first element of array can reside at any random index number.



(fig : int arr[5])

Accessing elements of an Array :-
To access any random element of an array we need the following information :
1. Base address of the array
2. size of an element in bytes.
3. which type of indexing, array follows.

Address of any element of 1D array can be calculate

Byte address of element A[i] = base address + size * (first - index)

Example: In an array, A[-10 .... +2] Base address (BA) = ggg, size of an element = 2 bytes, find location of A[-1].

solution: $L(A[-1]) = ggg + [(-1) - (-10)] \times 2$.

$= ggg + 18$

$= 1017$.

$\therefore$ location of $A[-1] = 1017$.

Passing array to the function :-

The name of the array represents the starting address or the address of the first element of the array.

Program:

```
#include <stdio.h>
int summation (int []);
void main ()
{
 int arr [5] = {0, 1, 2, 3, 4};
 int sum = summation (arr);
 printf ("%d ", sum);
}
int summation (int arr[])
 {
  int sum = 0, i;
  for(i=0 ; i<5 ; i++)
    {
     sum = sum + arr [i];
    }
  return sum ;
 }
```

2D Array :- 2D array can be defined as an array of arrays. The 2D array is organized as matrices which can be represented as collection of rows and columns.

How to declare 2D Array :-

The syntax for declaration of two dimensional array is as follows :

```
int arr [max-rows][max-columns];
```

However, it produces the data structure which looks like following :

	0	1	2	...	n-1
0	a[0][0]	a[0][1]	a[0][2]	.....	a[0][n-1]
1	a[1][0]	a[1][1]	a[1][2]	....	a[1][n-1]
2	a[2][0]	a[2][1]	a[2][2]	...	a[2][n-1]
:	:	:	:	...	:
n-1	a[n-1][0]	a[n-1][1]	a[n-1][2]		a[n-1][n-1]

a[n][n]

(fig : a[n][n])

How to access data in 2D-array :-

Due to fact that elements of 2D arrays can be random accessed.

```
int x = a[i][j];
```

where i, j are the rows and columns respectively.

Initializing 2D arrays :-

The syntax to declare and initialize the 2D array is given as follows :

```
int arr[2][2] = {0,1,2,3};
```

number of elements in 2D arrays
= number of rows * number of columns.

Mapping 2D array to 1 D array :-

The size of a two dimensional array is equal to the multiplication of number of rows and number of columns present in the array.

A 3x3 two dimensional array is as shown:-

	0	1	2
0	(0,0)	(0,1)	(0,2)
1	(1,0)	(1,1)	(1,2)
2	(2,0)	(2,1)	(2,2)

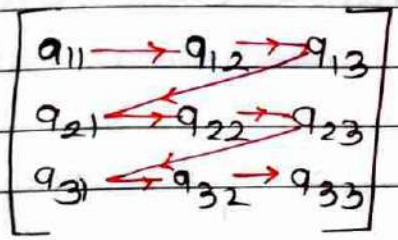
column index

row index

There are two main techniques of storing 2D array elements into memory.

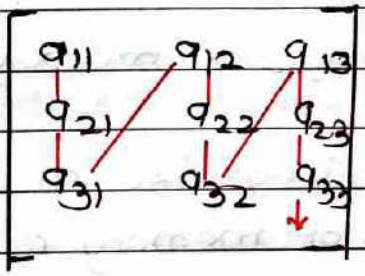
1. Row major ordering :—

In row major ordering, all the rows of 2D array are stored into memory contiguously.



2. Column major ordering :—

According to column major ordering, all the columns of 2D array are stored into the memory contigously.



Calculating address of random element of a 2D array:—

1). By row major order :—

If array is declared a[m][n] where m is the number of rows while n is number of columnes. then address of an element a[i][j] is calculated as,

Address (a[i][j]) = B.A + (i * n + j) * size

B.A → Base Address

2). By column major order :—

Address (a[i][i] = (j * m) + i) * size + B.A.

Linked list :-

Why there is a need of linked list ?

If we declare an array of size 3. As we know that all the values of an array are stored in a continous manner, so all three values of an array are stored in a sequential fashion.

Then, total memory space occupied by array would be $3 * 4 = 12$ bytes.

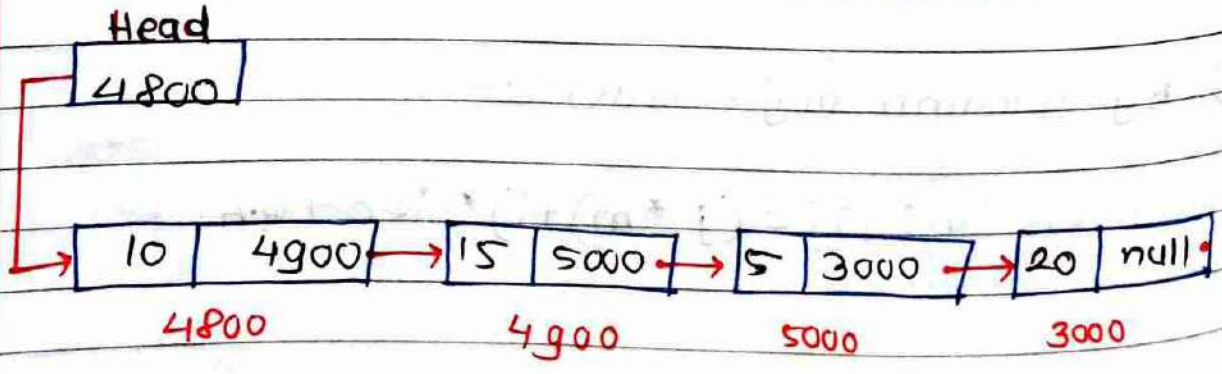
Drawbacks of using array :-

- we cannot insert more than 3 elements in above example because only 3 spaces are allocated by 3 elements.
- In case of array, the wastage of memory can occur.
- In array, we are providing fixed-size at compile time, due to which wastage of memory occurs.

The solution to this problem is to use linked list.

What is Linked list ?

A linked list is also a collection of elements, but the elements are not stored in a consecutive location. or linked list is a collection of the nodes in which one node is connected to another node and node consists of two parts i.e. one is data part and second one is the address part.



declaration of linked list :-

In linked list, one is variable and second one is pointer variable. we can declare linked list by using user-defined data type called as structure.

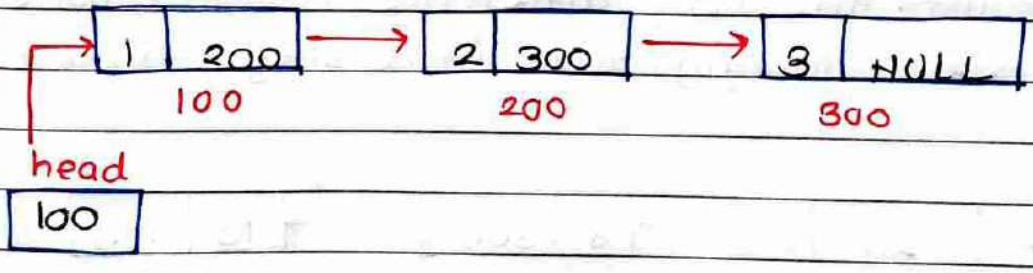
```
struct node
{
  int data;
  struct node *next;
}
```

Types of linked list :-

1). singly linked list :-

The singly linked list is most common. which consists of data part and address part. The address part in the node is known as a pointer.

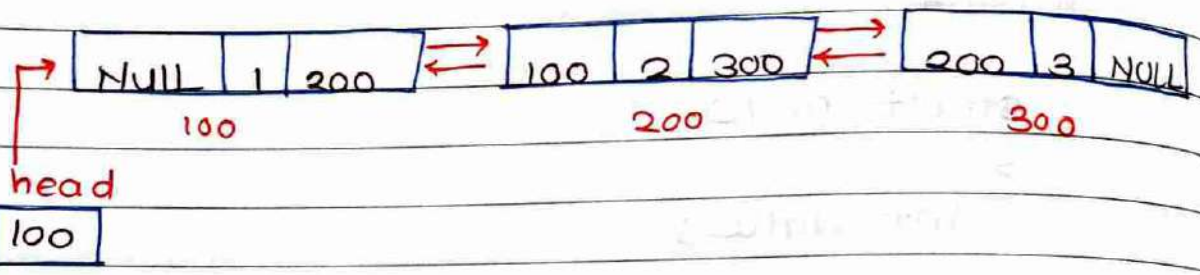
Example :- suppose we have three nodes and addresses of these three nodes are 100, 200 and 300 :



NULL means its address part does not point to any node. The pointer that holds the address of the initial node is known as a head pointer.

2. Doubly linked list :-

As name suggests, the doubly linked list contains two pointers. We define it in three parts the data part and the two address part.

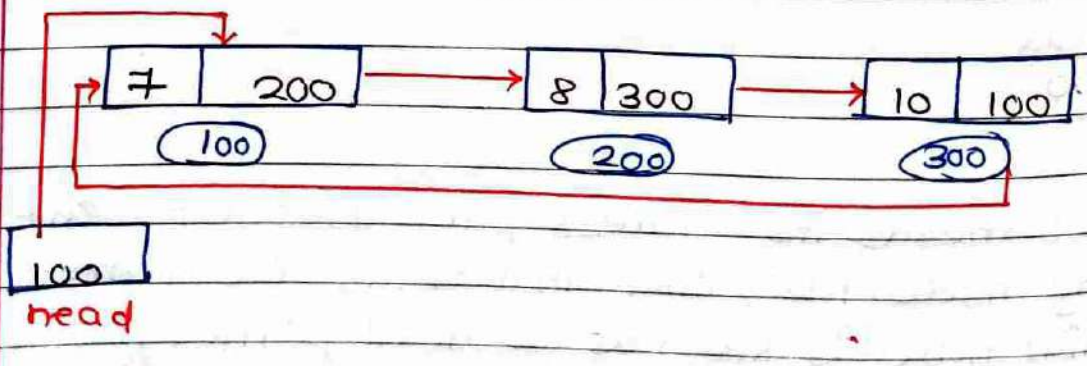


Representation of doubly linked list :-

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

3. Circular linked list :-

A circular linked list is a variation of a singly linked list. The only difference is "last node does not point to any node in a singly linked list".

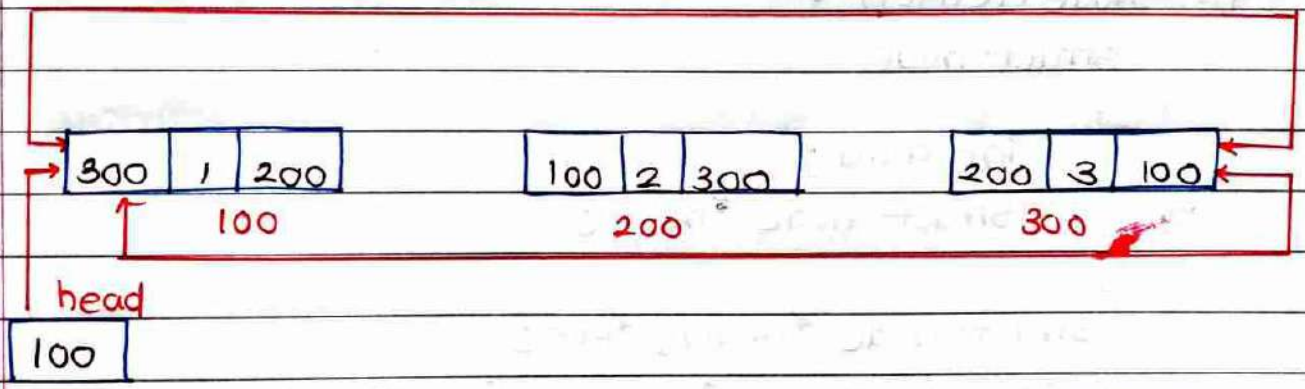


Representation of circular linked list :-

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

4. Doubly circular linked list :-

The doubly circular linked list has the features of both the circular linked list and doubly linked list.



The last node is attached to the first node and thus creates a circle.
The main difference is that doubly circular linked list does not contain NULL value in previous field of the node.

Representation of doubly circular linked list :-

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

Complexity :-

	Average				Space complexity
	Access	search	Insertion	deletion	Worst
Singly linked list	O(n)	O(n)	O(1)	O(1)	O(n)
	Worst				
	Access	search	Insertion	deletion	
singly linked list	O(n)	O(n)	O(1)	O(1)	

Operations on singly linked list :-

1). Node creation :-

```
struct node
{
    int data;
    struct node *next;
};
struct node *head, *ptr;
ptr = (struct node*) malloc (sizeof (struct node*))
```

2). Insertion :-

① Insertion at beginning :- It involves inserting any element at the front of the list. We just need a few link adjustment to make new node as head of list.

② Insertion at end of list :- The new node can be inserted as the only node in the list / it can be inserted as last one.

③ Insertion after specified node :- we need to skip desired number of nodes in order to reach node after which the new node will be inserted.

3). 3) Deletion and Traversing :-

① Deletion at beginning :- It just needs few adjustments in the node pointers

② Deletion at end of list :- The list can either be empty or full. Different logic is implemented for different scenario's.

Traversing :- In traversing, we simply visit each node of the list at least once in order to perform some specific operation in it, for example, printing data part of each node present in the list.

Searching :- In searching, we match each element of the list with the given element. If the element is found on any of the location of that element is returned otherwise null is returned.

operations on doubly linked list :-

1). Node creation :-

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

2). Insertion :-

① Insertion at beginning :- Adding the node into the linked list at beginning.

② Insertion at end :- Adding the node into the linked list to the end.

3). Deletion and Traversing :-

① Deletion at beginning :- Removing the node from beginning of the list.

② Deletion at end :- Removing the node from end of the list.

Traversing :- visiting each node of the list at least once in order to perform some specific operation like searching, sorting, display etc.

Searching :- comparing each node data with the item to be searched and return location of the item in the list if the item found else return null.

Skip list :-

* What is a skip list ?

A skip list is a probalistic data structure. The skip list is used to store a linked list of elements or data with a linked list. In one single step, it skips several elements of the entire list, which is why it is known as skip list.

Structure of skip list :-

skip list is built in two layers : The lowest layer and the top layer. The lowest layer of the skip list is a common sorted linked list, and the top layers of the skip list are the like an "express line" where elements are skipped.

complexity table :-

Sr.No.	complexity	Average case	Worst case
1).	Access complexity	$O(\log n)$	$O(n)$
2).	search comple.	$O(\log n)$	$O(n)$
3).	delete comple.	$O(\log n)$	$O(n)$
4).	Insert comple.	$O(\log n)$	$O(n)$
5).	space comple.	-	$O(n \log n)$.

Basic operations and its algorithms :-

1). Insertion operation :- It is used to add new node to a particular location in a specific situation.

2). Deletion operation :- It is used to delete a node in a specific situation.

3). search operation :- The search operation is used to search a particular node in a skip list.

Algorithm of insertion operation :-

```
Insertion (L, key)
local update [0 .... max - level + 1]
q = L → header
for i = L → level down to 0 do.
    while q → forward [i] → key forward [i]
update [i] = q
```

```
a = a → forward[0]
lvl = random_level()
if lvl > L → level then
for i = L → level + 1 to lvl do
    update[i] = L → header
    L → level = lvl
a = make node (lvl, key, value)
for i = 0 to level do
    a → forward[i] = update[i] → forward[i]
update[i] → forward[i] = a
```

Algorithm of deletion operation :-

```
Deletion (L, key)
local update[0.... max level + 1]
a = L → header
for i = L → level down 0 to do
    while a → forward[i] → key forward[i]
        update[i] = a
a = a → forward[0]
if a → key = key then
    for i = 0 to L → level do
    if update[i] → forward[i] ≠ a then break
    update[i] → forward[i] → forward[i]
free(a)
while L → level > 0 and L → header → forward[L → level]
                                    = NIL do
L → level = L → level - 1.
```

Algorithm of searching operation :-

searching (L, skey)

a = L → header

loop invariant : a → key level down to 0 do.

while a → forward [i] → key < skey forward [i]

a = a → forward [a]

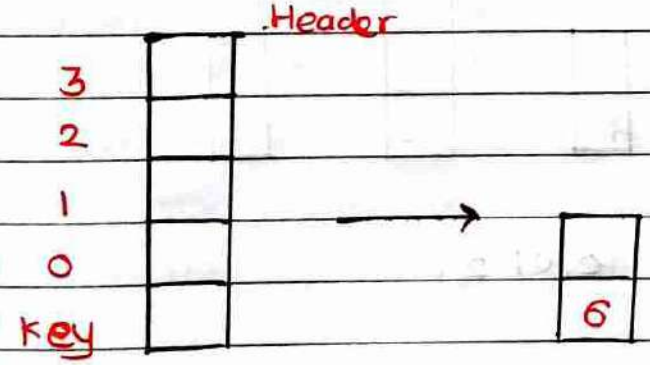
if a → key = skey then return a → value

else return failure.

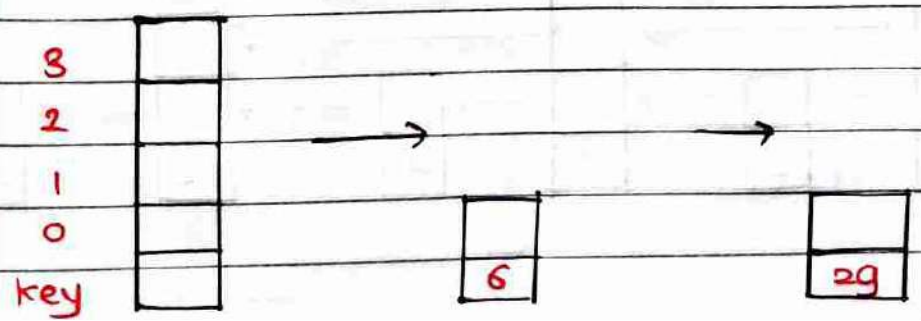
Example : create a skip list, we want to insert these following keys in empty skip list.

1. 6 with level 1
2. 2g with level 1
3. 22 with level 4.
4. g with level 3.
5. 17 with level 1.
6. 4 with level 2.

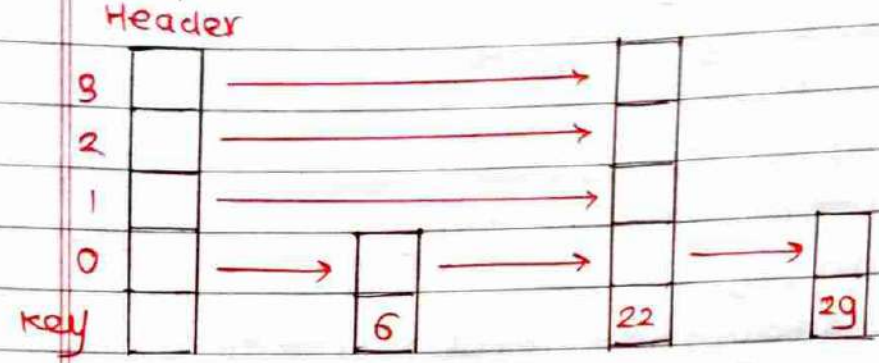
→ solution :- Insert 6 with level 1.



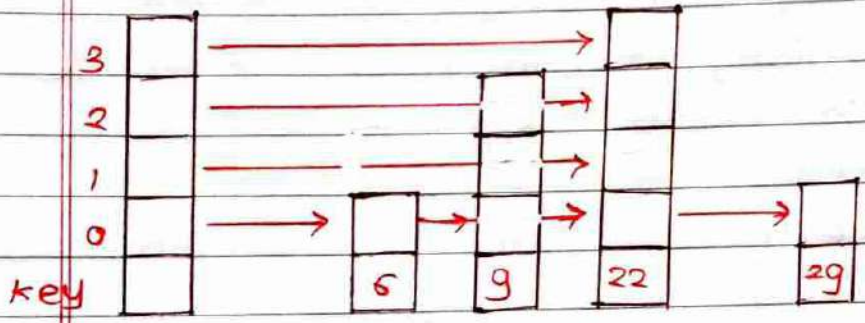
step 2 :- Insert 2g with level 1.



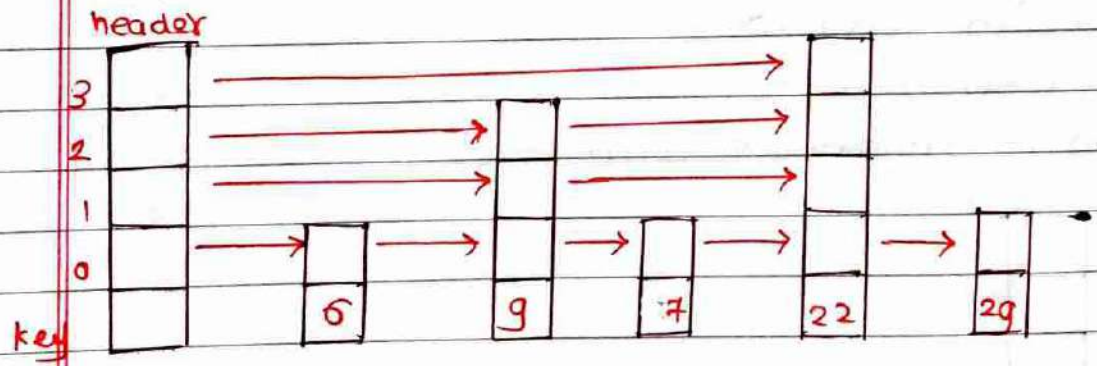
Step 3 : Insert 22 with level 4.



Step 4: Insert 9 with level 3.



Step 5: Insert 17 with level 1



Step 6: Insert 4 with level 2.