Pointer

Before learning function, array, string we should learn pointer. Because it is an important concept which plays a vital role in function, array, string.

What is a pointer?

↳ Pointer is a special variable which stores the address of another variable.

↳ A pointer points to the memory location where does another variable is present.

Note

simple variable declaration

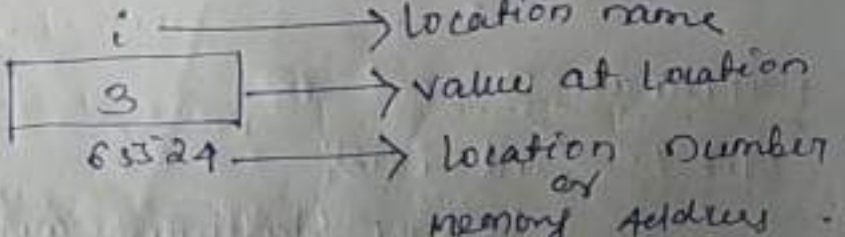
```
int i = 3;
```

Here the declaration tells the compiler to

i) reserve space in memory to hold integer value.

ii) Then give the name of memory location i.

iii) Store the value 3 at this location.



The memory address can be any thing instead of 65524 we have just taken an example.

Note

↳ Our aim is to find that location number or the memory address

↳ That memory address is always a +ve integer.

↳ To find that memory address or to work with the memory address we are learning pointer concept.

Note

There are two ways to print the address of any memory location.

Method 1: (using & operator)

&i or &variable name

ex

```
# include <stdio.h>
void main()
{
    int i = 3; //
    printf("Address of i = %u ", &i);
    printf("value of i = %d ", i);
}
```

i
3
65525

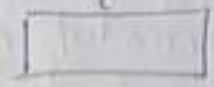
Explanation

Step 1: int i = 3;

↳ This statement will tell the computer to reserve memory to hold integer so [ ] memory location is reserved

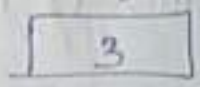
↳ The memory name will be given with i

i.e



↳ Store the value 3

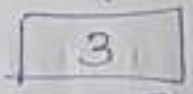
i.e



Note

for our easy understanding we have given a name i to the memory location but actually it is an address or number.

i.e



Step 2

printf (" Address of i = %u", &i);

↳ Here you can mark we have used %u generally u stands for unsigned i.e always +ve As address of memory location is always a +ve integer so we have used %u

↳ Here you can mark another thing that we have used & operator before the variable i in printf (as we know generally we don't use '&' operator in printf (" ") rather we use in scanf (" ").

↳ Since we have written &i here it means "address of i" as "&" is address of operator

↳ So it is print Address of i = 6552

Because the address of the memory Location which is associated with i is 65525.

Step 3

printf(" value of i = %d ", i);

↳ Here you can mark or have not used '&' before i. We have written only i.
only i means the variable.

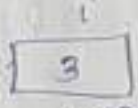
↳ So it'll print value of i = 3 .

Step 4

stop.

Note

int i = 3



i = 3 → value of i

&i = 65525 → Address of i.

So & : Address of operator returns the address of the variable associated with it.

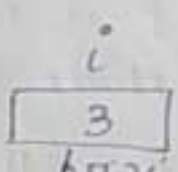
Method 2 (using & and * operator)

```
#include <stdio.h>
void main()
{
    int i = 3;
    int *P = &i; // initialization of Point
    printf(" Address of i = %u ", &i );
```

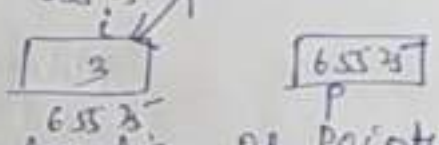
```
printf ("Address of i = %u ", P);
printf ("value of i = %d ", i);
Printf (" value of i = %d", *(&i));
Printf (" value of i = %d", *P);
}
```

Explanation

Step 1 : int i = 3; //



Step 2 : int *P = &i ;



↳ int *P is the declaration of Pointer.

↳ Here '*' is the indirection operator or value at address operator.

↳ int *P means P is a variable which stores the address of another variable and that another variable holds an integer value.
so P is a Pointer to an integer variable.

↳ int *P=&i ;
↳ &i is address of i.
↳ Address of i is assigned to int *P that means i is a variable which holds an integer value and address of variable i is stored in P. as P is a pointer variable.

Step 3: printf(" Address of i = %u ", &i);

↳ &i : Address of i

So it will print

Address of i = 65525

Step 4: printf(" Address of i = %u ", p);

↳ p is a pointer variable which stores the address of i

So o/p will be

Address of i = 65525

Step 5: printf(" value of i = %d", i);

↳ value of i = 3.

Step 6: printf(" value of i = %d", *(&i));

↳ *(&p) :-

↳ * → this is the indirection operator or value at address operator

↳ *(&i) :- value at address of i

↳ o/p :- value of i : 3,

Because address of i = 65525

value at 65525 = 3

So o/p - 3.

Step 7:- printf ("value of i = %d", *p);

↳ *p means value at address stored in p

↳ Here the address stored in p is 65525

↳ The value at address stored in p means the value at 65525.

↳ So o/p : 3.

Step 8: Stop.

Final output

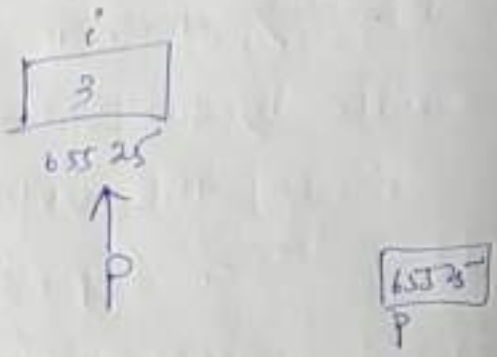
Address of i = 65525
Address of i = 65525
Value of i = 3
value of i = 3
value of i = 3

Note

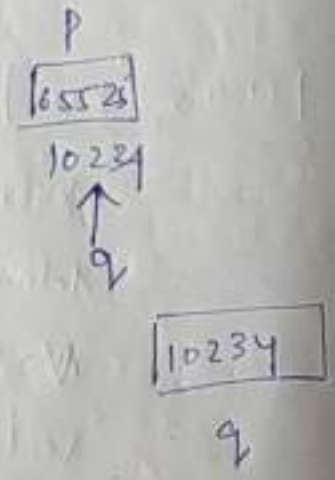
(i) int *p; // declaration of a pointer.
i.e p is a pointer which will hold the address of another variable which variable will store an integer value.
so int *p; is a pointer to an integer.
But till now it has not pointed to any memory location.

(ii) int *P = &i; // initialization of a Pointer

i.e P is a pointer which holds the address of i



iii) int **q = &P;



it means q is a special variable which stores the address of another pointer P.

iv) ↳ int *P = &i;

P is a pointer which stores the address of another variable i.

↳ int **q = &P;

q is a pointer which stores the address of another pointer P.

v) char *Ch;

Ch is a Pointer which will store the address of a variable which contains a character i.e address of a character value.

vi) float *S;

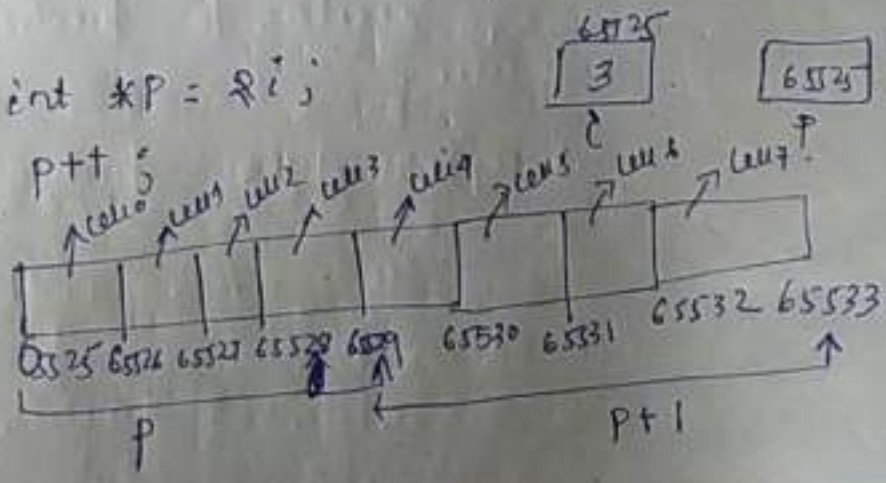
S is a pointer which will contain the address of a float value.

vii) ↳ int *P = &i; // 4 Byte will be taken by P as it is storing the address of as integer.

↳ Char c = 'a';
Char *ch = &c; // 1 Byte as ch is storing the address of a character.

↳ float f = 5.6;
↳ float *S = &f; // 4 B as S is storing the address of a float.

viii) int *P = &i;



Explanation

1) Since p is an integer pointer

So for storing an integer variable we need 4 byte. So p will take 4 B
65525 to 65528 (65525 - 65526, 65527 - 65528
65526 - 65527, 65527 - 65529)

2) if we'll increment p by 1 then it'll point to next 4 Byte.

i.e 65529 to 65532.

(65529 - 65530
65530 - 65531
65531 - 65532
65532 - 65533)

So we have just considered the starting address of each cell

Like wise for Character and float -

STRUCTURE

(1)

Suchismita Mohanty

Before going to "Structure", let's have a brief discussion about variables (Primary data types) and array (Secondary data type).

↳ As we know to store an integer value or a character value or a float type value we need a single variable of each type.

i.e if I want to store an integer value 20 there I need a variable num of Integer type

```
int num1 = 20;
```

Like wise for storing 3 integer values i.e 20, 30, 40, I need 3 variables num_1, num_2, num_3 of Integer type.

```
int num1, num2, num3;
num1 = 20, num2 = 30, num3 = 40;
```

Like wise for one float = 30.4

```
float num1 = 30.4;
```

and for 3 float values i.e 30.4, 40.4, 50.4

```
float num1 = 30.4, num2 = 40.4, num3 = 50.4
```

% means if we are going to use any Primary data type then the no. of values = the no. of variables.

↳ So to avoid the disadvantages of primary datatype (the no. of value = the no. of variable) we are going to use one of the secondary data type Array.

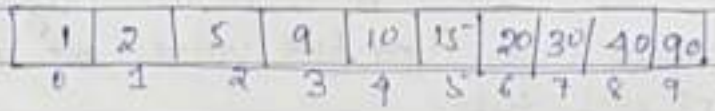
↳ Array is a Secondary data type which is used to store a collection of similar type of data in contiguous Memory location.

i.e If we want to store 10 integer type of value in memory then we need an array.

```
int arr[10] = {1, 2, 5, 9, 10, 15, 20, 30, 40, 90};
```

description

Here arr is an array variable which is going to store 10 integer type data in contiguous memory location (0 to 9 indices)



Disadvantage of using array:-

We can store only collection of similar type of data.

i.e. if we want to store a combination of integer, float, character or any other data then it is not possible through Array.

↳ So to avoid the disadvantage of array we are going to use Structure

Structure

↳ Structure is a secondary data type which is used to store collection of dissimilar data type.

↳ ex:- A book is a collection of title, author, publisher, number of pages, date of publication etc. So for this type of data C programming provides a data type called Structure.

↳ So to store real time dissimilar data we are going to use Structure.

Syntax:- for Structure declaration

```
Struct <structure name>
    {
    structure elements
    }<structure variable name>;
```

ex:-
```
struct book
{
    char bk-nm;     // book name
    float bk-price; // book Price
    int bk-pgs;     // book pages
} b1, b2, b3;       // Structure variable
```
Structure elements

Description

↳ The structure name is book. It contains the frame/ structure of a book.

↳ bk.nm, bk-price, bk-pgs are the elements of structure book.

↳ b1, b2, b3 are the Structure variable. Here in structure book 3 elements present.

Syntax 2 for structure declaration

```
struct <structure name>
{
    Structure elements
};

struct <structure name> <structure variables> ;
```

ex:-
```
struct book          // book ← structure name
{
    char bk.nm;     // book name
    float bk-price; // book Price
    int bk-pgs;     // book pages
};                  // Structure elements

struct book b1, b2, b3; // Structure variable
```

Structure INITIALIZATION syntax 1

↳ We need to give value to these structure variables i.e we need to give value to these structure elements. For that we need to access each element of a structure variable.
↳ To access the element of a structure variable we need to use (.) operator i.e dot operator.
ex. b1. name , b1. price, b1. pages ;

ex. of Structure Initialization

```
scanf(" %c %f %d", &b1.name, & b1.price,
        b1.pages ); // it'll access the value of
                    // each element from keyboard
```

Structure Initialization Syntax-2

The structure variables can also be initialized when they are declared.

ex:-
```
struct books   // In structure S is small s
   {           /* i.e structure books */
     char name; float price; int pages;
   } ;

struct books b1 = {'b', 130.50, 550};
```

Note:-
Structure declaration doesnot reserve any memory. At it only define the form of structure. After initialization memory is reserved.

IIT Kharagpur

(5)

Q. WAP to store name (a string), price (a float) and number of pages (an int) of 3 books.

Solution without using Structures

```
#include <stdio.h>
int main()
{
    char name[3];
    float price[3];
    int pages[3];
printf("enter names, prices, pages of 3 books\n");
/* Array initialization */ for(i=0; i<=2; i++)
        scanf("%c %f %d", &name[i],
                &price[i], &pages[i]);
    }
printf("the elements of array that you entered");
/* Array element display */ for(i=0; i<=2; i++)
    {
        printf("%c %f %d", name[i],
            price[i], pages[i]);
    }
return 0;
}
```

O/P:

```
enter names, prices, pages of 3 books
A 100.0 354
C 256.50 682
F 233.70 512

the elements of array that you entered
A 100.0 354
C 256.50 682
F 233.70 512
```

Solution with using structure

```
#include <stdio.h>
int main()
  {
    Struct book
      {
        char name;
        float price;
        int pages;
      };
    Struct book b1, b2, b3;
    Printf(" Enter names, Prices & no. of Pages
            of 3 books \n");
    Scanf("%c %f %d", b1.name, b1.price, b1.pages);
    Scanf("%c %f %d", b2.name, b2.price, b2.pages);
    Scanf("%c %f %d", b3.name, b3.price, b3.pages);

    Printf(" the values that you entered \n");

    Printf("%c %f %d", b1.name, b1.price, b1.pages);
    Printf("%c %f %d", b2.name, b2.price, b2.pages);
    Printf("%c %f %d", b3.name, b3.price, b3.pages);

    return 0;
  }
```

Providing values to the structure elements

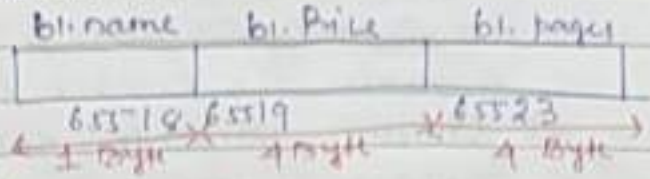
Retrival of structure element values

O/P:

```
Enter names, prices & no. of pages of 3 books
A 100.00 354
C 256.50 682
F 233.70 512
the values that you entered
A 100.00 354
C 256.50 682
F 233.70 512
```

Storage of Structure element

All the structure elements are always stored in adjacent locations.



Description

name is character type. So it'll take 1B memory
Price is float type so it'll take 4B memory
pages is integer type so it'll take 1B memory

Total Memory of structure = memory reserved for b1 + memory reserved for b2 + memory reserved for b3

= (1B+4B+1B) + (1B+4B+1B) + (1B+4B+1B)

= 3x9B

= 27B

Note:

↳ Structure declaration can be done within main() or before main() i.e starting of the source code.

↳ But the initialization will be done within main function.

Structure declaration within main()

```
#include <stdio.h>
Void main()
{
  Struct book
  {
    char nm;
    float Pri;
  }b1;
  Scanf("%c %d", &b1.nm,
        &b1.Pri);
  Printf("%c %d", b1.nm,
         b1.Pri);
```

Structure declaration outside of main() i.e starting of Source code

```
#include <stdio.h>
  Struct book
  {
    char nm;
    float Pri;
  };
Void main()
  {
    Struct book b1;
  printf("enter nm, Pri");
  Scanf("%c %d", &b1.nm,
        b1.Pri);
  printf("%c %d", b1.nm,
         b1.Pri);
```

Array of structures

↳ For one book we have one structure variable, like wise for 3 books we need 3 structure variable so if we increase the number of entity, the number of structure variable will be increased instead of creating so many variables, we can use an array of these variables.

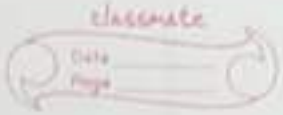
/* WAP to display name, price, pages of 10 books using structure */

```
#include <stdio.h>
void main()
{
    struct book
    {
        char bk_nm; float bk_pric;
        int bk_pgs;
    };
    struct book b[10]; // Structure variable declared
    for(int i=0; i<=9; i++)
    {
        printf("Enter name, price, pages \n");
        scanf("%c %f %d", &b[i].bk_nm,
        &b[i].bk_pri, &b[i].bk_pgs);
        fflush(stdin); //To flush out the keyboard
                       // buffer otherwise, Enter
                       // will be taken as user input
    }
    for(int i=0; i<=9; i++)
    printf("display the value you entered");
    printf("%c %f %d", b[i].bk_nm,
        b[i].bk_pri, b[i].bk_pgs );
    }
}
```

Structure declaration — of defining form of book

Structure variable initialization through user input from keyboard

To retrive the book name, price and pages of 10 books



Description

i=0 b[0].bk_nm = 'A' } Book 1 name
b[0].bk_price = 130.00 } Book 1 Price
b[0].bk_pge = 350 } Book 1 pages

Likewise for remaining 9 books information.
Since we have taken here char bk_nm, so we can only enter single character for book name.

/* WAP to create Student database using Structure */

/* We need to create a database of 100 students which will store name, registration number, Age, mark */

```
#include<string.h>      // for puts( ). To access the string
#include <stdio.h>
Struct  Student
    {
     char firstName[50];
     int roll; int age;
     float marks;
    } S[100];

Void main ()
    {
     int i;
     Printf ("Enter information of Students \n");
     for (i=0; i<100; i++)
         { Printf("enter roll, name, age, marks");
           S[i].roll = i+1;
           Printf (" roll no = %d", S[i].roll);
           Scanf("%s", &S[i].firstName);
           Scanf("%d", &S[i].age);
           Scanf("%f", &S[i].marks);
         }
```

Storing Structure information element

```
for (i=0; i<100; i++)
    {
    printf ("Roll no = %d", S[i].roll);
    printf (" age = %d", S[i].age);
    printf (" marks = %f", S[i].marks);
    puts (S[i].firstName);
    }
}
```

Retrieving structure element information

Sample run

```
enter information of students
enter roll, name, age, marks
1       sachin      20      560
2       suchi       19      750
3
.
.
.
100     ABC         18      650
```

Roll no = 1 age = 20 marks = 560 sachin

.
.
.

Roll no = 100 age = 18 marks = 650 ABC

Structure vs Union

Structure	Union

1. #include<stdio.h>
void main()
{
struct book
{
char nm; //1B
float pric; //4B
int pgs; //4B
} b;
}
	1. #include<stdio.h>
	void main()
	{
	union book
	{
	char nm; //1B
	float Pric; //4B
	int Pgs; //4B
	} b;
	}

2. Size of Structure = sum of size of all Structure element i.e 1B+4B+4B=9B	2. Size of Union = Size of largest element i.e 4B
3. We can retrieve the elements of Structure simultaneously as different memory location has allocated to different Structure element.	4. We can retrieve the elements of union one after another because same memory is allocated to all union elements; i.e shared by individual members of union.
5. We can initialize the elements of Structure simultaneously as different memory location has allocated to different Structure element.	5. We need to initialize the elements of union one by one and we need to retrive immediatly after storing Because all the elements share the same memory location otherwise elements will be overwritten and we'll be able to retrieve the last stored element

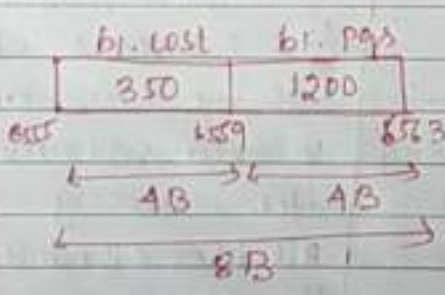
Structure

6. Ex:-
```
Struct book
{
  int cost;
  int pgs;
};

Void main()
{
  Struct book b1;
  Printf(" enter cost,pgs");
  Scanf("%d %d",
        &b1.cost,&b1.pgs);
  Printf("%d %d",
        b1.cost, b1.pgs);

Size of b1 // Printf("%d", sizeof(b1));
}
```

Sample run
```
enter cost, pgs
350   1200
350   1200
8
```

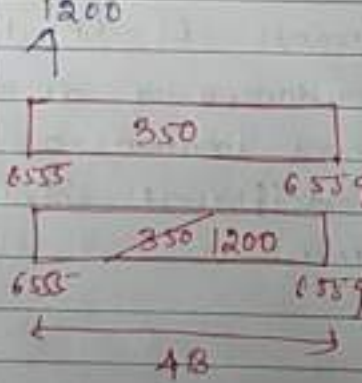


Union

Ex:-
```
union book
{
  int cost;
  int pgs;
}

void main()
{
  union book b1;
  Printf(" enter cost");
  Scanf("%d",&b1.cost);
  Printf("%d", b1.cost);
  Printf(" enter pages");
  Scanf("%d", &b1.pgs);
  Printf("%d", b1.pgs);
  Printf("%d", sizeof(b1));
}
```

Sample run
```
enter cost
350
350
enter pages
1200
1200
```



LINKED LIST

In a game of Treasure Hunt, you start by looking for the first clue. When you find it, instead of having the treasure, it has the Location of the next clue and so on. You keep ~~the~~ following the clues until you get to the treasure.
A Linked list is similar to this game.

Linked List

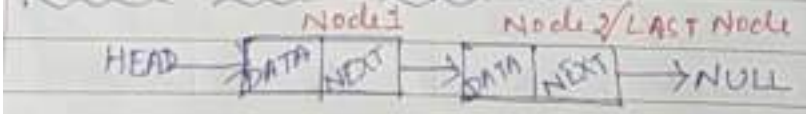
↳ Linked list is a set of dynamically allocated nodes, arranged in such a way that each node contains one value and one pointer. The pointer always points to the next member of the list. If the pointer is NULL, then it is the last node in the list.

↳ Linked list is a linear data structure, in which the elements are not stored at contiguous memory location.

↳ A node can store data and address of the next node.

↳ The data in node may be a number, a string or any other type of data.

Linked List Representation



↳ HEAD :- Stores the address of Node 1
↳ DATA :- DATA Stored in Node 1
↳ NEXT :- Pointer to the next node
↳ NULL :- The next portion of the Last node is NULL as it is not pointing to any other node.

Structure of Linked list node

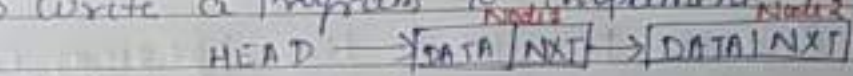
```
Struct node
{ int data;
  Struct node * next;
};
```

Note: Here we have used Structure because a Node consists of data and address which are of different type.

Data: It can be integer, string or any other type here we have taken integer.

Address:- Since one node will point to another node which is also Structure type so here we have taken pointer to structure

Let's write a Program to implement




```
#include<stdio.h>
struct node
{
    int data;
    struct node *next;
};
struct node
void main()
{
    struct node *head;
    struct node *one = NULL;
    struct node *two = NULL;
    struct node *three = NULL;
    one = malloc(sizeof(struct node));
    two = malloc(sizeof(struct node));
    three = malloc(sizeof(struct node));
/*  [1|NF]  */  one->data = 1;
/*  [2|NF]  */  two->data = 2;
                three->data = 3;
/*  [1|NF]->[2|NF]  */  one->next = two;
/*Last node so next=NULL*/  two->next = three; three->next = NULL;
    head = one; // Head stores the address of Node1
}
```

Structure of a node

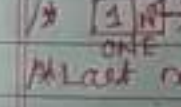
Pointers of node type till now not pointing to any memory. So initialized with NULL

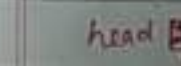
Dynamic Memory allocation, which addresses are stored in one and two respectively.

(creation of Node1 and Node2)

Stores the data in Node1 and Node 2

Stores the address



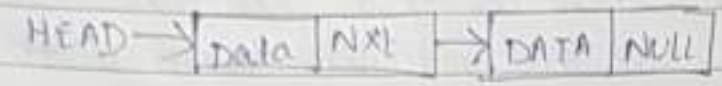


Types of Linked List

There are 3 common types of linked list

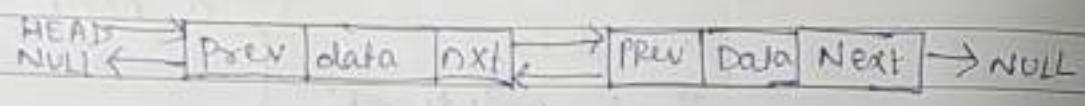
i) Singly linked list
ii) Doubly linked list
iii) Circular linked list

Singly linked list → It is the most common. Each node has data and a pointer to the next node.



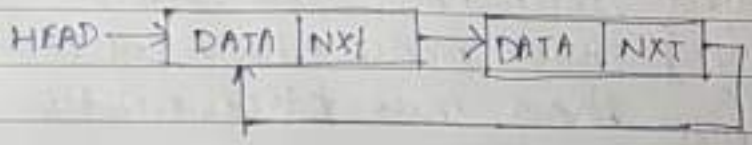
Doubly linked list

We add a pointer to the previous node in a doubly linked list. Thus, we can go in either direction: forward or backward.



Circular linked list

A circular linked list is a variation of linked list in which the last element is linked to the first element. This forms a circular loop.



Note

→ it is a structure operator. It is used to access the data portion and next portion of a Node.

Difference between Array and Linked List

Array	Linked List
1. Arrays are of fixed size.	1. Linked lists are dynamic in size.
2. It is not flexible. So can't extend and contract in its size.	2. It is flexible and can extend and contract in size.
3. Memory is assigned during compile time.	3. Memory is allocated during execution or runtime.
4. Elements are stored consecutively in arrays.	4. Elements are stored randomly in Linked List.
5. The requirement of memory is less as actual data being stored within the index in the array.	5. More memory is needed in Linked List due to storage of additional next and previous referencing elements.
6. Insertion of new element and deletion of existing element is expensive. (as new place is needed to create and an existing place is needed to delete)	6. Ease of insertion/deletion. i.e faster.
7. Random access of elements is allowed. It is possible through index.	7. Random access of elements is not allowed. We have to access elements sequentially starting from the first node. So we can't do a binary search with Linked List.
8. Accessing an array element is easy and faster.	8. Linked list takes linear time so bit slower than array.

USE of POINTERS in SELF-referential structures

↳ Self Referential structure is the data structure in which the pointer refers (points) to the structure of the same type.

↳ It is used in many of the data structures like in Linked List, Trees, Graphs, Heap etc.

↳ It can also be defined as the structures that have one or more pointers which point to the same type of structures as their member.

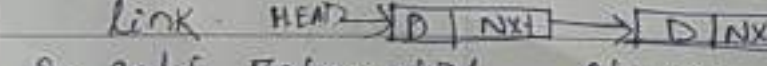
Ex:-
```
Struct node
{
    int data1;
    char data2;
    Struct node* link;
};
```

In linked list one node (structure type) points to another node (structure type).

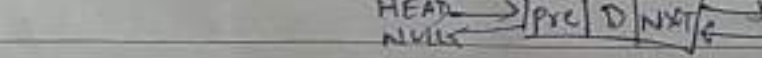
↳ In the above example 'link' is a pointer to a structure of type 'node'. Hence the structure 'node' is a self referential structure with 'link' as the referencing pointer.

Types of Self Referential Structures

1. Self referential structure with single link



2. Self Referential structure with multiple links.



Searching Algorithm

↳ We have already learnt about array. Now we'll learn some of the implementation of array.

↳ Let we have a set of data and we want to check that whether the desired data in the array are not. For this reason we use Searching algorithm.

↳ Mainly there are two types of Searching algorithm.

- ↳ Linear Search.
- ↳ Binary Search.

Linear Search:-

Step 1:
↳ Here a set of data will be present in an array.

Step 2: User should have a desired data which he/she wants to check whether it is present inside array or not.

Step 3: The whole array will be traversed and the desired data will be compared with each array element.

Step 4: When the desired element will be matched with any element of array the position of the array will be returned.

Linear search is also known as sequential search.

Program - 1:

```
#include <stdio.h>
void main()
{
  int Lin_Srch[100], Srch_elmnt, cntr, n;
  printf("Enter no. of elements in array");
   scanf("%d", &n);
  printf("Enter %d integers", n);
  for (cntr = 0; cntr < n; cntr++)
   scanf("%d", &array[cntr]);
  printf("Enter a number to search");
  scanf("%d", &Srch_elmnt);
  for (cntr = 0; cntr < n; cntr++)
   {
    if (array[cntr] == Srch_elmnt)
     {
      printf("%d is present at location
                    %d", Srch_elmnt, cntr+1);
      break;
     }
   }
  if (cntr == n)
  printf("%d is not present in the array", Srch_elmnt);
  return 0;
}
```

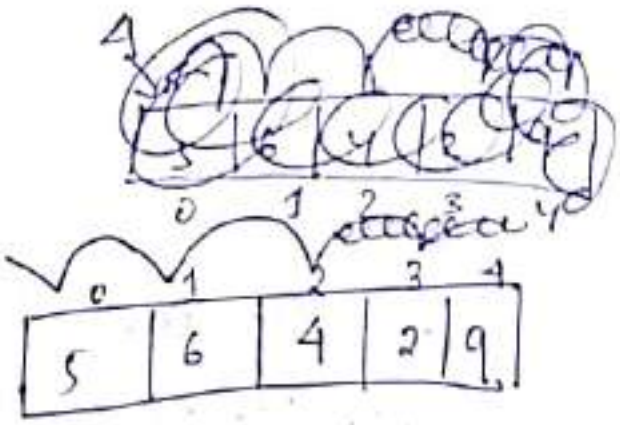
output:

Enter the number of elements in array

5

enter 5 numbers

5
6
4
2
9



Enter the number to search

4

4 is present at location 3.

Binary Search:

↳ This is also used to search a desired element in an array.

↳ It can only be used for sorted arrays.

↳ It is fast as compared to Linear Search.

↳ If the array is not sorted then you need to sort using ~~merge~~ any sorting algorithm then apply Binary Search.

↳ If the element to be search is found then its position is printed.

algo

Step 1: take an array of elements.

Step 2: if it is in sorted order then goto step 3 otherwise sort the array using any sorting algorithm.

Step 3: Find the mid position. The elements before the mid position will be smaller than the element present at mid position. The elements present after the mid position will be larger than the element present at mid position →

$$\text{mid position} = \frac{\text{Zeroth index} + \text{last index}}{2}$$

Step 4: If the desired element is equal to the mid element then return the mid position.

Step 5: If the desired element is less than mid element then

firstindex= lastindex=(mid−1) index.

Step 6: If the desired element is greater than mid element then

firstindex = (mid+1) index.

Step 7:- Continue step 3 to 6 until get the desired data.

Step 8:- Else element is not present.

Note 9

Sorting algorithm

↳ This algorithm is used for sort an array in ascending order or descending order.

↳ Sorting algorithm is used to sort the array for binary search.

↳ There are various sorting algorithm.

- ↳ Bubble sort
- ↳ Insertion sort
- ↳ Selection sort
- ↳ Quick sort
- ↳ Merge sort
- ↳ Heap sort.

Bubble sort

↳ It is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are not in proper order.

↳ It is sometimes referred to as sinking sort.

↳ It'll traverse the array until the array is sorted. So there is a possibility that it'll traverse ~~two or three times~~ max number of times i.e. total number of elements - 1.

How bubble sort works:-

Step 1: We'll take an unsorted array.

Step 2: Bubble sort starts with very first two elements comparing them to check which one is greater. then swapping is done.

Step 3: At each iteration at least one value moves at the end.

Step 4: When there is no swap required, bubble sorts learns that array is completely sorted.

Ex:-

6	1	2	3	4	5

first traversal of array i.e 1st pass:

Step 1:- compare 6 with 1. 6>1 . so swap:

6	1	2	3	4	5

1	6	2	3	4	5

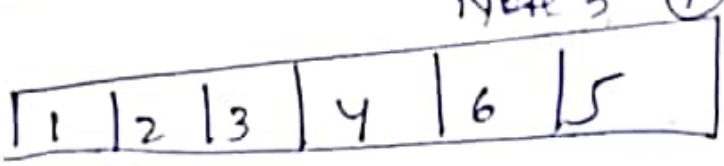
Step 2: compare 6 with 2 6>2 so swap.

1	2	6	3	4	5

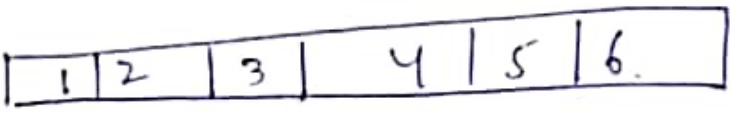
Step 3:- compare 6 with 3 6>3 so swap

1	2	3	6	4	5

Step 4: compare 6 with 4 6>4 so swap

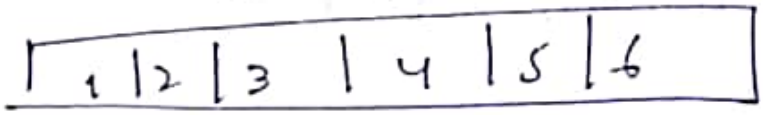


step 5: compare 6 with 5. 6>5. So swap



1st pass completed.

2nd traversal of array i.e Pass-2

Step 1. 

compare 1 with 2. 1<2 OK. No swap.

Step 2. compare 2 with 3 2<3 OK. No swap

Step 3. compare 3 with 4 3<4 OK. No swap

Step 4: compare 4 with 5 4<5 OK No swap

Step 5. compare 5 with 6 5<6 OK No swap.

So 2nd pass completed and array is sorted.

Program:

Go through your Note book.

Logic

Note 3 (8)

Ex:- According to program (Do like this in exam if array is given)

5	3	1	9	8	2	4	7

i=0 j Pass 1

0 — 5 | 3 | 1 | 9 | 8 | 2 | 4 | 7 — 5>3 swap

1 — 3 | 5 | 1 | 9 | 8 | 2 | 4 | 7 — 5>1 swap

2 — 3 | 1 | 5 | 9 | 8 | 2 | 4 | 7 — 5<9 NO swap

3 — 3 | 1 | 5 | 9 | 8 | 2 | 4 | 7 — 9>8 swap

4 — 3 | 1 | 5 | 8 | 9 | 2 | 4 | 7 — 9>2 swap.

5 — 3 | 1 | 5 | 8 | 2 | 9 | 4 | 7 — 9>4 swap

6 — 3 | 1 | 5 | 8 | 2 | 4 | 9 | 7 — 9>7 swap

3 | 1 | 5 | 8 | 2 | 4 | 7 | 9 — i/p to pass 2

i=1 j Pass 2

0 — 3 | 1 | 5 | 8 | 2 | 4 | 7 | 9 — 3>1 swap

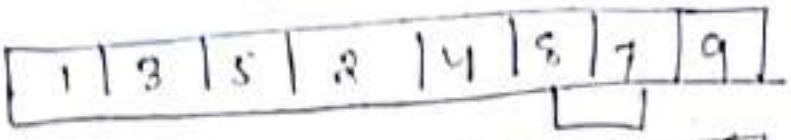
1 — 1 | 3 | 5 | 8 | 2 | 4 | 7 | 9 — 3<5 NO swap

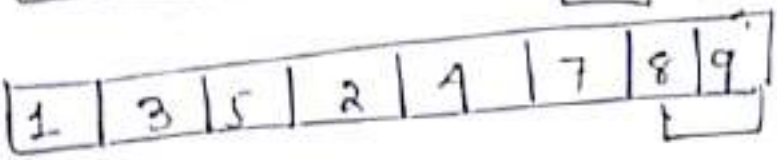
2 — 1 | 3 | 5 | 8 | 2 | 4 | 7 | 9 — 5<8 NO swap

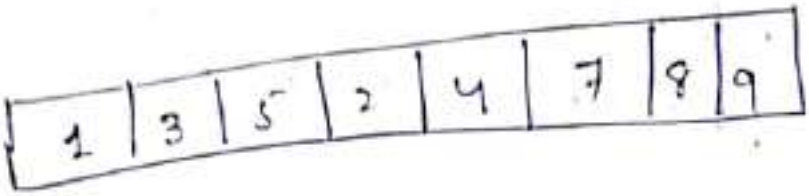
3 — 1 | 3 | 5 | 8 | 2 | 4 | 7 | 9 — 8>2 swap

4 — 1 | 3 | 5 | 2 | 8 | 4 | 7 | 9 — 8>4 swap

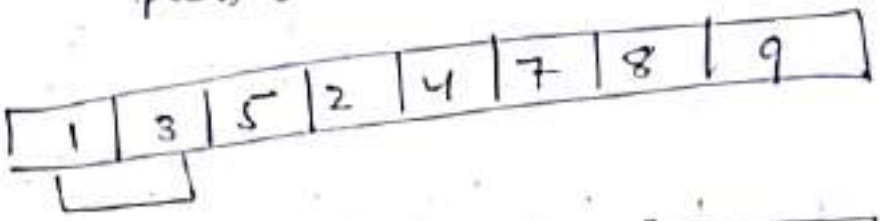
Note 3 ⑨

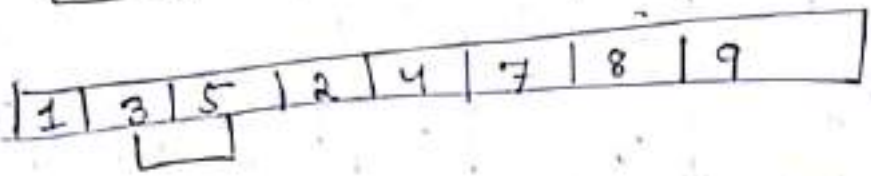
5  8 > 7 swap

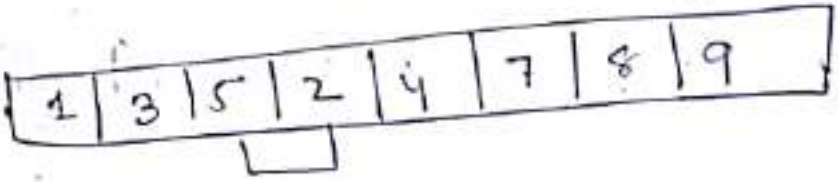
6  8 < 9 No swap

 i+1 to 2nd pass

i=2 j Pass 3

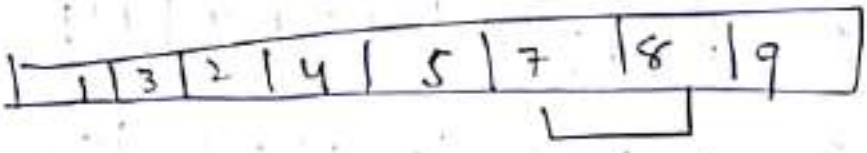
0  1 < 3 no swap

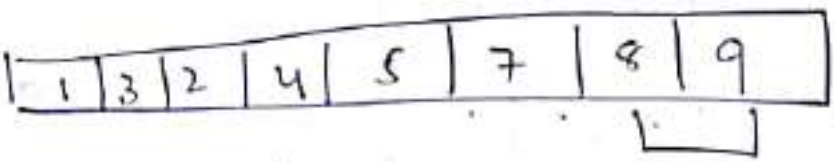
1  3 < 5 no swap

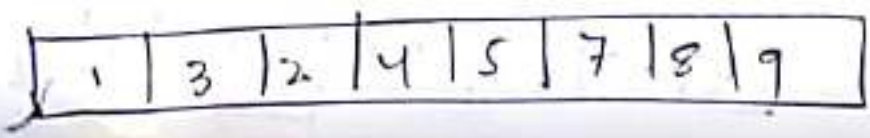
2  5 > 2 swap

3  5 > 4 swap

4  5 < 7 no swap

5  7 < 8 No swap

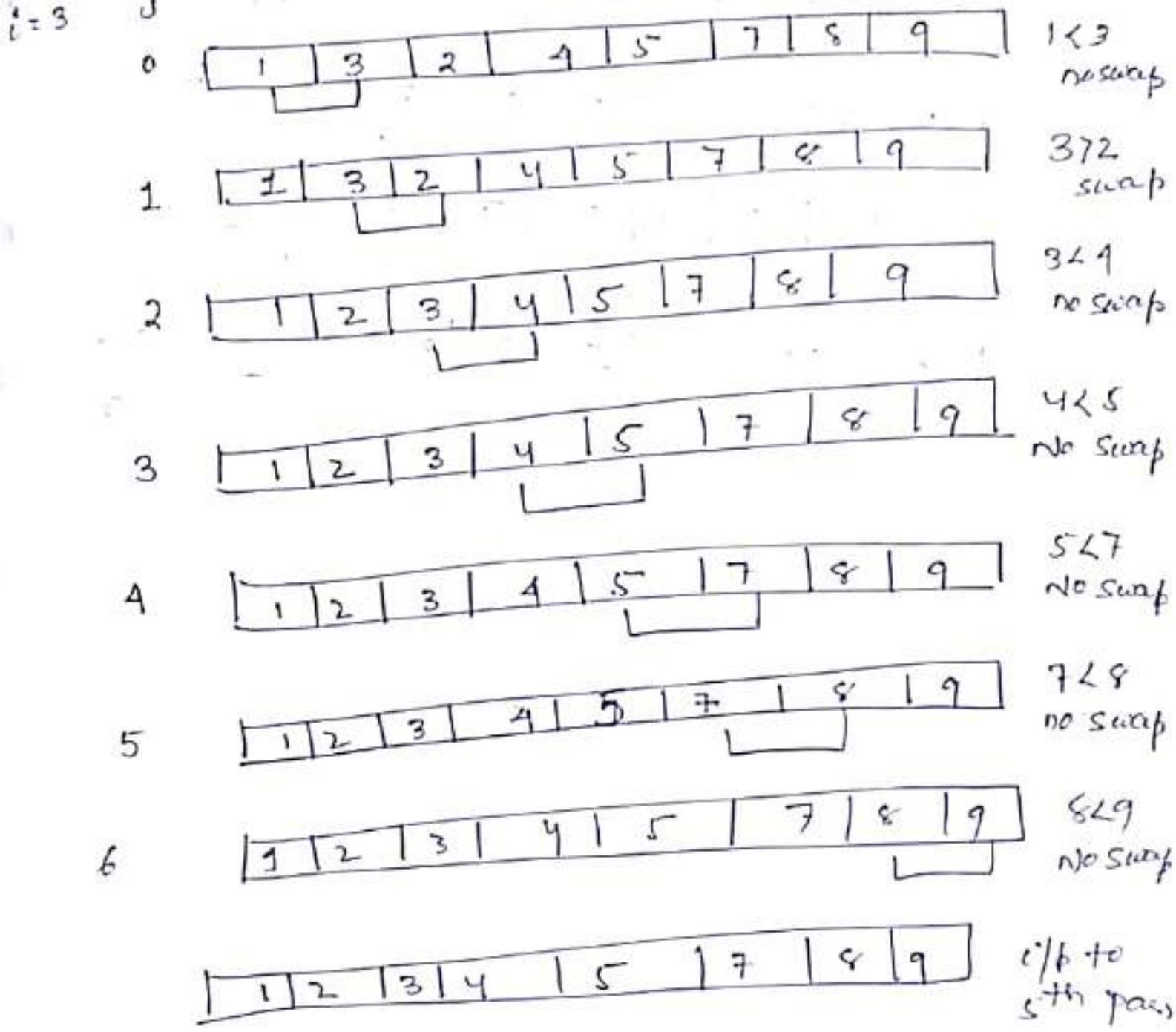
6  8 < 9 No swap

 i+1 to 1th pass

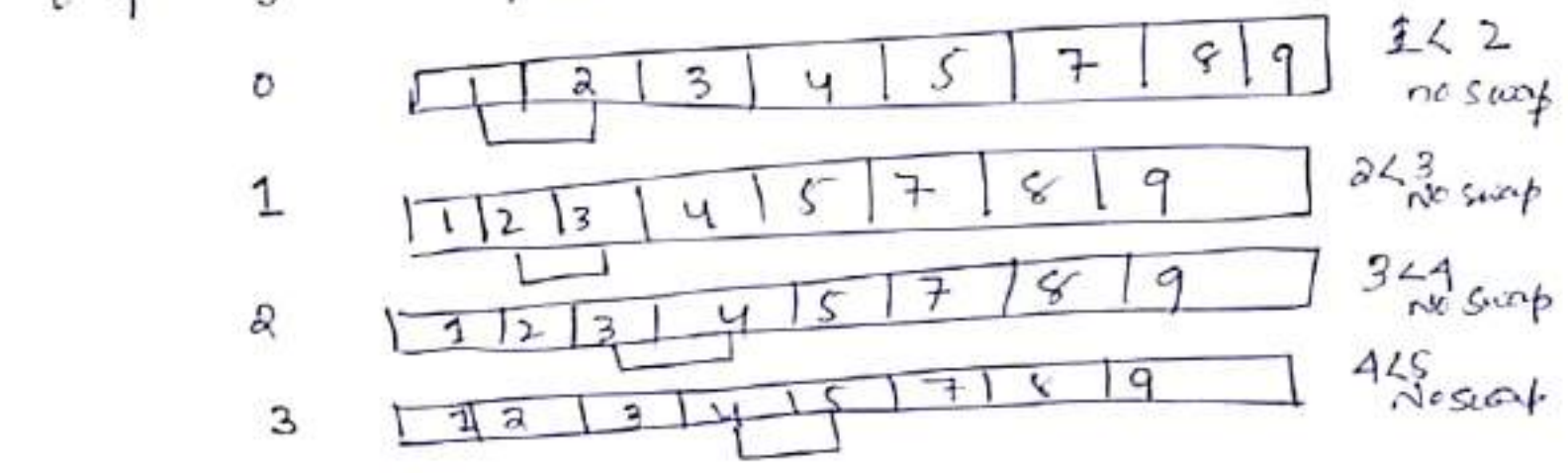
Note 3 (10)

pass 1

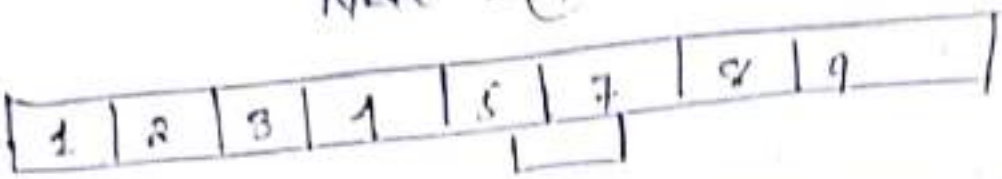
i=3 j



i=4 j pass 5



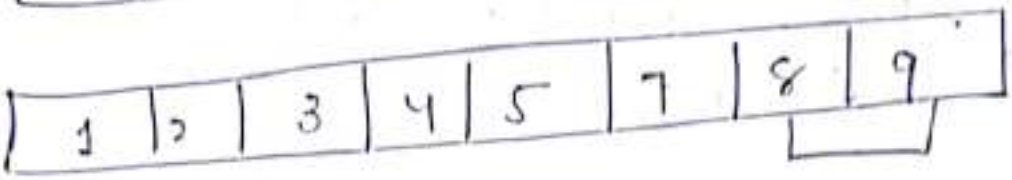
Next 3 (11)

4 

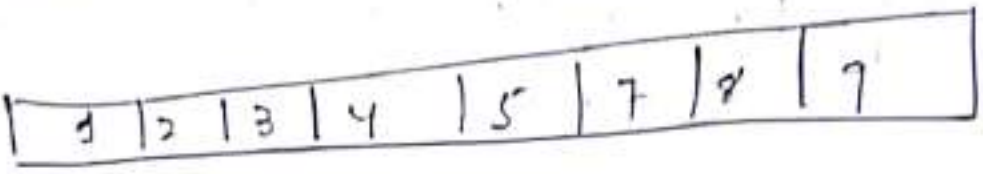
5<7 no swap

5 

7<8 no swap

6 

8<9 No swap.



Sorted array

Insertion Sort:

↳ This is used to sort an array of elements either in ascending order or by descending order by inserting the elements at its proper position.

↳ Algorithm:-

Step 1: The second element of an array is compared with the elements that appears before it (only first element in this case). If the second element is smaller than first element, second element is inserted in the position of first element. After first step, first two elements of an array will be sorted.

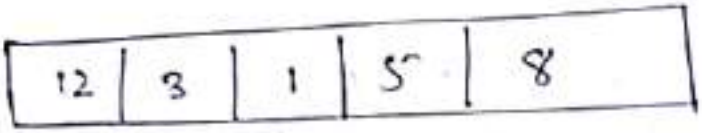
Step 2: The third element of an array is compared with the elements that appears before it (first and second element). If third element is smaller than first element, it is inserted in the position of first element. If third element is larger than first element but, smaller than second element, it is inserted in the position of second element. If third element is larger than both the elements, it is kept in the position as it is. After second step, first three elements of an array will be sorted.

Step 3: Similarly the fourth element of an array is compared with the elements that appear before it (first, second and third elements) and the same procedure is applied and that element is inserted in the proper position. After third step, first four elements of an array will be sorted.

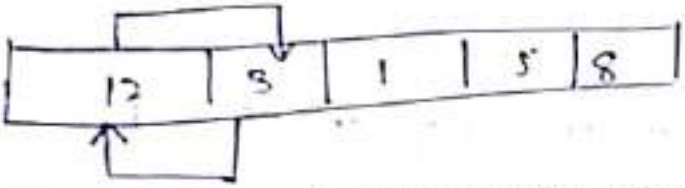
Note:

If there are 'n' elements to be sorted, then this procedure is repeated n-1 times to get sorted list of array.

Ex:

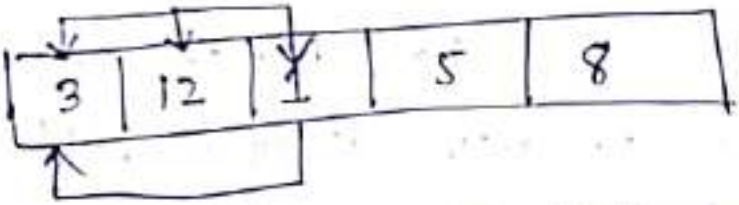


Step 1:



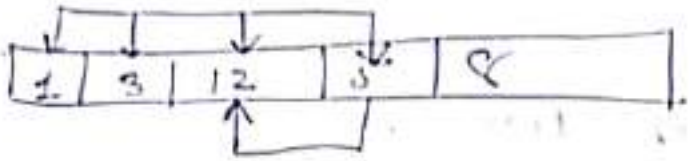
Checking second element of array with element before it and inserting it in proper position. In this case 3 is inserted in position of 12.

Step 2:



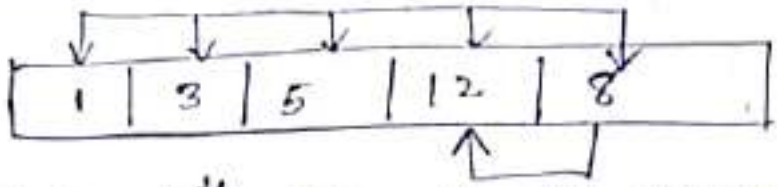
Checking third element of array with elements before it and inserting it in proper position. In this case 1 is inserted in position of 3.

Step 3:

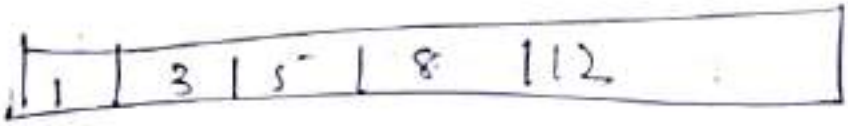


Checking fourth element of array with elements before it and inserting it in proper position. In this case, 5 is inserted in position of 12.

Step 4:



Checking 5th element of array with elements before it and inserting it in proper position. In this case, 8 is inserted in position of 12.



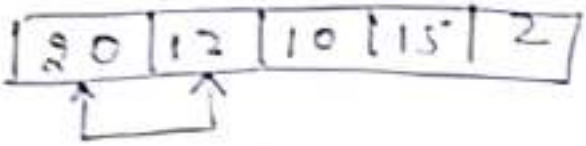
Sorted Array in ascending order.

Logic:

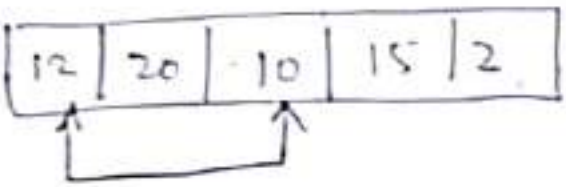
Pass 1

Ex:-

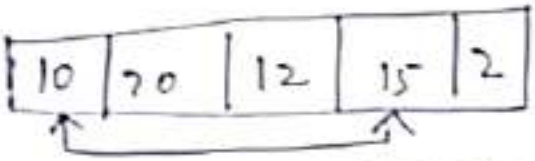
step 1:



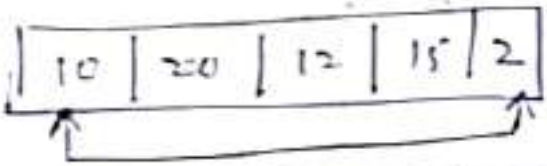
step 2:



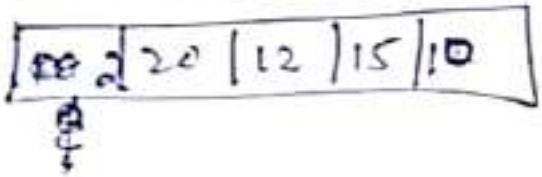
Step 3:



Step 4:

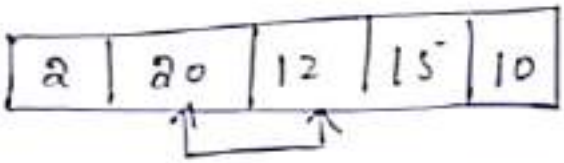


Step 5:-

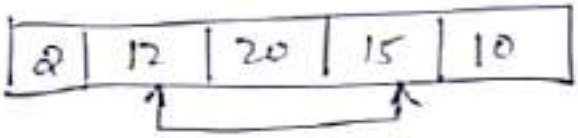


Pass - 2

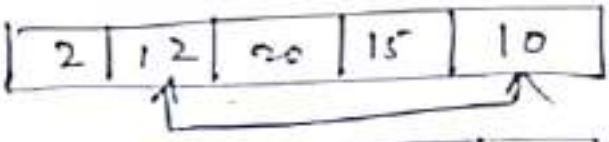
Step 1



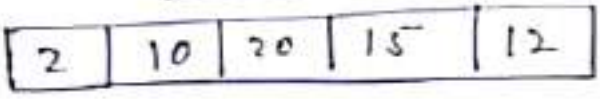
Step 2



Step 3

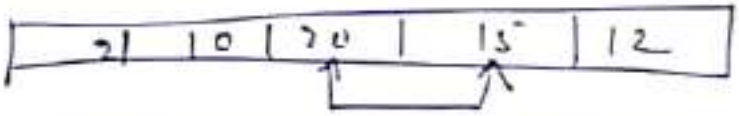


Step 4:

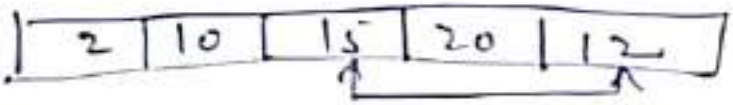


Pass - 3

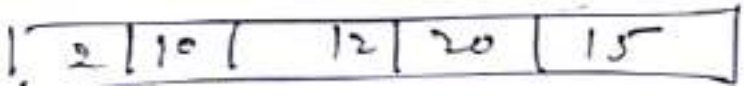
Step 1



Step 2

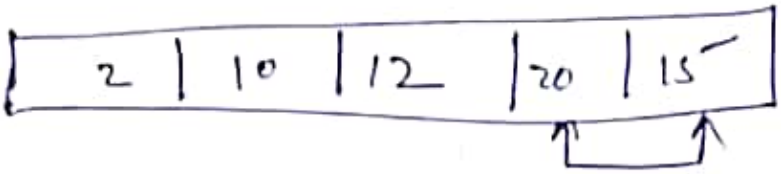


Step 3.

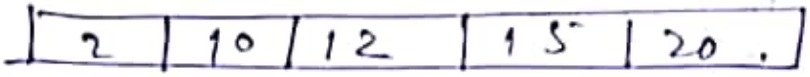


Pass: 4

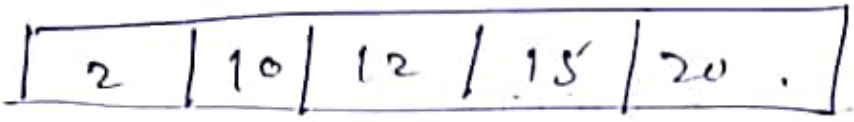
Step 1



Step 2



The Sorted array



Program:-

Go through Lab Record.

Program

Go through the Lab Record.

Selection Sort:

Step 1: It starts by comparing first two elements of an array and swapping if necessary i.e if you want to sort the elements of array in ascending order and if the first element is greater than second then, you need to swap the elements. but, if the first element is smaller than second, leave the element as it is.

Step 2:- Again first element and third element are compared and swapped if necessary.

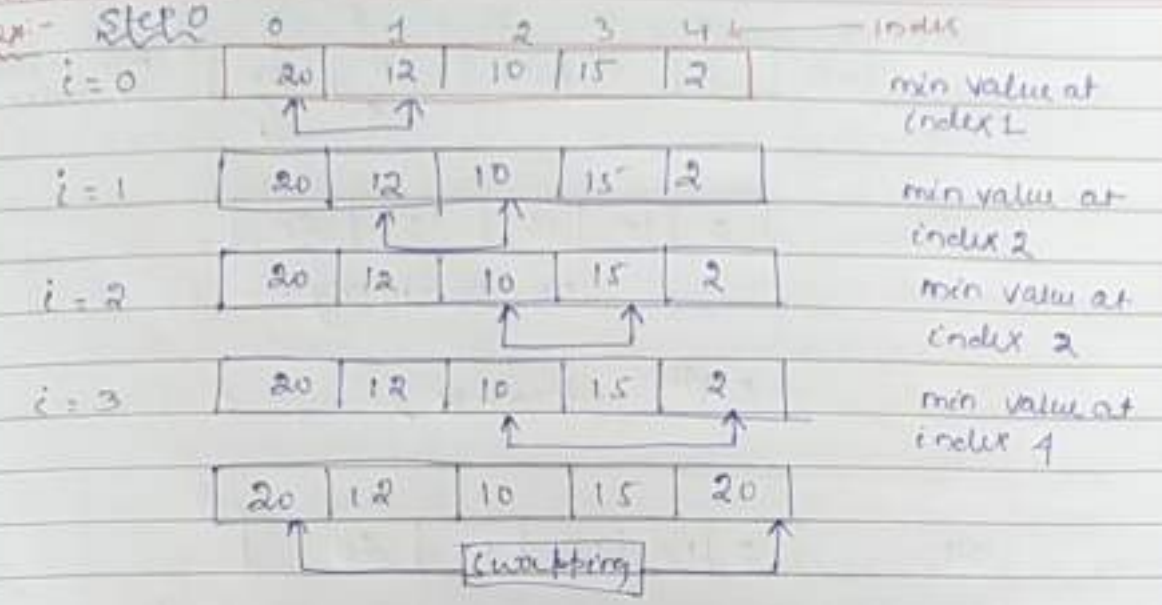
Step 3:- This process goes on until first and last element of an array is compared

Step 4:- Like wise 2nd element will be compared with 3rd to last element.

Step 5:- This process will be continued untill the array is sorted.

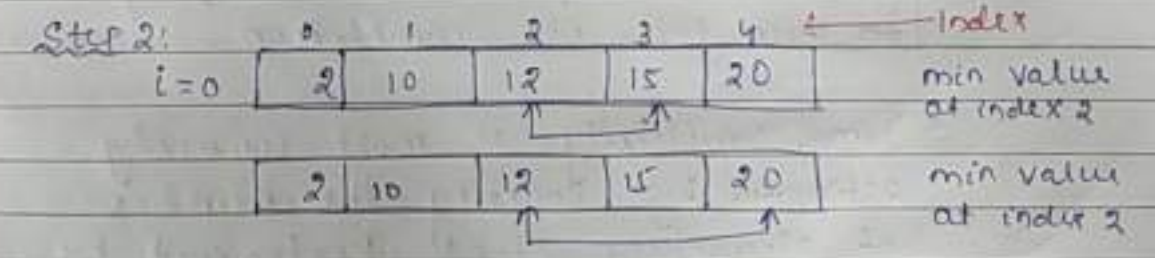
Note:- If there are 'n' elements to be sorted then the process will be repeated n-1 times to get required result.

Ex of Selection sort



(This is the first iteration)

(This is second iteration)



(no swapping)

(This is 3rd iteration)

Step 3:

	0	1	2	3	4
i=0	2	10	12	15	20

min value at index 3

2	10	12	15	20

no swapping

(4th iteration)

O/P:

2	10	12	15	20

Complexity of Algorithm

↳ For any defined Problem, there can be N number of solution. But which is the best solution i.e based on the circumstances

↳ So for any algorithm there are two circumstances i.e Time & space.

↳ **Time complexity**

* Time complexity of an algorithm is the total time required by the program to run till its completion.

* Time complexity is most commonly estimated by counting the number of elementary steps performed by an algorithm to finish execution.

* The time complexity of algorithm is most commonly expressed using the big O notation.

* big oh (O) is an asymptotic notation.

* Algorithm's performance may vary with different types of input data. Hence there are 3 types of time complexity
  i) Best case time complexity
  ii) Average case time complexity
  iii) Worst case time complexity

* For an algorithm we generally use the worst-case time complexity because that is the maximum time taken for any input size.

Best case time complexity
The minimum time taken to complete execution of an algorithm for any input size.

Average case time complexity
The average time taken to complete execution of an algorithm for any input size.

Worst case time complexity

The maximum time taken to complete execution of an algorithm for any input size.

Space complexity

↳ Whenever a solution to a problem is written some memory is required.

↳ For any algorithm memory can be used for following

1. Variables (constant values, temporary values)
2. program instruction
3. Execution

↳ Space complexity is the amount of memory used by the algorithm (including the input values to the algorithm) to execute and produce the result.

↳ Space complexity = Auxiliary space + Input space

↳ Auxiliary space is the extra space or the temporary space used by the algorithm during it's execution.

↳ While executing algorithm memory space is used for 3 reasons
   i) Instruction space
   ii) Environmental stack
   iii) Data space.

↳ While calculating the space complexity of an algorithm, we usually consider only data space and we neglect the instruction space and environmental stack.

↳ Instruction space
   It is the amount of memory used to save the compiled version of instruction.

↳ Environmental stack
   Sometimes an algorithm (function) may be called inside another algorithm (function). In such a situation, the current variables are pushed on to the system stack,

where they wait for further execution and then call to the inside algorithm (function) is made.

ex: If a function A() calls function B() inside it, then all the variables of function A() will get stored on the system stack temporarily, while the function B() is called and executed inside the function A()

↳ Dat space
Amount of space used by the variables and constants.

Calculating the space complexity:-

↳ For calculating the space complexity, we need to know the value of memory used by different types of data type variables.

↳ size of variable varies for different operating systems, but the method for calculating the space complexity remains the same.

Data type	size
1. bool, char, unsigned char, signed char, __int8	1 Byte
2. __int16, short, unsigned short, wchar_t, __wchar_t	2 Bytes
3. float, __int32, int, unsigned int, long, unsigned long	4 Bytes
4. double, __int64, longdouble, long long	8 Bytes.

ex. to compute space complexity

```
{ int a,b,c;   // a=4B, b=4B, c=4B
  int z= a+b+c; // z=4B
  return(z);  // 4B to return
}            ∴ Total space = 4B+4B+4B+4B = 24B
```

Here the space requirement is fixed so it is called

Constant space complexity

ex:-
```
int sum (int a[], int n)        // n is the length of array
{
  int x = 0;        // 4 B for x
  for (int i = 0; i<n; i++) // 4 B for i
  {
    x = x + a[i];
  }
  return x;
}
```

↳ no. of element in array a[] is n so space required = 4×n

↳ 4 B for each x, n, i and return value.
4+4+4+4 = 16

↳ Total Memory = 4n+16

It is increasing linearly with the increase in the input value n. Hence it is called as Linear space complexity

↳ We should always focus on writing algorithm in such a way that we keep the space complexity minimum.

Time complexity calculation

Bubble sort

No. of iteration	No. of comparisons
1st	(n-1)
2nd	(n-2)
3rd	(n-3)
⋮	⋮
Last	1

Total no. of comparisons:

(n-1) + (n-2) + (n-3) + ··· + 1

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

$= n(n+1)/2 \approx n^2$

complexity $= O(n^2)$

Note

The bubble sort uses two loops so the complexity is $n*n = n^2$

Worst case complexity $O(n^2)$

If we want to sort in ascending order and the array is in descending order then the worst case occurs.

Best case complexity $O(n)$

↳ If the array is already sorted, then there is no need for sorting.

Average case complexity $O(n^2)$

It occurs when the elements of the array are in jumbled order (neither ascending nor descending)

Applications of Bubble Sort

1. Bubble sort will be used if the complexity of the code does not matter.
2. A sort code is preferred.

Space complexity of Bubble Sort

$O(1)$ as constant space.

Insertion Sort time complexity

Each element has to be compared with each of the other elements. So for every n^{th} element $(n-1)$ no. of comparisons are made. Thus total no. of comparisons $= n*(n-1) \approx n^2$

∴ Time complexity $= O(n^2)$

Worst case time complexity = $O(n^2)$
Best case time complexity = $O(n)$
Average case time complexity = $O(n^2)$

Insertion Sort Applications
i) The array has small number of elements
ii) There are only few elements left to be sorted

Space complexity of Insertion Sort
$O(1)$ as constant space.

Selection Sort Time complexity

Iteration	no. of comparisons
1st	$n-1$
2nd	$n-2$
3rd	$n-3$
.	
.	
.	
Last	1

Total no. of comparisons:

$(n-1)+(n-2)+\cdots+1$
$= 1+2+\cdots+(n-2)+(n-1)$
$= n(n-1)/2 \approx n^2$
$\therefore$ Time complexity $= O(n^2)$

Worst case time complexity = $O(n^2)$
Best " " " = $O(n^2)$
Average " " " = $O(n^2)$

Here the time complexity of Selection Sort is same in all cases because At every step, we need to find minimum element and Put it in right place. The minimum element is not known until the end of the

array is not reached.

Applications of selection sort

It will be used when

i) A small list is to be sorted

ii) cost of swapping does not matter

iii) checking of all the elements is compulsory.

iv) cost of writing to a memory matters like in a flash memory (number of writes/swaps is $O(n)$ as compared to $O(n^2)$ of bubble sort

Space complexity

$O(1)$ i.e constant space

SEARCHING ALGORITHM:

Linear Search:

Linear search in C programming: The following code implements linear search (Searching algorithm) which is used to find whether a given number is present in an array and if it is present then at what location it occurs. It is also known as sequential search. It is straightforward and works as follows: We keep on comparing each element with the element to search until it is found or the list ends.

Linear search C program

```
1.  #include <stdio.h>
2.
3.  int main()
4.  {
5.    int array[100], search, c, n;
6.
7.    printf("Enter number of elements in array\n");
8.    scanf("%d", &n);
9.
10.  printf("Enter %d integer(s)\n", n);
11.
12.  for (c = 0; c < n; c++)
13.    scanf("%d", &array[c]);
14.
15.  printf("Enter a number to search\n");
16.  scanf("%d", &search);
17.
18.  for (c = 0; c < n; c++)
19.  {
20.    if (array[c] == search)    /* If required element is found */
21.    {
22.      printf("%d is present at location %d.\n", search, c+1);
23.      break;
24.    }
25.  }
26.  if (c == n)
27.    printf("%d isn't present in the array.\n", search);
28.
29.  return 0;
30.}
```

Out put:

Enter the number of elements in array

5

Enter 5 elements

5

6

4

2

9

Enter the number to search

4

4 is present at location 3

Binary Search Program:

C program for binary search: This code implements binary search in C language. It can only be used for sorted arrays, but it's fast as compared to linear search. If you wish to use binary search on an array which isn't sorted, then you must sort it using some sorting technique say merge sort and then use the binary search algorithm to find the desired element in the list. If the element to be searched is found then its position is printed. The code below assumes that the input numbers are in ascending order.

```
1.  #include <stdio.h>
2.
3.  int main()
4.  {
5.     int c, first, last, middle, n, search, array[100];
6.
7.     printf("Enter number of elements\n");
8.     scanf("%d",&n);
9.
10.   printf("Enter %d integers\n", n);
11.
12.   for (c = 0; c < n; c++)
13.      scanf("%d",&array[c]);
14.
15.   printf("Enter value to find\n");
16.   scanf("%d", &search);
17.
18.   first = 0;
```

```
   last = n - 1;
   middle = (first+last)/2;

   while (first <= last) {
      if (array[middle] < search)
         first = middle + 1;
      else if (array[middle] == search) {
         printf("%d found at location %d.\n", search, middle+1);
         break;
      }
      else
         last = middle - 1;

      middle = (first + last)/2;
   }
   if (first > last)
      printf("Not found! %d isn't present in the list.\n", search);

   return 0;
}
```

Enter number of Elements

7

Enter 7 integers

-4

5

8

9

11

43

485

Enter value to find

11

11 found at location 5

SORTING ALGORITH:

BUBBLE SORT

Bubble sort algorithm starts by comparing the first two elements of an array and swapping if necessary, i.e., if you want to sort the elements of array in ascending order and if the first element is greater than second then, you need to swap the elements but, if the first element is smaller than second, you mustn't swap the element. Then, again second and third elements are compared and swapped if it is necessary and this process go on until last and second last element is compared and swapped. This completes the first step of bubble sort.If there are n elements to be sorted then, the process mentioned above should be repeated n-1 times to get required result. But, for better performance, in second step, last and second last elements are not compared becuase, the proper element is automatically placed at last after first step. Similarly, in third step, last and second last and second last and third last elements are not compared and so on.

/*C Program To Sort data in ascending order using bubble sort.*/

```
#include <stdio.h>
int main()
{
int data[100],i,n,step,temp;

printf("Enter the number of elements to be sorted: ");

scanf("%d",&n);

for (i=0;i<n;++i)

 { printf("%d. Enter element: ",i+1);

scanf("%d",&data[i]);
}
for(step=0;step<n-1;++step)
for(i=0;i<n-step-1;++i)
{
 if(data[i]>data[i+1])   /* To sort in descending order, change > to < in this line. */
 {
 temp=data[i];
data[i]=data[i+1];
```

```
data[i+1]=temp;
 }
 }
 printf("In ascending order: ");
 for(i=0;i<n;++i)
printf("%d  ",data[i]);
return 0;
}
```

```
Enter the number of elements to be sorted: 6
1. Enter element: 12
2. Enter element: 3
3. Enter element: 0
4. Enter element: -3
5. Enter element: 1
6. Enter element: -9
In ascending order: -9 -3 0 1 3 13
```

INSERTION SORT:

Suppose, you want to sort elements in ascending order. Then,

Step 1: The second element of an array is compared with the elements that appear before it (only first element in this case). If the second element is smaller than first element, second element is inserted in the position of first element. After first step, first two elements of an array will be sorted.

Step 2: The third element of an array is compared with the elements that appears before it (first and second element). If third element is smaller than first element, it is inserted in the position of first element. If third element is larger than first element but, smaller than second element, it is inserted in the position of second element. If third element is larger than both the elements, it is kept in the position as it is. After second step, first three elements of an array will be sorted.

Step 3: Similarly, the fourth element of an array is compared with the elements that appear before it (first, second and third element) and the same procedure is applied and that element is inserted in the proper position. After third step, first four elements of an array will be sorted. If there are n elements to be sorted. Then, this procedure is repeated n-1 times to get sorted list of array.

```
/*sorting Elements of an array in ascending order using insertion sort algorithm*/

#include<stdio.h>

int main()

{

int data[100],n,temp,i,j;
```

```
printf("Enter number of terms(should be less than 100): ");

scanf("%d",&n);

printf("Enter elements: ");

for(i=0;i<n;i++)

{

scanf("%d",&data[i]);

}

for(i=1;i<n;i++)

{

temp = data[i];

j=i-1;

while(temp<data[j] && j>=0)

/*To sort elements in descending order, change temp<data[j] to temp>data[j] in
above line.*/

{

data[j+1] = data[j];

--j;

}

data[j+1]=temp;

}

printf("In ascending order: ");

for(i=0; i<n; i++)

printf("%d\t",data[i]);

 return 0;

}
```

OUT PUT:

Enter number of terms (should be less than 100): 5

Enter elements: 12

1

2

5

3

In ascending order: 1 2 3 5 12

SELECTION SORT:

Selection sort algorithm starts by comparing first two elements of an array and swapping if necessary, i.e., if you want to sort the elements of array in ascending order and if the first element is greater than second then, you need to swap the elements but, if the first element is smaller than second, leave the elements as it is. Then, again first element and third element are compared and swapped if necessary. This process goes on until first and last element of an array is compared. This completes the first step of selection sort.

If there are n elements to be sorted then, the process mentioned above should be repeated n-1 times to get required result. But, for better performance, in second step, comparison starts from second element because after first step, the required number is automatically placed at the first (i.e, In case of sorting in ascending order, smallest element will be at first and in case of sorting in descending order, largest element will be at first.). Similarly, in third step, comparison starts from third element and so on.

```
#include <stdio.h>

int main()

 {

int data[100],i,n,steps,temp;

printf("Enter the number of elements to be sorted: ");

scanf("%d",&n);

   for(i=0;i<n;++i)

{
```

```
printf("%d. Enter element: ",i+1);
scanf("%d",&data[i]);
 }
 for(steps=0;steps<n;++steps)
for(i=steps+1;i<n;++i)
{
if(data[steps]>data[i])
/* To sort in descending order, change > to <. */
        {
           temp=data[steps];
                    data[steps]=data[i];
                    data[i]=temp;
                   }
                   }
printf("In ascending order: ");
for(i=0;i<n;++i)
 printf("%d  ",data[i]);
 return 0;
}
```

OUT PUT:

Enter the number of elements to be sorted: 5

1. Enter element: 12

2. Enter element: 1

3. Enter element: 23

4. Enter element: 2

5. Enter element: 0

In ascending order: 0 1 2 12 23

STRUCTURE AND UNION

Structure is a collection of variables (can be of different types) under a single name.

For example: You want to store information about a person: his/her name, citizenship number and salary. You can create different variables name, citNo and salary to store these information separately.

What if you need to store information of more than one person? Now, you need to create different variables for each information per person: name1,citNo1, salary1, name2, citNo2, salary2etc. A better approach would be to have a collection of all related information under a single name Person structure, and use it for every person

```
#include <stdio.h>

struct student

{

char name[50];

int roll;

float marks;

} s;

int main()

{

printf("Enter information:\n");

 printf("Enter name: ");

scanf("%s", s.name);
```

```
printf("Enter roll number: ");
scanf("%d", &s.roll);
printf("Enter marks: ");
scanf("%f", &s.marks);
 printf("Displaying Information:\n");
printf("Name: ");
puts(s.name);
printf("Roll number: %d\n",s.roll);
 printf("Marks: %.1f\n", s.marks);
 return 0;
}
```

```
Enter information:
Enter name: Jack
Enter roll number: 23
Enter marks: 34.5
Displaying Information:
Name: Jack
Roll number: 23
Marks: 34.5
```

Program on Union:

```
#include <stdio.h>
union Job
{
```

```
 float salary;
 int workerNo;
} j;
int main()
{
j.salary = 12.3;
j.workerNo = 100;
printf("Salary = %.1f\n", j.salary);
printf("Number of workers = %d", j.workerNo);
return 0;
}
```

DYNAMIC MEMORY ALLOCATION:

An array is a collection of fixed number of values of a single type. That is, you need to declare the size of an array before you can use it.

Sometimes, the size of array you declared may be insufficient. To solve this issue, you can allocate memory manually during run-time. This is known as dynamic memory allocation in C programming.

There are 4 library functions defined under <stdlib.h> makes dynamic memory allocation in C programming. They are malloc(), calloc(), realloc() and free().

malloc():

Syntax of malloc()

ptr = (cast-type*) malloc(byte-size)

Example:

ptr = (int*) malloc(100 * sizeof(int));

Considering the size of int is 4 bytes, this statement allocates 400 bytes of memory. And, the pointer ptr holds the address of the first byte in the allocated memory.

However, if the space is insufficient, allocation fails and returns a NULL pointer.

calloc()

The name "calloc" stands for contiguous allocation.

The malloc() function allocates a single block of memory. Whereas, calloc() allocates multiple blocks of memory and initializes them to zero.

Syntax of calloc()

ptr = (cast-type*)calloc(n, element-size);

Example:

ptr = (float*) calloc(25, sizeof(float));

This statement allocates contiguous space in memory for 25 elements each with the size of float.

free()

Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on their own. You must explicitly use free() to release the space.

Syntax of free()

free(ptr);

This statement frees the space allocated in the memory pointed by ptr.

Example 1: malloc() and free()

This program calculates the sum of n numbers entered by the user. To perform this task, memory is dynamically allocated using malloc(), and memory is freed using free() function.

```
#include <stdio.h>

#include <stdlib.h>

int main()

{

int n, i, *ptr, sum = 0;

printf("Enter number of elements: ");

scanf("%d", &n);

ptr = (int*) malloc(n * sizeof(int));

if(ptr == NULL)

{

printf("Error! memory not allocated.");

exit(0);

}

printf("Enter elements: ");
```

```
 for(i = 0; i < n; ++i)

{

scanf("%d", ptr + i);

sum += *(ptr + i);

}

printf("Sum = %d", sum);

free(ptr);

return 0;

}
```

Example 2: calloc() and free()

This program calculates the sum of `n` numbers entered by the user. To perform this task, `calloc()` and `free()` is used.

```
#include <stdio.h>

#include <stdlib.h>

int main()

{

   int n, i, *ptr, sum = 0;

   printf("Enter number of elements: ");

   scanf("%d", &n);

   ptr = (int*) calloc(n, sizeof(int));

         if(ptr == NULL)

    {

        printf("Error! memory not allocated.");

                  exit(0);
```

```
 }

printf("Enter elements: ");

for(i = 0; i < n; ++i)

{

 scanf("%d", ptr + i);

sum += *(ptr + i);

}

printf("Sum = %d", sum);

free(ptr);

return 0;

}
```

realloc():

If the dynamically allocated memory is insufficient or more than required, you can change the size of previously allocated memory using realloc() function

Syntax of realloc()

ptr = realloc(ptr, x);

Here, ptr is reallocated with new size x.

Example 3: realloc()

```
#include <stdio.h>

#include <stdlib.h>

int main()

{

int *ptr, i , n1, n2;
```

```
printf("Enter size of array: ");

scanf("%d", &n1);

 ptr = (int*) malloc(n1 * sizeof(int));

printf("Addresses of previously allocated memory: ");

for(i = 0; i < n1; ++i)

printf("%u\n",ptr + i);

printf("\nEnter new size of array: ");

scanf("%d", &n2);

 ptr = realloc(ptr, n2 * sizeof(int));

 printf("Addresses of newly allocated memory: ");

 for(i = 0; i < n2; ++i)

printf("%u\n", ptr + i);

return 0;

}
```

output will be:

```
Enter size of array: 2

Addresses of previously allocated memory:26855472

26855476

Enter new size of array: 4

Addresses of newly allocated memory:26855472

26855476

26855480

26855484
```

PROGRAM ON POINTER:

Pointers are used in C program to access the memory and manipulate the address.

If you have a variable var in your program, &var will give you its address in the memory, where & is commonly called the reference operator.

You must have seen this notation while using scanf() function. It was used in the function to store the user inputted value in the address of var.

scanf("%d", &var);

In above source code, value 5 is stored in the memory location 2686778. var is just the name given to that location.

Pointer variables

In C, you can create a special variable that stores the address (rather than the value). This variable is called pointer variable or simply a pointer.

How to create a pointer variable?

data_type* pointer_variable_name;

int* p;

Above statement defines, p as pointer variable of type int.

Reference operator (&) and Dereference operator (*)

As discussed, & is called reference operator. It gives you the address of a variable.

Likewise, there is another operator that gets you the value from the address, it is called a dereference operator *.

Below example clearly demonstrates the use of pointers, reference operator and dereference operator.

Note: The * sign when declaring a pointer is not a dereference operator. It is just a

similar notation that creates a pointer.

Example: How Pointer Works?

```
#include <stdio.h>
int main()
{
int* pc, c;
c = 22;
printf("Address of c: %u\n", &c);
printf("Value of c: %d\n\n", c);
 pc = &c;
printf("Address of pointer pc: %u\n", pc);
printf("Content of pointer pc: %d\n\n", *pc);
c = 11;
 printf("Address of pointer pc: %u\n", pc);
 printf("Content of pointer pc: %d\n\n", *pc);
*pc = 2;
printf("Address of c: %u\n", &c);
 printf("Value of c: %d\n\n", c);
return 0;
}
```

Output

```
Address of c: 2686784
Value of c: 22
Address of pointer pc: 2686784
Content of pointer pc: 22
Address of pointer pc: 2686784
```

Content of pointer pc: 11

Address of c: 2686784

Value of c: 2

Explanation of the program

int* pc, c;
Here, a pointer pc and a normal variable c, both of type int, is created.

Since pc and c are not initialized at first, pointer pc points to either no address or a random address. And, variable c has an address but contains a random garbage value.

c=22;
This assigns 22 to the variable c, i.e., 22 is stored in the memory location of variable c.
Note that, when printing &c (address of c), we use %u rather than %d since address is usually

expressed as an unsigned integer (always positive).

pc=&c;
This assigns the address of variable c to the pointer pc.
You see the value of pc is same as the address of c and the content of pc is 22 as well.

c=11;
This assigns11tovariable c.
Since, pointer pc points to the same address as c, value pointed by pointer pc is 11 as well.

*pc = 2;
This change the value at the memory location pointed by pointer pc to 2.
Since the address of the pointer pc is same as the address of c, value of c is also changed to 2.