Chapter:1 Introduction and Overview

1 DEFINATIONS:

Before going in to the depth of the subject, we will discuss few terms which is related to the subject Data and File Structure. In this section we will discuss some basic terminologies and concepts.

1.1 System

System or Computer is a machine which can perform arithmetic and logical operations at a very fast speed. It can store the data and retrieve the data or information as per users' request or instruction.

The term 'Computer' is derived from the word 'Compute' means doing calculations. But today, most of the work done by the Computer in which no computation is involved. For example, we can listen music, watching movies, doing chat with friends, making resume using word processor, sending a mail, searching something on Internet etc., do not have any type of computation. So, we can define today's computer as a data processor which takes data as input and produce information as an output.

1.2 Data

Data is unstructured raw materials and unstructured facts which will provide necessary inputs to the computer system. Here unstructured means value or set of values are not in structured format or not processed.

Data can be available in different formats. For example, numbers (34, 28, 76 etc.), numbers with decimal points (3.14, 78.65, 25.001 etc.), characters ('A', 'a', 'E' etc.), group of characters which also known as strings ("BAOU", "Computer Science", "Data and File Structure" etc.), Date, ISBN number of the book and many other types of formats are available.

Data is a value or set of values which is not processed. For example, "Ram's presence in the class on some date", "Shyam's marks of Maths subject" etc. By mean of one day attendance, we cannot predict that Ram is regular attending the class or not. Similarly, if we have marks of the Shyam of any one subject we cannot predict intelligence of Shyam.

1.3 Information

Information is structured data, or we can say - processed data is information. For example, if we record attendance of the Ram every day for whole semester and after processing it if we find his attendance is 78.20% is an information. If we collect marks of

the Shyam for all the subjects of particular semester and generate the grade sheet after processing the marks is information. So, information is nothing but collection of processed data, which produces some meaningful output.

From the above discussion it is clear that data is unstructured facts and input to the system, while information is the processed data in meaningful or summarized format produced as an output by the system. Following figure will clarify the terms ‘data’ and ‘information’ more clearly.

Fig. 1.1 System Diagram

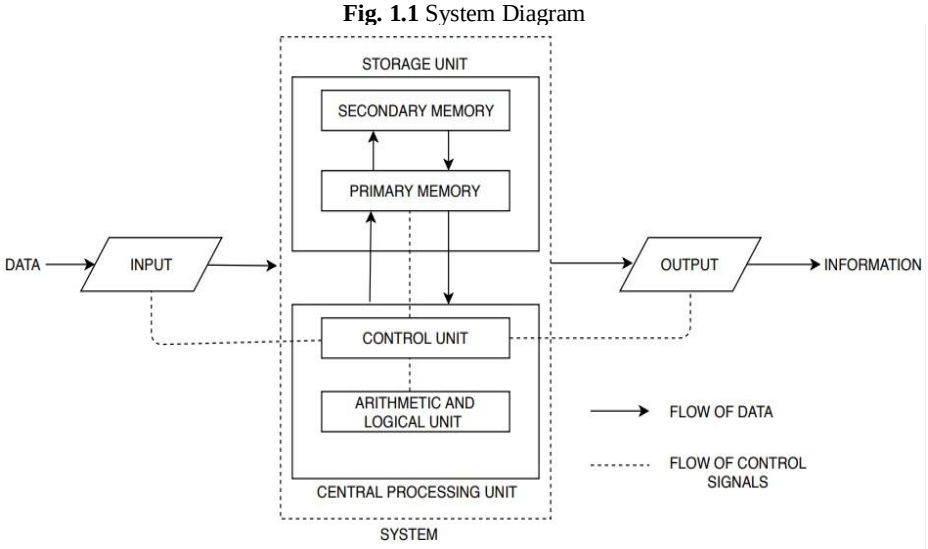


Diagram shown above is describing how system act as a data processor, which takes data from the input devices, processed it and produces information as an output. System uses storage unit, which consists of primary memory to store intermediate results and data at the time of processing and secondary storage to store information.

1.4 Data Types

Data type is used to denote a type of data we are storing in either primary or secondary memory. By specifying data type at the time of declaring a variable, we are actually specifying how much space has to be reserve by a system to store the data. In the C-Language when we declare a variable x of type int (int x;), we are instructing the system to allocate 2 bytes of memory to be given to the variable x, so that it can store any integer number in the range -32768 to 32767.

Built-in data type

Every programming language has its own set of data types. These data types are known as Built-in data types, primitive data types or system defined data types. For example.

1. C-Language has char, int, float, double etc. data types
2. Basic language has Integer, Single, Char, String etc. data types
3. Pascal language has Integer, Real, Character, Boolean etc. data types

Derived data types

Derived data types are defined by the user to extend the capability of primitive data types. For example, 'int' is primitive data type. Variable to type 'int' can hold single integer value. But if we declare any variable of type 'int *' then it will store the address of any other integer variable instead of value. So, pointer extends capability to store addresses, we can say it is derived data type. Similarly, if we declare 'int x [10]' then x variable can store 10 integer values instead of 1 value. Array is extending capability of primitive data types and allow users to store more than one values as a single unit, and it is also derived from the basic data types. So, we can say, Pointers, Arrays are derived data types.

C and C++ languages allow user to create new and customized data types, by using primitive data types which are called user defined data type. Class, Structure, Union, Enumeration are user defined data types. Because of user defined data types are also derived from the primitive data types they are also derived data types.

So, Pointer, Array, Structure, Union, Class, Enumeration are derived data types.

Abstract data types

Abstract data types are mathematical model of set of data elements that shares similar type of behavior and independent of implementation. In the problem-solving process for complex problem, it is necessary to reduce complexity. To reduce the complexity, programmer uses abstraction process. In the process of abstraction, rather than focusing on whole problem, implementation of individual functionalities is focused. For example, to implement stack or application in which stack is used, we are thinking stack as abstract (hidden or encapsulated). Regardless of what is there in the stack, we are focusing on the function how to insert an element i.e. push and how to remove the element i.e. pop. Abstract data types are also known as ADT.

So, to reduce the complexity of complex program abstract data type is used where stack, queue become abstract and how insertion and deletion is done so that LIFO and FIFO can be implemented is focused. Abstract data types are theoretical concepts and used in analysis and design, data structure, developing of system software to reduce complexity.

Data Structures

Implementation of abstract data type is called data structure. As we have discussed that abstract data types are logical, data structures are concrete or implementation. For example, List can be implemented by Array, Link List. Because of array, link list, stack, queue etc. are implementation, they are considered as data structures.

2 Introduction to Data Structures

As we know data structures are implementation, now we discuss different types of data structures.
There two types of data structure:

1. Static data structures
2. Dynamic data structures.

Static data structure has fixed size. If we declare '***int x [10]***' then array x, is a static data structure, as it is occupying fixed size of memory and not be able to store not more than 10 elements. Whereas Link List is dynamic data structure, in which memory allocation is done at runtime.
Data structures can be classified in two categories:

1. Linear data structure
2. Non-Linear data structures

In Linear data structure each element except first and last elements, fixed predecessor and successor. Array, Link List, Stack, Queue are linear data structures. In the array or Link List, apart from first and last elements every element has one previous and one next element is there. In the Non-Linear data structure element can be arranged in any desire fashion and there no restriction is there in the arrangement. Trees, Graphs are examples of non-linear data structures. Where any element of tree and graph can have n number of predecessor or successor. Linear and non-linear data structures are represented in Fig:1.2

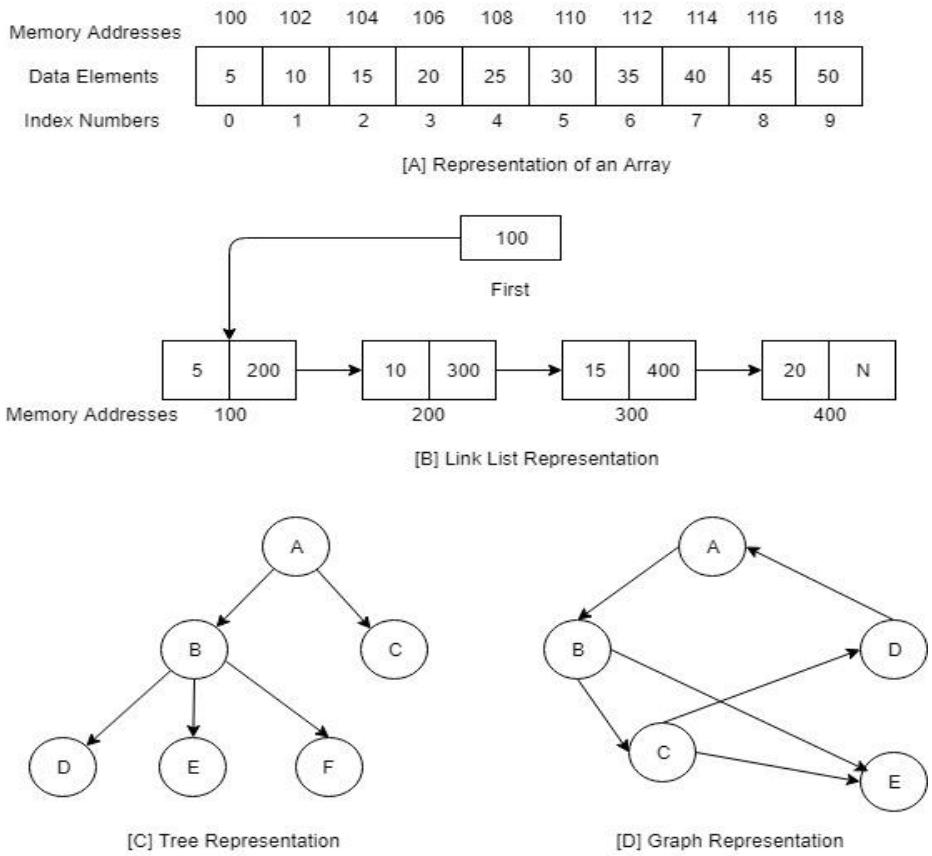
In the fig. 1.2 A-B represents linear data structures array and link list. In array and link list data elements follows strict order. In the link list we cannot visit 15 before visiting 5 and 10. Where, C-D represent non-linier data structures tree and graph. In the tree after visiting node 'A' user can visit either 'B' or 'C'. Similarly, in the graph after visiting 'B' user can visit either 'C' of 'E'. there is no strict formation is there, no order is implemented. Such data structures are non-linear data structures.

Data structure is fundamental of computer science, can be defined as a triplet of Domain, Functions and Axioms which can be defined as follows:

1. Domain: Domain refers to the set of values. For example, 0, $\pm$1, $\pm$2, $\pm$3, …. etc. For example in the design of stack if we take array of type integer then data must in the range of -32,768 to 32, 767. Domain is a set of all possible values which data structure might store as a data.
2. Functions: Function is a set of legal operations that can be performed on the data structure. For example, in the stack insertion is done by push () function and removal of the data element is done by the pop () function. User cannot insert or remove the element in the stack in any other way.
3. Axioms: Set of rules with each function is designed is called Axioms. For example, in the stack, push () is a function which will insert the value on the top, similarly of stack is not empty then data element can be removed pop () function which will remove the element from the top position.

Data structure is a triplet of Domain, Functions, and Axioms.

Fig. 1.2. Linear and Non-Linear Data Structures



3 List of data structures

As we know what is data structure, here is the list of data structures we will discuss in this book.

1. Arrays
2. Link Lists
3. Stacks
4. Queues
5. Trees
6. Graphs
7. Files

Chapter:2 Arrays

1 Introduction

Arrays are finite and ordered collection of homogeneous data. Homogeneous means same type of data. As arrays are collection of the data, we can store more than one element in the array under one common name, but make sure all the elements are of same type.

2 Understanding Arrays

2.1 Defining Array

If in the C-Language, you write '***int x [10];***' in the declarative section of the program then you have a single variable (array) named 'x' which can store 10 elements of same type ('int' in this case) for you.

2.2 Initializing Array

We can initialize arrays in three different ways:

1. Initializing array at the time of its declaration

int x [10] = {5,10,15,20,25,30,35,40,45,50};
char y [5] = {'A' , 'E' , 'I' , 'O' , 'U' };

in the above examples, 'x' is an integer array which is initialized by 10 data elements i.e. 5, 10, 15, 20, 25, 30, 35, 40, 45 and 50. First data element '5' will be stored on the first 0^{th} position of the array x. That means x [0] is '5'. Similarly, 10 is stored on 1^{st} position, 15 is stored on 2^{nd} position and so on. Finally, data element 50 will be stored on 9^{th} position. In C-Language array index starts from 0 so last element we can find on the passion Size – 1. Here the size of array 'x' is 10 so last data element '50' will be stored on passion 9.

In the another example we have declared an array 'y' of type character, which is initialize by five character data elements 'A' , 'E' , 'I' , 'O' and 'U'. First data element 'A' will be stored on 0^{th} position and 'U' will be stored on 4^{th} position.

2. Declaring array and initializing it with some static values:

int x [5];
char y [5];
x [0] =5;

x [1] = 10;
x [2] = 15;
and so on. Similarly, array 'y' can be initialized as,
y [0] = 'A';
y [1] = 'E';
and so on.

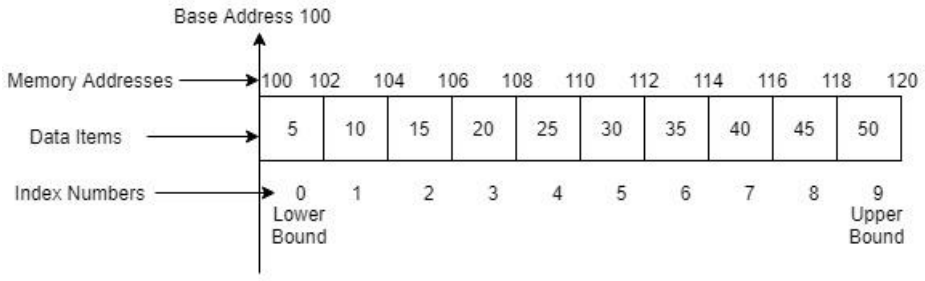
3. Declaring and initializing array by user input

```
int x [10] , i ;
for (i=0; i< 10; i++)
{
   printf (" Enter Any Number:");
   scanf("%d", & x[i]);
}
```

2.3 Terminologies

1. Type: Type of an array represents type of data can be stored in the array. For example, array can be of type int, float, char, double etc.
2. Size: Size of an array represent number maximum data elements that can be accommodated in the array. For example, if we declare 'int x [10]' then array 'x' can accommodate maximum 10 elements. So, we can say size of array x is 10.
3. Index: Each element stored in the array has unique reference number is called index number. It is denoted in the square brackets like [ ]. As we have discussed in the above example x [0] = 5, means data element 5 is stored in at the index number 0 of the array 'x'.
4. Range: Range is a set of all valid index numbers. We know that in C-Language array index starts from 0, it is called Lower bound of the array. If we declare array 'int x [10];' then last element we can place on 9^{th} position. It is called upper bound of an array. Set of all possible integers from lower bound to upper bound is called range of an array. So, the range of array x is 0 to 9.
5. Base: Starting memory location of an array is called base address of an array. In C-Language array name itself refers base address of an array.

Fig.2.1 Array Terminologies



In the fig.2.1 memory representation of an integer single dimension array is shown. Array of 10 integer values are stored in the consecutive memory location starting 100 to 120. 100 is the starting address of the array is called base address. Because type of an array is integer and we know each integer occupies 2 bytes of space in the memory first data element of the array i.e. '5' will be stored from memory address 100 to 102. Second value 10 will be stored from 102 to 104 and so on. Index number of first data element 5 is 0 is called lower bound of the array and similarly last element is stored on position 9 is called upper bound of an array.

Let is we have an array having Lower bound LB and upper bound UB. Then the index number of i^{th} element will be always: Index (Xi) = LB + i -1. For example, in C-Language index number of 5th element will be 0 + 5 -1 =4.

Similarly, size of an array is: Size = UB – LB – 1. In C-Language if UB of and array is 9 the size = 9 – 0 + 1 =10.

The following Fig. 2.1 will explain all the terms discussed above. Array can be One – dimensional (Linear), Two – dimensional (Matrix) or Multi – dimensional. One – dimensional and Two – dimensional arrays are discussed below in details.

3 One-Dimensional Array

One-dimensional array is an array which requires only one index or subscript number to reference any element stored in the array. Array presented in the Fig.2.1 is one-dimensional array. Value and Address of any element having index 'i', in the array 'X' can be found by following equations:

$$\text{Value of element can be accessed } X[i] \tag{1}$$

$$\text{Address of } X[i] = \text{Base address of an array} + i * \text{size of element} \tag{2}$$

For example, in the integer array whose base address is 100, address of element located on index number 4 is: 100 + 4 * 2=108. Here 100 is the base address of an array, 4 is the index number of an element and because of the type on an array is integer size of element will be 2.

3.1 Operations on arrays

Different types of operations can be performed on the array such as, Traversing, Insertion, Deletion, Searching, Sorting, Merging etc.

Traversing:

Accessing each element of an array is called traversing operation. Consider the following program in which we are printing all the values stored by the user.

```
#include<stdio.h>
void main()
{
   int x[10]={2,9,11,28,34,45,57,78,83,90};
   int i;
   printf("\nArray X contains: ");
   for(i=0;i<10;i++)
   {
      printf("%d\t",x[i]);
   }
}
```

Program:2.1

Output:
Array X contains: 2 9 11 28 34 45 57 78 83 90

In the above program we have declare an integer array 'x' having 10 integer values. Using variable 'i' and a for loop we have printed each element of the array. Accessing all the elements of the array for printing or any other calculation purpose is called traversing the array.

Insertion in the array:

Consider an array having some values are inserted and other cells are empty (having value 0). Now suppose user want to insert a new data element on potion 4. In this case we have to shift 9^{th} element of an array to 10^{th} position, 8^{th} element have to shift on 9^{th} position and in that way all the elements of the array up to 4^{th} position, has to be shifted. Finally, the newer element has to be placed on the position number 4. The following Fig.2.2 shows how the elements are shifted to make a space for the new element and how the new element is inserted in the array. Last element of an array (i.e. 0 in our case) will be discarded.

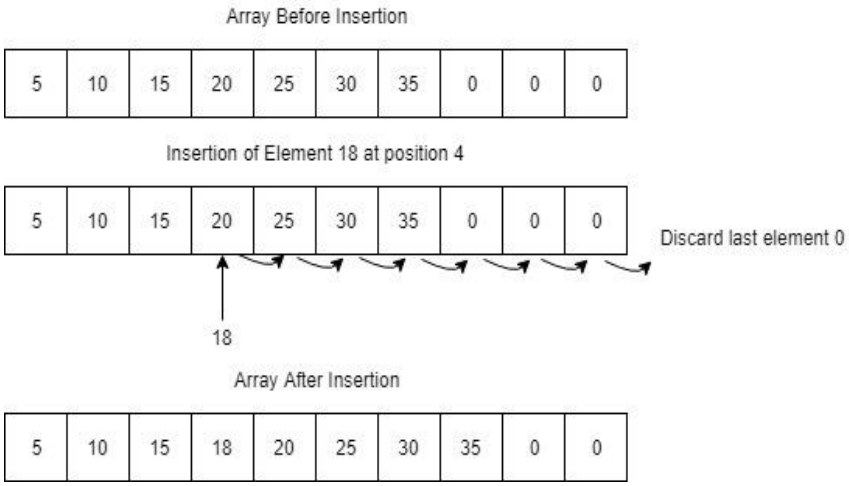


Fig: 2.2 Insertion in the array.

Program to insert an element in the is given below. In this program some elements are already filled, whereas some elements are empty. In the program empty elements are considered as 0. Program takes two inputs from the user. It takes position where new element to be inserted and element to be inserted. Program needs to shift all elements from that position to end. Last element of an array has to be discarded.

```
#include<stdio.h>
#define MAX 10
void main()
{
   int arr[MAX]={5,10,15,20,25,30,35, 0,0,0}, i, pos, val;
   printf("\nValues in the Array:");
   for(i=0;i<MAX && arr[i]!=0;i++)
      printf("\n%d", arr[i]);
   printf("\nEnter Position where you want to Insert the value:");
   scanf("%d", &pos);
   printf("Enter Value:");
   scanf("%d", &val);
   for(i=MAX-1;i>=pos;i--)
      arr[i]=arr[i-1];
   arr[i]=val;
   printf("\nValues in the Array after Insertion:");
   for(i=0;i<MAX && arr[i]!=0;i++)
      printf("\n%d", arr[i]);
}
```

Output:

```
Values in the Array:
5
10
15
20
25
30
35
Enter Position where you want to Insert the value:4
Enter Value:18
Values in the Array after Insertion:
5
10
15
18
20
25
30
35
```

Removing an element from an array:

Removing an element from an array requires reverse process than previous. In the previous example we have shifted all elements by one position to the right. In the removal, we have to shift all elements of an array to the left. At the last position we have to add 0 (Empty) element. Removal process in the array is shown as below:

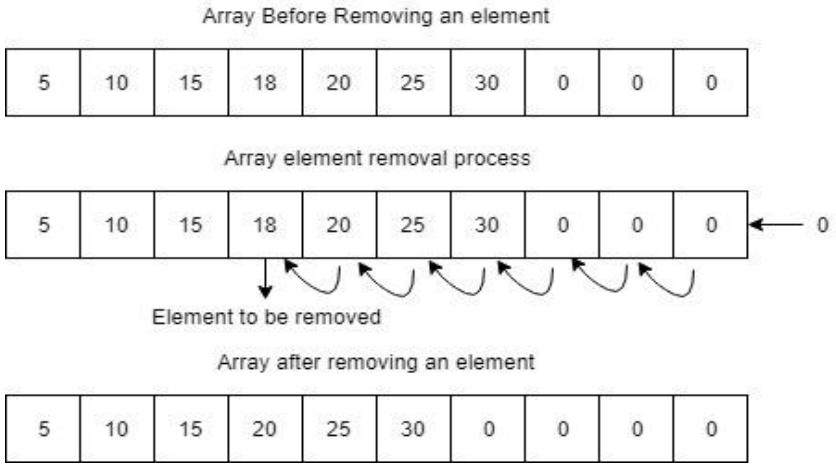


Fig: 2.3 Removing an element from an array.

Program to remove an element is given below, where user will position and program will remove an element from that position, shift all other elements to the left, append 0 at the end and show all elements of an array after removing an element.

```
#include<stdio.h>
#define MAX 10
void main()
{
   int arr[MAX]={5,10,15,18,20,25,30,0,0,0}, i, pos;
   printf("\nValues in the Array:");
   for(i=0;i<MAX && arr[i]!=0;i++)
      printf("\n%d", arr[i]);
   printf("\nEnter Position where you want to Remove an Element:");
   scanf("%d", &pos);
   for(i=pos-1;i<MAX-2;i++)
      arr[i]=arr[i+1];
   arr[MAX-1]=0;
   printf("\nValues in the Array:");
   for(i=0;i<MAX && arr[i]!=0;i++)
      printf("\n%d", arr[i]);
}
```

Output:

Values in the Array:
5
10
15
18
20
25
30
Enter Position where you want to Remove an Element:4
Values in the Array:
5
10
15
20
25
30

Searching an element from an Array:

Searching is the process, where we scan the array for some specific element and return the position of that element in the array. If the value, which user is looking for, is not there in the array, we have to show the message that “Value does not Exists:”.

Searching in the array can be done by two methods [1] Linear Search and [2] Binary search. Both searching algorithms are described below:

[1] Linear Search: Linear search is the process, where we are comparing an element to searched with each element of an array. Will start from index number 0 and go up to last element. If we find search element (i.e. search_element==array[i]) then we print the value of that index number by adding 1 (As we know array index starts from 0, but for the user prospective first element of the array will be on position 1). If we compare all elements of an array with the search element, but didn't match any element of an array, we will print the message that "Search Element do not Exists in the Array:".

Linear search can be applied on any type of array (i.e. Sorted or Unsorted). But the major drawback of it is, we have to compare each element of an array with search element, which takes longer time (algorithm is slower). In worst case, we have to make 'n' comparisons. Therefore, the worst-case complexity of this algorithm is O(n).

The program and its output of linear search program is given below:

```
/* Program of Linear Search (Can be used on Any Array Sorted or Unsorted) */
#include<stdio.h>
void main()
{
   int x[10];
   int i,se;
   // Accepting values for the Array from the User
   for(i=0;i<10;i++)
   {
      printf("Enter Value:");
      scanf("%d",&x[i]);
   }
   // Taking Search Element from the user
   printf("Enter Search Element:");
   scanf("%d",&se);
   for(i=0;i<10;i++)
   {
      if(x[i]==se)
      {
         printf("\nValue found on %d position", i+1);
         break;
      }
   }
   if(i==10)
   {
      printf("\nValue Does not Exists:");
   }
}
```

Output:

```
Enter Value:76
Enter Value:45
Enter Value:32
Enter Value:67
Enter Value:2
Enter Value:97
Enter Value:28
Enter Value:42
Enter Value:57
Enter Value:65
Enter Search Element:28
Value found on 7 position
```

[2] Binary Search: In Linear search algorithm, we are comparing search element with all elements of the array sequentially. This process is time consuming and if the array has 'N' elements, then in worst case we need to do 'N' comparisons. Binary search algorithm can be applied only on sorted array. Here, we are comparing the search element directly to the middle element of an array. If the search element is small then the value can be from 0 to mid -1 position, and if the search element is greater than the value at middle position then it can be from mid+1 to MAX -1 position. If we do the first comparison, half elements will become out from the comparison. If continues the same process again and again we can find search elements.

Here, number of comparisons are reduced, and therefore Binary search is the faster algorithm compare to Linear search algorithm. The complexity of Binary algorithm is: O (Log n). The following tracing will show how the algorithm is working:

Consider an array of 10 elements as shown below, and we want to search element 35:

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Value	5	10	15	20	25	30	35	40	45	50

We will take three variables beg (beginning), end and mid (middle). Initial values of variable beg=0 as array starts from position 0 and end =9 as last index number of an array is 9.

Iteration:1

Mid = (beg + end) /2 = (0+9) / 2= 4.5 =4 (Integer)

On 4^{th} position of an array value is 25, so 25 is compared to 35.

Because 35 is greater the than 25, search element can be there from 5^{th} to 9^{th} position. So, for second iteration we have to start from 5^{th} position (beg=mid+1) therefore, beg=5.

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Value	5	10	15	20	25	30	35	40	45	50
	beg				mid					end

Iteration:2
Mid = (beg + end) /2 = (5+9) / 2 = 7
On 7th position of an array value is 40, so 40 is compared to 35.
Because 35 is smaller than 40, search element can be there from 5th to 6th position. So, for third iteration we have to start from 5th position to 6th position (end=mid-1) therefore, end=6.

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Value	5	10	15	20	25	30	35	40	45	50
						beg		mid		end

Iteration:3
Mid = (beg + end) /2 = (5+6) / 2 = 5.5 = 5 (Integer)
On 5th position of an array value is 30, so 30 is compared to 35.
Because 35 is greater than 30, search element can be there from 6th to 6th position. So, for fourth iteration we have to start from 6th position to 6th position (beg=mid+1) therefore, beg=6.

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Value	5	10	15	20	25	30	35	40	45	50
						beg mid	end			

Iteration:4
Mid = (beg + end) /2 = (6+6) / 2 = 6
On 6th position of an array value is 35, so 35 is compared to 35.
Here the value at mid position (6th position) is 35 is equal to search element 35.

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Value	5	10	15	20	25	30	35	40	45	50
							beg mid end			

See, in this example we have searched 35 in just 4 comparisons, on the other hand if we use Linear search algorithm, then it takes 7 iterations. If the array has 1 lacs of elements, then using binary search algorithm we can find any element in less than 15 comparisons. In every iteration half values of an array will become out from the competition. The major drawback of Binary search algorithm is, it can be applied only on sorted array. If the array is unsorted then we have only one choice that is Linear search. Following program takes 10 inputs from the user and perform binary search to locate search element entered by the user.

```
/* Program of Binary Search (Can be used only on Sorted Array) */
#include<stdio.h>
void main()
{
   int x[10];
   int i,beg,end, mid,val;
   // Accepting values for the Array from the User
   printf("Enter Array elements in Sorted order:\n");
   for(i=0;i<10;i++)
   {
      printf("Enter Value:");
      scanf("%d",&x[i]);
   }
   printf("\nEnter Search Element:");
   scanf("%d",&val);
   //Searching Array elements in the array
   beg=0;
   end=9;
   for(mid=(beg+end)/2;beg<=end;mid=(beg+end)/2)
   {
      if(x[mid]==val)
      {
         printf("\nValue found on %d position:",mid+1);
         break;
      }
      else if(x[mid]<val)
      {
         beg=mid+1;
      }
      else
      {
         end=mid-1;
      }
   }
   //If value does not Exists
   if(beg > end)
   {
      printf("\nValue does not Exists:");
   }
}
```

Output:

Enter Array elements in Sorted order:
Enter Value:5
Enter Value:10
Enter Value:15

Enter Value:20
Enter Value:25
Enter Value:30
Enter Value:35
Enter Value:40
Enter Value:45
Enter Value:50
Enter Search Element:35
Value found on 7 position:

Sorting elements of an Array:
Sorting is the process of arranging array elements either in ascending order (from lower to higher) or descending (from higher to lower). To sort an array different algorithm can be used which are listed below:

1. Selection sort
2. Bubble sort
3. Insertion sort
4. Quick sort
5. Merger sort

Selection Sort:
In selection sort algorithm each element of an array from index number 1 to MAX-1 (last index number of an array) will be compared with 0^{th} element of an array. If an element if smaller than the elements located on 0^{th} position then will swap those elements and will put the smaller element on 0^{th} position. When we complete all the iterations from 1 to MAX -1, will get the smallest element at position 0. The same process is repeated and all the element from 2 to MAX-1 will be compared with 1^{st} element of an array. If the element is smaller than it will be swapped with 1^{st} element and we can ger second smaller element on 1^{st} position of an array. If we continue this process till MAX -2 position the we can get sorted array.
Following example will shows the tracing on selection sort algorithm. To do the tracing consider an array of 5 elements as shown below:

Iteration 1: Value of I =0 and J=I+1=0+1=1		
J=1	**57** **23** 75 19 10	Here I^{th} element 57 is compared with J^{th} element that is 23. Because of 23 is smaller than 57, will exchange the values of I^{th} and J^{th} element.
J=2	**23** 57 **75** 19 10	Here I^{th} element 23 is compared with J^{th} element that is 75. Because of J^{th} element 75 is bigger than I^{th} element, there is no swap
J=3	**23** 57 75 **19** 10	Here I^{th} element 23 is greater than J^{th} element, that is 19. Because of J^{th} element is smaller than I^{th} element then we will swap I^{th} and J^{th} elements.
J=4	**19** 57 75 23 **10**	Here again J^{th} element 10 is smaller that I^{th} element, so after swapping we get...

	10 57 75 23 19	

After completion we get the smallest element at the beginning of an array that is:10

Iteration 2: Value of I =1 and J=I+1=1+1=2		
J=2	**10 57 75** 23 19	57 is smaller than 75, so no swap
J=3	**10 57** 75 **23** 19	23 is smaller than 57, so will swap the values
J=4	**10 23** 75 57 **19**	19 is smaller than 23, so will swap the values
	10 19 75 57 23	So, at the end of the second iteration will get second smaller value 19 at 1st position.

Iteration 3: Value of I =2 and J=I+1=2+1=3		
J=3	**10 19 75 57** 23	57 is smaller than 75, so will swap the values
J=4	**10 57 57** 75 **23**	23 is smaller than 57, so will swap the values
	10 19 23 75 57	So, at the end of the third iteration will get third smaller value 23 at 2nd position.

Iteration 4: Value of I =3 and J=I+1=3+1=4		
J=4	**10 19 23 75 57**	57 is smaller than 75, so will swap the values
	10 19 23 57 75	So, at the end of the fourth iteration will get fourth smaller value 57 at 3rd position.

And that way we get, sorted array. In this algorithm I^{th} value is compared with J^{th} value if J^{th} value is smaller than I^{th} value then will swap both the element. Program to implement selection sort is given below:

```
/* Program of Selection Sort */
#include<stdio.h>
void main()
{
   int x[5];
   int i,j,tmp;
   // Accepting values for the Array from the User
   for(i=0;i<5;i++)
   {
      printf("Enter Value:");
      scanf("%d",&x[i]);
   }
   //Printing unsorted array
   printf("\n Array Before Sorting:\n");
   for(i=0;i<5;i++)
      printf("\t%d",x[i]);
   //Sorting an array
   for(i=0;i<4;i++)
   {
```

```
    for(j=i+1;j<5;j++)
    {
      if(x[i]>x[j])
      {
        //If Ith element is Greater than Jth element then swap the value
        tmp=x[i];
        x[i]=x[j];
        x[j]=tmp;
      }
    }
  }
  //Printing Sorted Array
  printf("\n Array After Sorting:\n");
  for(i=0;i<5;i++)
    printf("\t%d",x[i]);
}
```

Output:

```
Enter Value:57
Enter Value:23
Enter Value:75
Enter Value:19
Enter Value:10

 Array Before Sorting:
      57     23     75     19     10
 Array After Sorting:
      10     19     23     57     75
```

Note: The program given above will sort the array in the ascending order, If you want to sort the array in descending order then instead of 'if(x[i]>x[j])' statement use 'if(x[i]<x[j])', which will sort the array in descending order.
The worst-case complexity of Selection sort algorithm is: $O(n^2)$.

Bubble Sort:

In the selection sort algorithm, we have compared I^{th} and J^{th} element, and we have swapped the values if J^{th} element is smaller the I^{th} element. In the Bubble sort algorithm we will compare J^{th} and $(J+1)^{th}$ element. If $(J+1)^{th}$ element is smaller than J^{th} element then we will swap both the elements. In the first iteration of the selection sort algorithm we are getting smaller element at the beginning of an array, similarly in the bubble sort algorithm we are getting the biggest element at the end of an array.

Following example will shows the tracing on bubble sort algorithm. To do the tracing consider an array of 5 elements as shown below:

Iteration 1: Value of I =0		
J=0	**57 23** 75 19 10	Here $(J+1)^{th}$ element 23 is compared with J^{th} element that is 57. Because of 23 is smaller than 57, will exchange the values of $(J+1)^{th}$ and J^{th} element.
J=1	23 **57 75** 19 10	If J=1 then J+1=2; 1st and 2nd elements are compared. 1st element is smaller so swapping is not done.
J=2	23 57 **75 19** 10	If J=2 then J+1=3; 3rd element 19 is smaller than 2nd element 57 so will swap both values.
J=3	23 57 19 **75 10**	If J=3 then J+1=4; 3rd and 4th value will be compared, because 10 is smaller than 75, will swap the values and we can get the largest number 75 at the end of the list.
	23 57 19 10 **75**	

Iteration 2: Value of I =1		
J=0	**23 57** 19 10 **75**	Here $(J+1)^{th}$ element 57 is compared with J^{th} element that is 23. Because of 23 is smaller than 57, will not swap the values.
J=1	23 **57 19** 10 **75**	If J=1 then J+1=2; 1st and 2nd elements are compared. 1st element 57 is greater than 2nd element 19. So, will swap the values.
J=2	23 19 **57 10** **75**	If J=2 then J+1=3; 3rd element 10 is smaller than 2nd element 57 so will swap both values.
	23 19 10 **57** **75**	Here we get the second largest element 57 at the second last position of an array.

Iteration 3: Value of I =2		
J=0	**23 19** 10 **57** **75**	Here $(J+1)^{th}$ element 19 is compared with J^{th} element that is 23. Because of 19 is smaller than 23, will swap the values.
J=1	19 **23 10** **57** **75**	If J=1 then J+1=2; 1st and 2nd elements are compared. 1st element 23 is greater than 2nd element 10. So, will swap the values.
	19 10 **23** **57** **75**	Here we get the third largest element 23, placed on third last position of an array.

Iteration 3: Value of I =2		
J=0	**19 10** **23** **57** **75**	Here $(J+1)^{th}$ element 10 is compared with J^{th} element that is 19. Because of 10 is smaller than 19, will swap the values.
	10 19 **23** **57** **75**	Here we can get the sorted array.

The program to implement bubble sort is given below:

```
/* Program of Bubble Sort */
#include<stdio.h>
void main()
{
   int x[5];
   int i,j,tmp;
   // Accepting values for the Array from the User
   for(i=0;i<5;i++)
   {
      printf("Enter Value:");
      scanf("%d",&x[i]);
   }
   //Printing unsorted array
   printf("\n Array Before Sorting:\n");
   for(i=0;i<5;i++)
      printf("\t%d",x[i]);
   //Sorting an array
   for(i=0;i<5;i++)
   {
      for(j=0;j<5-i;j++)
      {
         if(x[j]>x[j+1])
         {
            //If Jth element is Greater than J+1th element then swap the value
            tmp=x[j];
            x[j]=x[j+1];
            x[j+1]=tmp;
         }
      }
   }
   //Printing Sorted Array
   printf("\n Array After Sorting:\n");
   for(i=0;i<5;i++)
      printf("\t%d",x[i]);
}
```

Output:

```
Enter Value:57
Enter Value:23
Enter Value:75
Enter Value:19
Enter Value:10
 Array Before Sorting:
      57     23     75     19     10
 Array After Sorting:
      10     19     23     57     75
```

Note: The program discussed above will sort all the array elements in the ascending order. If you write the statement '***if(x[j]<x[j+1])***' statement instead of ***if(x[j]>x[j+1])***' then you can get sorted array in the descending order.
The worst-case complexity of Bubble sort algorithm is: $O(n^2)$.

Insertion Sort:
In the Selection sort and Bubble sort algorithm we are swapping two elements, whenever we are getting smaller element at right hand side. In the Insertion sort algorithm whenever we are getting smaller element at the right-hand side, we checking all the left-hand side elements from it and will insert that element at a proper place. If we are inserting element, all other elements are shifter towards right-hand side. Following example will show the tracing of an Insertion sort algorithm.

Iteration 1: Value of I =1, J=I=1		
J=1	**57 23** 75 19 10	We are comparing I^{th} element 23 with $(J-1)^{th}$ element that is 57. Here I^{th} element 23 is smaller so it is inserted before 57. If we want to insert 23 before 57, we have to shift 57 to 1^{st} position from 0^{th} place.
	23 57 75 19 10	So, after first iteration we get this array.

Iteration 2: Value of I =2, J=I=2		
J=1	23 **57 75** 19 10	Here $(J-1)^{th}$ element 57 is compared with I^{th} element that is 75. Because I^{th} element 75 is not smaller than $(J-1)^{th}$ element. So, array will remain unchanged.
J=0	**23** 57 **75** 19 10	Here $(J-1)^{th}$ element 23 is compared with I^{th} element that is 75. Because I^{th} element 75 is not smaller than $(J-1)^{th}$ element. So, array will remain unchanged.
	23 57 75 19 10	So, after Second iteration we get this array

Iteration 3: Value of I =3, J=I=3		
J=3	23 57 **75 19** 10	Here $(J-1)^{th}$ element 57 is compared with I^{th} element that is 19. Because I^{th} element 19 is smaller than $(J-1)^{th}$ element. So, we scan all element of left-side of 19 to find proper place of 19.
J=2	23 **57** 75 **19** 10	Here I^{th} element 19 is smaller than $(J-1)^{th}$ element 57.
J=1	**23** 57 75 **19** 10	Here I^{th} element 19 is smaller than $(J-1)^{th}$ element 23. Now, J-1 is 0, So, 19 has to be inserted before 23. So, all elements from 23 to 75 will be shifted to right-side by 1 position and 19 is inserted before 23, that is 0^{th} place.
	19 23 57 75 10	So, after Third iteration we get this array

Iteration 3: Value of I =4, J=I=4		
J=4	19 23 57 **75** **10**	Here $(J-1)^{th}$ element 75 is compared with I^{th} element that is 10. Because I^{th} element 10 is smaller than $(J-1)^{th}$ element. So, we scan all element of left-side of 10 to find proper place of 10.
J=3	19 23 **57** 75 **10**	Here I^{th} element 10 is smaller than $(J-1)^{th}$ element 57.
J=2	19 **23** 57 75 **10**	Here I^{th} element 10 is smaller than $(J-1)^{th}$ element 23.
J=1	**19** 23 57 75 **10**	Here I^{th} element 10 is smaller than $(J-1)^{th}$ element 19. Now, J-1 is 0, So, 10 has to be inserted before 19. So, all elements from 19 to 75 will be shifted to right-side by 1 position and 10 is inserted before 19, that is 0^{th} place.
	10 19 23 57 75	Finally, in the fourth iteration we get the sorted array.

Programing implementation of the Insertion sort algorithm is given below. The worst-case complexity of Bubble sort algorithm is: $O(n^2)$.

```
/* Program of Insertion Sort */
#include<stdio.h>
void main()
{
   int x[5];
   int i,j,tmp;
   // Accepting values for the Array from the User
   for(i=0;i<5;i++)
   {
      printf("Enter Value:");
      scanf("%d",&x[i]);
   }
   //Printing unsorted array
   printf("\n Array Before Sorting:\n");
   for(i=0;i<5;i++)
      printf("\t%d",x[i]);
   //Sorting an array (Remember the loop of I starts from 1
   for(i=1;i<5;i++)
   {
      //Copying x[i] element to variable tmo
      tmp=x[i];
      /* If J is greater than 0 and x[J-1]th element is greater than tmp then bring J-
1th element to Jth position.
      Remember the loop of J starts from I and J will be decremented on every itera-
tion. */
```

```
    for(j=i;j>0 && x[j-1]>tmp; j--)
       x[j]=x[j-1];
    //place tmp to proper position that is Jth position
    x[j]=tmp;

  }
  //Printing Sorted Array
  printf("\n Array After Sorting:\n");
  for(i=0;i<5;i++)
    printf("\t%d",x[i]);
}
```

Output of the program will remain same as given in Selection sort and Bubble sort program.

Quick Sort:

In the quick sort algorithm, we are considering one element as a pivot element and we try to place that element to its proper place in the array. That means after placing pivot elements to its proper place all small elements has to be at the left side of the pivot element and larger elements placed at the right side of an array. If we do the same process in the set of smaller values, and set of larger elements then every element will be placed on its proper position and we get sorted array. Tracing of first Iteration of Quick sort algorithm is given below:

Consider an array having 10 elements as shown in the figure given below:

11	2	9	13	57	25	17	1	90	3

Now, first element of an array is considered as pivot element. So, 11 will be a pivot element. Will take two variable 'p' and 'q' initialize as 1 and 9 as shown below, last row of the table showing index number of each element in the array:

11	2	9	13	57	25	17	1	90	3
Pivot	p								q
0	1	2	3	4	5	6	7	8	9

Now, run a loop for variable 'p' compare p^{th} element of an array with pivot element. If the element is smaller the increment variable p by 1. If value of p^{th} element is greater than pivot element the break the loop.

Here 2 and 9 are smaller than pivot element 11, and 13 is greater than pivot element 11, so loop of variable 'p' quit here and variable 'p' has value 3.

11	**2**	**9**	**13**	57	25	17	1	90	3
Pivot			p						q
0	1	2	3	4	5	6	7	8	9

In the same way start the loop of variable ‘q’ from 9 and compare q^{th} element of an array with pivot element. If the q^{th} element of an array is greater than pivot element then decrements variable q by 1. If the value of q^{th} element is smaller than pivot element the quit the loop of variable q.
Here qth element is 3 which is smaller than pivot element 11. So, the loop of variable q will quit on its first iteration.

11	**2**	**9**	**13**	57	25	17	1	90	**3**
Pivot			p						q
0	1	2	3	4	5	6	7	8	9

Now, when the loop of both the variable completes, swap the pt^{h} and qt^{h} variable of an array.

11	**2**	**9**	**3**	57	25	17	1	90	**13**
Pivot			p						q
0	1	2	3	4	5	6	7	8	9

Repeat the same process again run the loop of variable ‘p’ until will not get the greater value than pivot. So, when p becomes 4 then loop is terminated as 57 is greater than pivot 11. Run the loop for variable q until will not get smaller element then pivot. Here 13, 90 are greater than pivot, and next element 1 is smaller than pivot. So, loop of variable q will stop on index number 7 and value 1.

11	**2**	**9**	**3**	**57**	25	17	**1**	**90**	**13**
Pivot				p			q		
0	1	2	3	4	5	6	7	8	9

Again, swap p^{th} and q^{th} element.

11	**2**	**9**	**3**	**1**	25	17	**57**	**90**	**13**
Pivot				p			q		
0	1	2	3	4	5	6	7	8	9

Again, start the loop of p until we get higher value then 11. So, loop of p will stop on index 5, as it has value 25. Start the loop of q until we get the smaller element then pivot. So, q loop will be stopped when q=4, because 57, 17 25 are greater than 11 and 1 is smaller than 11.

11	**2**	**9**	**3**	**1**	**25**	**17**	**57**	**90**	**13**
Pivot				q	p				
0	1	2	3	4	5	6	7	8	9

When, p > q, then swap q^{th} element with pivot.

1	2	9	3	**11**	25	17	57	90	13
Pivot				q	p				
0	1	2	3	4	5	6	7	8	9

Here pivot element 11 is placed on its proper place as all the left-hand side elements are smaller than 11, and right-hand side elements are greater than 11. Now, pivot is placed on position 4, so do the same process recursively on the array from 0 to 3, and 5 to 9. If we continue this process then all elements will set to their proper place in the array and we get the sorted array.
Programmatically implementation of quick sort algorithm is given below:

```
/* Quick Sort */
#include <stdio.h>
//Prototype declaration of Function split and quick sort
int split ( int*, int, int ) ;
void quicksort ( int *, int , int);
//Main function
void main( )
{
   int arr[10] = { 11, 2, 9, 13, 57, 25, 17, 1, 90, 3 } ;
   int i ;
   printf ( "\nArray before sorting:\n") ;

   for ( i = 0 ; i <= 9 ; i++ )
      printf ( "%d\t", arr[i] ) ;
//Calling quicksort function and passing base address of the array, start and end index of the array
   quicksort ( arr, 0, 9 ) ;

   printf ( "\nArray after sorting:\n") ;

   for ( i = 0 ; i <= 9 ; i++ )
      printf ( "%d\t", arr[i] ) ;
}

void quicksort ( int *a, int lower, int upper )
{
   int i ;
   if ( upper > lower )
   {
      i = split ( a, lower, upper ) ;
      quicksort ( a, lower, i - 1 ) ; //Recursion of quick sort function from lower to mid -1
      quicksort ( a, i + 1, upper ) ; //Recursion of quick sort function from mid+1 to upper
   }
}
```

```
int split ( int a[ ], int lower, int upper )
{
   int pivot, p, q, t ;

   p = lower + 1 ;
   q = upper ;
   pivot = a[lower] ;
//Variable I which is initialized as 0th element is a pivot element
//Start loop of P variable from 1 and loop of Q from 9

   while ( q >= p )
   {
      //if Pth element is smaller than pivot element then continues the loop
      while ( a[p] < pivot )
         p++ ;
     //If Qth element is greater than pivot element then continues the loop
      while ( a[q] > pivot )
         q-- ;
     //When program comes out of both loop swap Pth and Qth elements
      if ( q > p )
      {
         t = a[p] ;
         a[p] = a[q] ;
         a[q] = t ;
      }
   }
//Finally swap Qth element with pivot element and return the value of Q
   t = a[lower] ;
   a[lower] = a[q] ;
   a[q] = t ;

   return q ;
}
```

Output:

Array before sorting:
11 2 9 13 57 25 17 1 90 3
Array after sorting:
1 2 3 9 11 13 17 25 57 90

Merge Sort:

Merge sort algorithm sort an array using following three steps:

1. **Divide:** In this process, we are recursively split the array in two part. We find middle, that is average of lower bound and upper bound of an array, and split the array from the middle. So, array of n elements is split into two arrays of

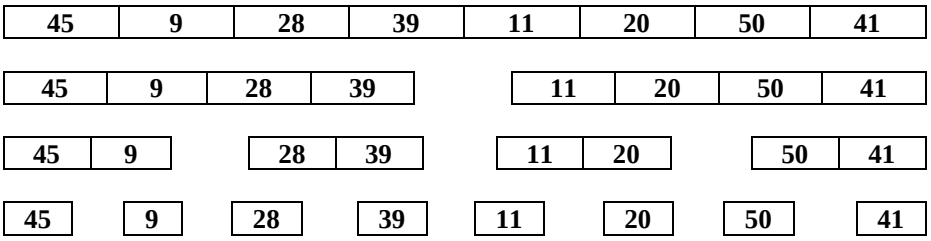
size n/2. We continue this process on the sub-arrays of size n/2 recursively till each element of an array is separated.

2. **Conquer:** In this process we perform sorting, two sub arrays recursively using merge sort.
3. **Combine:** In this process we merge two sorted array of size n/2, to produce array of size n, in such a way that the resultant array will be in sorted order.

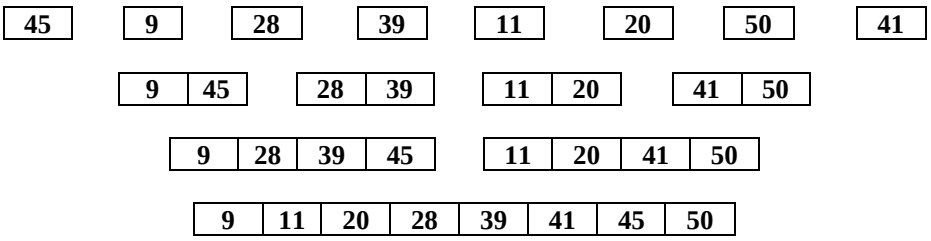
To understand these processes discussed above, consider the following example.

45	9	28	39	11	20	50	41

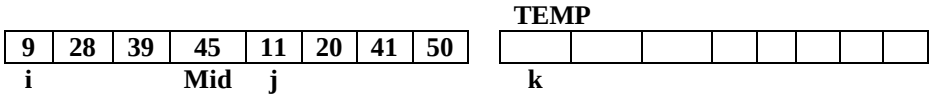
Consider the array given above. We want to split the array using merger sort algorithm. So, first we use divide method to split the array of n elements in to two sub-array of size n/2 recursively as shown below:



Once, we split the array recursively till, each elements of an array gets separated, we will start another process of merging and sorting the array as shown below:

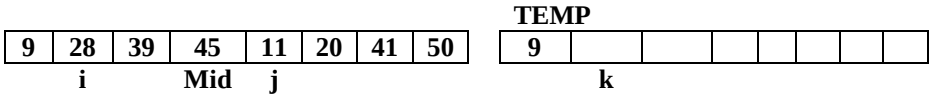


To understand the merging process in details, consider the following example:
In the merging process we have an array of n elements, divided into two sub arrays of size n/2. Both the sub-arrays are sorted array. To produce sorted merge array from the given two sorted sub-arrays of size n/2, we take an additional array TEMP. We take variable i=0 and j=mid+1, to reference first elements of both sorted sub-arrays of size n/2.

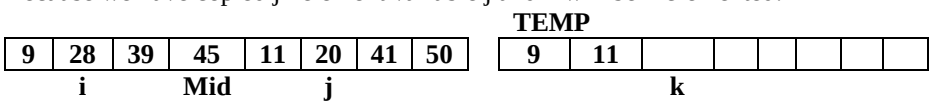


Consider an array contains two sorted sub arrays. From Index number 0 to Mid that is 3, we have one sorted sub-array having elements 9, 28, 39 and 45. Similarly from Mid+1 that is 4 to end that is 7 another sorted sub-array having elements 11, 20, 41 and

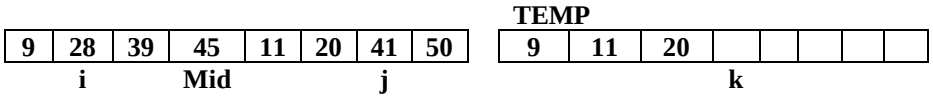
50. We start with i=0 and j=Mid+1 which indicates first elements of both sorted sub-arrays. We will compare i^{th} element with j^{th} element. Because i^{th} element is 9 is smaller than j^{th} element 11, we copy smaller element 9 to the TEMP array on kth position. Variable k has been initialized with 0. Now because of we have copied i^{th} element, will increment variable i and k as shown below:

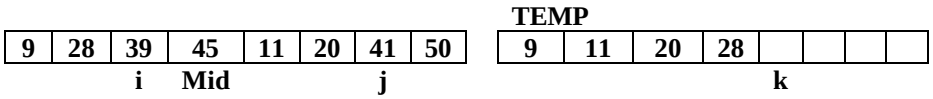


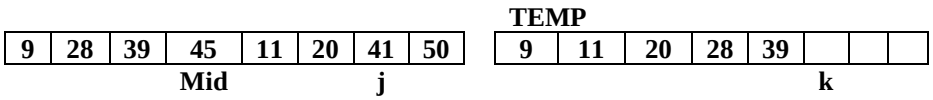
Now, again we will compare i^{th} (28) and j^{th}(11). Because j^{th} (11) element is smaller than i^{th} (28) element, we will copy j^{th} element of an array to TEMP array at k^{th} position. Because we have copied j^{th} element variable j and k will be incremented.

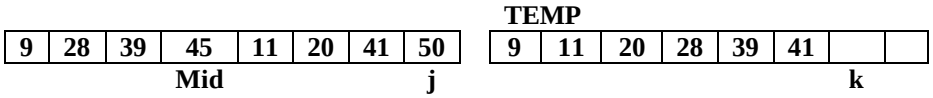


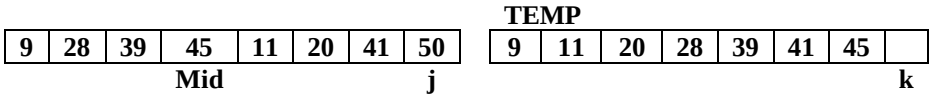
The same process is repeated again and again as shown below:



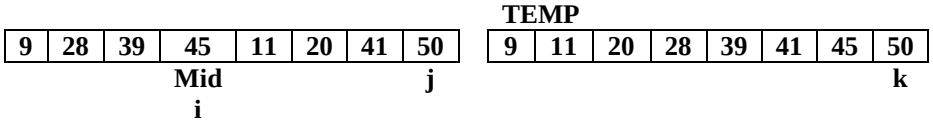








Once all the elements of any one sub-array are copied to the TEMP array, rest of the elements of another sub-array have to be copied to the array TEMP. In our example variable i reaches to Mid and that element is copied in the TEMP array, so first array is copied. Now we have to copy all the pending elements of second subarray. Second sub array have only one element left that is 50, so we will copy 50 to the TEMP array at k^{th} potion.



Once the process is completed then we will get TEMP array is merged and sorted. We will copy all element of TEMP array to the original array.

The worst case complexity of Merge sort array is O(n log n). but it needs additional space of O(n) for an array TEMP. The program of Merger sort algorithm is given below:

```
/*Merge sort */
#include<stdio.h>
//Prototype declaration
void mrg(int *,int, int, int);
void merge_sort(int *, int, int);
//Main function
void main()
{
   int x[10],i,j,k,n;
   printf("Enter Number of Elements you want to Enter:");
   scanf("%d",&n);
   for(i=0;i<n;i++)
   {
      printf("Enter Element for X[%d]:",i);
      scanf("%d",&x[i]);
   }
   printf("\nArray Before Sorting:\n");
   for(i=0;i<n;i++)
      printf("\t%d",x[i]);
   //Sorting an array
   merge_sort(x,0,n-1);
    printf("\nArray After Sorting:\n");
   for(i=0;i<n;i++)
      printf("\t%d",x[i]);
}
void merge_sort(int *x, int beg, int end)
{
   int mid;
   if(beg<end)
   {
      mid=(beg+end)/2;
      //After calculating MID split the array recursively
      merge_sort(x,beg,mid);
      merge_sort(x,mid+1,end);
      //After completing SPLIT start MERGE process
```

```
        mrg(x,beg,mid,end);
    }
}
void mrg(int *x,int beg, int mid, int end)
{
    int i=beg, j=mid+1, k=beg, temp[10];
    /* Array1 is an array from beg to mid, Array2 is an Array from mid+1 to end
    Compare Ith element of the Array1 to Jth element of Array2 and copy the
    smaller element to the temp array at Kth position. */
    while((i<=mid) && (j<=end))
    {
        if(x[i]<x[j])
        {
            temp[k]=x[i];
            i++;
        }
        else
        {
            temp[k]=x[j];
            j++;
        }
        k++;
    }
    if(i>mid)
    {
        //Copying all remaining elements of Array2 to temp array
        while(j<=end)
        {
            temp[k]=x[j];
            k++;
            j++;
        }
    }
    else
    {
        //Copying all remaining elements of Array1 to temp array
        while(i<=mid)
        {
            temp[k]=x[i];
            i++;
            k++;
        }
    }
    //Copy temp array to original array X
    for(i=beg;i<k;i++)
```

```
  {
    x[i]=temp[i];
  }
}
```

Output:

```
Enter Number of Elements you want to Enter:8
Enter Element for X[0]:45
Enter Element for X[1]:9
Enter Element for X[2]:28
Enter Element for X[3]:39
Enter Element for X[4]:11
Enter Element for X[5]:20
Enter Element for X[6]:50
Enter Element for X[7]:41

Array Before Sorting:
    45    9    28    39    11    20    50    41
Array After Sorting:
    9    11    20    28    39    41    45    50
```

Exercise:

Question: 1 Do as directed

1. What is an array? How can we declare and initialize it?
2. What is searching? List and explain all searching method in the array?
3. Write an algorithm/program for Linear search.
4. Write an algorithm/program to implement Binary search.
5. Write an explain following sorting algorithms/programs:
   5.1. Selection sort
   5.2. Bubble sort
   5.3. Insertion sort
   5.4. Quick sort
   5.5. Merge sort

Question:2 Fill in the blank:

1. In _____ sorting algorithm two nearby values are compared.
2. We take pivot element in _____ sorting algorithm.
3. Time complexity for Merge sort algorithm is _____.
4. Worst case complexity of Bubble sort algorithm is _____.
5. _____ sorting algorithm has average case complexity O(n log n) and wort case complexity $O(n^2)$.

6. Worst case complexity of Linear search algorithm is _____ and Best-case complexity is _____.
7. Average case complexity of Binary search algorithm is _____.

Answers:

1. Bubble sort
2. Quick sort
3. O (n log n)
4. $O(n^2)$
5. Quick sort
6. O(n) and O(1)
7. O (log n)

Chapter:3 Linked Lists

In the previous chapter we have discussed, data structure arrays. Array data structures needs contagious memory space. Means the array elements are stored in contagious memory locations. In working computer, so many programs and data we are loading in the computer's memory and after finishing we terminate programs. So, after sometimes if we observe memory then some random areas of memory have some data and instructions where some area of the memory locations are free. This problem is called fragmentation.

Consider a case in which we are running a C-Language program, and we have declared an array of size 100 (int x [100] ;). When we run the program, system has to allocate 200 Bytes contagious memory to variable 'x'. Suppose, if the system has 300 Bytes of free memory space, but not contagiously (Due to fragmentation problem) system can not run the program. Here we have sufficient free space in the main memory, but still memory can not be allocated to array 'x' as it need contagious space.

To overcome the problem of contagious memory, we store data elements in the Linked Lists. **Linked List is collection of data elements, which stores data elements non-contagiously manner in the memory.** So, to store data elements in the memory Link List do not requires contagious memory, whereas Array is collection of data elements which needs contagious memory.

Introduction to Link List

Link list is a linear data structure, which stores collection of data elements same as array. The main difference between Array and Link List is Array stores data elements in contagious memory locations, where Link List stores data elements in non-contagious manner.

Link List is a collection of nodes, where node is structure which includes data elements (user's data) and a pointer variable which stores address to the next node. We declare only one global pointer variable of type node, and it gives address of the first node. First node will give first data element and it gives address (reference) to the next (second) node. Second node give second data element and it also gives address to the next (Third) node. And in that way, we can access all nodes. The next part of the last node will be NULL. Declaration of the node is shown below:

```
struct node
{
   int data;
   struct node * next;
} *first=NULL;
```

Create Function:

When user choose this option, we will ask user that how many values user want to insert in the Link List. Create function will take that much values from the user and create nodes accordingly. Each node has data part which stores, data entered by the user

and next part which has address to the next node. The address of the first node will stored in the first pointer, and next part of the last node should have NULL value. Coding of the create function is given below:

```
void create()
{
   struct node *new1;
   int i,n;
   printf("\nEnter Number of Elements to be entered in the Link List:");
   scanf("%d", &n);
   for(i=1;i<=n;i++)
   {
      if(first==NULL)
      {
         new1=(struct node *)malloc(sizeof(struct node));
         first=new1;
      }
      else
      {
         new1->next=(struct node *)malloc(sizeof(struct node));
         new1=new1->next;
      }
      printf("Enter Value:");
      scanf("%d", &new1->data);
      new1->next=NULL;
   }
}
```

In this function, we will take how many elements user want to insert in the link list into variable 'n'. We run loop of variable 'i' from 1 to variable 'n'. We check the first variable, if it is NULL, this is the first node. We create the first node using new1 variable and malloc() function and copy the address of new1 to global variable first. If first node is already created then we create the address of the new node to the next part of the node which is pointed by new1. We will copy the address of newly created node to new1, so new1 pointer is pointing to the newly created node. We take the value from the user and put it into the data part of the new1. We also set next part to NULL, as the node pointed by new1 is always last node.

Display Function:

When user select this option then we have to display all the values, exists in the link list. First, we check the global variable first. If first variable is NULL, then we will print link list is not yet created or it is empty. If it is, we will return from the function. We take pointer current (i.e. curr) and copy the address of first node into variable curr. We will move curr till end of the link list (curr != NULL). On each node we print the data part and move our curr pointer to the next node (curr=curr->next). The complete display function is given below:

```
void display()
{
   struct node *curr;
   if(first==NULL)
   {
      printf("\nLink List is Empty:");
      return;
   }
   curr=first;
   while(curr->next!=NULL)
   {
      printf("%d-->",curr->data);
      curr=curr->next;
   }
   printf("%d",curr->data);
}
```

The complete program with create() and display() function is given below:

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
   int data;
   struct node *next;
}*first=NULL;
void create()
{
   struct node *new1;
   int i,n;
   printf("\nEnter Number of Elements to be entered in the Link List:");
   scanf("%d", &n);
   for(i=1;i<=n;i++)
   {
      if(first==NULL)
      {
         new1=(struct node *)malloc(sizeof(struct node));
         first=new1;
      }
      else
      {
         new1->next=(struct node *)malloc(sizeof(struct node));
         new1=new1->next;
      }
      printf("Enter Value:");
```

```
      scanf("%d", &new1->data);
      new1->next=NULL;
   }
}
void display()
{
   struct node *curr;
   if(first==NULL)
   {
      printf("\nLink List is Empty:");
      return;
   }
   curr=first;
   while(curr->next!=NULL)
   {
      printf("%d-->",curr->data);
      curr=curr->next;
   }
   printf("%d",curr->data);
}
void main()
{
   create();
   printf("\n");
   display();
}
```

Output:

```
Enter Number of Elements to be entered in the Link List:5
Enter Value:5
Enter Value:10
Enter Value:15
Enter Value:20
Enter Value:25

5-->10-->15-->20-->25
```

[Note: I am using code block editor to write the C-Program. In the code block application, all dynamic memory functions (such as malloc(), calloc(), and free() are available in the stdlib.h header file. If you are Turbo or Borland C user then you include alloc.h header file instead of stdlib.h]

The complete program of the link list is given below. Please refer the comments written in the program which will explain the complete logic of the program.

```
#include<stdio.h>
#include<stdlib.h>
/* Defining structure node and first pointer variable which will be initialized with NULL value */
struct node
{
   int data;
   struct node *next;
}*first=NULL,*second=NULL;

/* General Insert function can be used without using create() function. It inserts the value at the end of the link list */

void insert(int val)
{
   struct node *new1,*curr;
   //Creating new node and setting its data and next part
   new1=(struct node *)malloc(sizeof(struct node));
   new1->data=val;
   new1->next=NULL;
   //If link list is not created then creating first node
   if(first==NULL)
   {
      first=new1;
      return;
   }
   // If link list is created then finding the last node
   curr=first;
   // Last node is that in which we will get NULL in its next part
   while(curr->next!=NULL)
   {
      curr=curr->next;
   }
   //Copying the address of the new node to the next part of the last node
   curr->next=new1;
}
/* Displaying Link List */
void display(struct node *f)
{
   struct node *curr;
   if(f==NULL)
   {
      printf("linked list is empty:");
      return;
   }
```

```
   curr=f;
   while(curr!=NULL)
   {
      printf("\n(%d)",curr->data);
      curr=curr->next;
   }

}
/*Deleting First Node*/
void delbeg(int val)
{
   struct node *curr;
   if(first==NULL)
   {
      printf("\n Link List Is Empty:");
      return;
   }
   curr=first;

   if(first->data==val)
   {
      first=first->next;
      free(curr);
      return;
   }
}

/*Deleting Last Node*/
void delend(int val)
{
   struct node *curr,*prev;
   if(first==NULL)
   {
      printf("\n Link List Is Empty:");
      return;
   }

   curr=first;

   while(  curr!=NULL)
   {
      prev=curr;
      curr=curr->next;

       if(curr==NULL)
```

```
    {
        printf("\n value does not exist");
        return;
    }
    }

    prev->next=curr->next;
    free (curr);
}
//Deleting any value from the Link List
void delnode(int val)
{
    struct node *curr,*prev;
    //If Link list is Empty
    if(first==NULL)
    {
        printf("\n linked list is empty");
        return;
    }
    //Place curr pointer to the fist node
    curr=first;
    //If value found om the first node
    if(first->data==val)
    {
        first=first->next;
        free (curr);
        return;
    }
    //Search the value in the Link List
    while(curr->data!=val && curr!=NULL)
    {
        //Previous pointer is following current
        prev=curr;
        curr=curr->next;
    }
    //If value not found
    if(curr==NULL)
    {
        printf("\n value does not exist");
        return;
    }
    //Copying the next node address to the next part of previous node
    prev->next=curr->next;
    //Deleting node
    free (curr);
```

```
}

/*Inserting value as First node*/
void insbeg(int val)
{
   struct node *new1;
   new1=(struct node *)malloc(sizeof(struct node));
   new1->data=val;
   new1->next=NULL;

   if(first==NULL)
   {
      printf("\n linklist is empty");
      return;
   }
   new1->next=first;
   first=new1;

   }
/*Inserting value to the End of the link list*/

void insend(int val)
{
   struct node *new1,*curr;
   new1=(struct node *)malloc(sizeof(struct node));
   new1->data=val;
   new1->next=NULL;

   curr=first;
   while(curr->next!=NULL)
   {
      curr=curr->next;
   }
   curr->next=new1;

}
//Inserting value before some key value
void insbfr(int val,int key)
{
   struct node *new1,*curr,*prev;
   new1=(struct node *)malloc(sizeof(struct node));
   new1->data=val;
   new1->next=NULL;
   if(first==NULL)
   {
```

```
        printf("\n linklist is empty");
        return;
    }
    curr=first;
    if(curr->data==key)
    {
        first=new1;
        new1->next=curr;
        return;
    }
    while(curr->data!=key && curr!=NULL)
    {
        prev=curr;
        curr=curr->next;
    }
    if(curr==NULL)
    {
        printf("\nkey does not exist");
        return;
    }
    new1->next=prev->next;
    prev->next=new1;
}
//Inserting value after, some key value
void insaft(int val,int key)
{
    struct node *new1,*curr;
    new1=(struct node *)malloc(sizeof(struct node));
    new1->data=val;
    new1->next=NULL;
    if(first==NULL)
    {
        printf("\n linklist is empty");
        return;
    }
    curr=first;
    while(curr->data!=key && curr!=NULL)
    {
        curr=curr->next;

        if(curr==NULL)
        {
            printf("\nkey does not exist");
            return;
        }
```

```
    }
    new1->next=curr->next;
    curr->next=new1;
}
//Split link list into two separate link lists
void split(int val)
{
    struct node *curr,*prev;
    if(first==NULL)
    {
        printf("\n Link list is empty:");
        return;
    }
    curr=first;
    while(curr->data!=val && curr!=NULL)
    {
        prev=curr;
        curr=curr->next;

        if(curr==NULL)
        {
            printf("\nkey does not exist");
            return;
        }
    }
    prev->next=NULL;
    second=curr;
}
//Joining to separate link lists into one link list
void merge()
{
    struct node *curr;
    if(first==NULL || second==NULL)
    {
        printf("\n Link list is empty:");
        return;
    }
    curr=first;
    while(curr->next!=NULL)


        curr=curr->next;

    curr->next=second;
    second=NULL;
```

```
}
//Sorting the link list
void sort()
{
   struct node *i,*j;
   int temp;
   i=first;
   if(i==NULL)
   {
      printf("\n linked list is empty");
      return;
   }
   while(i->next!=NULL)
   {
      j=i->next;
      while(j!=NULL)
      {
         if(i->data > j->data)
         {
               temp=i->data;
               i->data=j->data;
               j->data=temp;
         }
         j=j->next;
      }
      i=i->next;
   }
}
//Reversing the link list, using recursion
struct node *reverse(struct node *t)
{
       struct node *yes;
       if(t->next == NULL)
       {
          first = t;
          return t;
       }
       yes = reverse(t->next);
       yes -> next = t;
       return t;
}
//Main function of the program
void main()
{
   int val,ch,key,n;
```

```
struct node *no;
do{
   printf("\n----------MENU----------");
   printf("\n----------1.INSERT----------");
   printf("\n----------2.DISPLAY----------");
   printf("\n----------3.DELETE AT FIRST----");
   printf("\n----------4.DELETE AT LAST----");
   printf("\n----------5.DELETE----------");
   printf("\n----------6.INSERT AT FIRST----");
   printf("\n----------7.INSERT BEFORE----");
   printf("\n----------8.INSERT AFTER-----");
   printf("\n----------9.INSERT AT LAST----");
   printf("\n----------10.SORT----------");
   printf("\n----------11.REVERSE----------");
   printf("\n----------12.SPLIT----------");
   printf("\n----------13.MERGE----------");
   printf("\n----------14.EXIT----------");

   printf("\n----------ENTER YOUR CHOICE---------\n");
   scanf("%d",&ch);

   if(ch==1)
   {
      printf("\n Enter value:");
      scanf("%d",&val);
      insert(val);
   }
   else if(ch==2)
   {
      printf("\n Enter your coice");
      scanf("%d",&n);
      if(n==1)
      {
         display(first);
      }
      else
      {
         display(second);
      }

   }
   else if(ch==3)
   {
      printf("\n Enter value which you want to delete:");
      scanf("%d",&val);
```

```
    delbeg(val);
}
else if(ch==4)
{
    printf("\n Enter value which you want to delete:");
    scanf("%d",&val);
    delend(val);
}

else if(ch==5)
{
    printf("\n Enter value which you want to delete:");
    scanf("%d",&val);
    delnode(val);
}
else if(ch==6)
{
    printf("\n Enter value:");
    scanf("%d",&val);
    insbeg(val);
}
else if(ch==7)
{

    printf("\n Enter value:");
    scanf("%d",&val);
    insend(val);
}
else  if(ch==8)
{
    printf("\n Enter value:");
    scanf("%d",&val);
    printf("\n Enter key:");
    scanf("%d",&key);
    insaft(val,key);
}
else if(ch==9)
{
    printf("\n Enter value:");
    scanf("%d",&val);
    insert(val);
}
else if(ch==10)
{
    sort();
```

```
    }
    else if(ch==11)
    {
       no = reverse(first);
       no -> next = NULL;
    }
    else if(ch==12)
    {
       printf("\n Enter value:");
       scanf("%d",&val);
       split(val);
    }
    else if(ch==13)
    {
       merge();
    }
    else if(ch==14)
    {
       printf("\n GOOD BYE");
    }
    else
    {
       printf("\n INVALID INPUT");
    }
  }while(ch!=14);
}
```

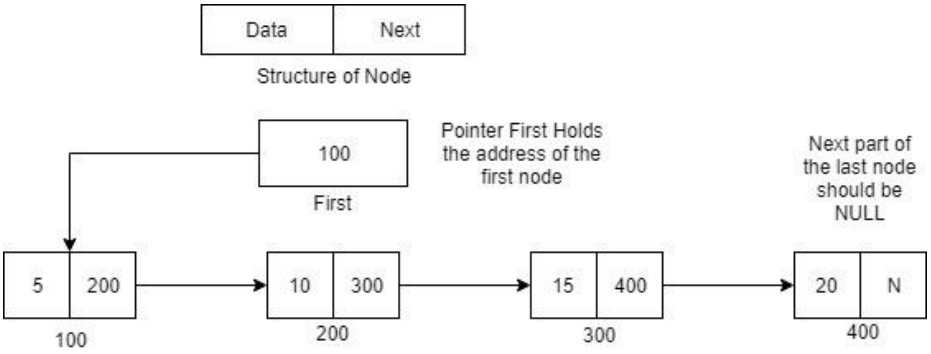
Types of Link Lists:

There are three types of link lists are available, which are discussed below:

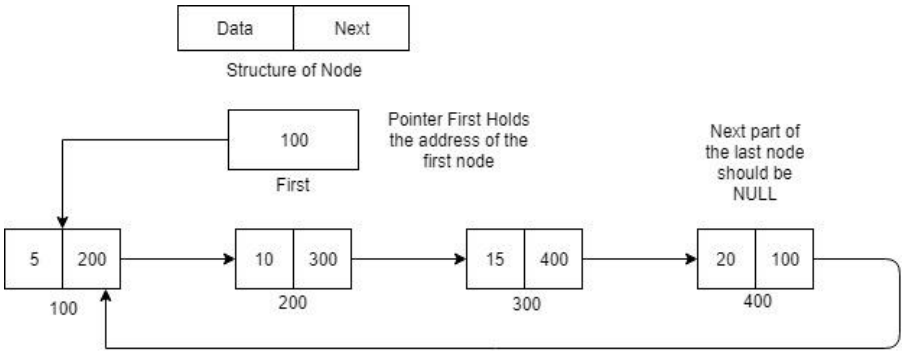
1. Singly Link List: The figure give below is of singly link list. In the singly link list each node has two sections. [1] Data: which is used to store the data value for the node. Usually data value will be given by the user, and [2] Next: This stores the address of the next node. The 'Next' part of the last node will NULL. In this type of link list every node has the address of the next node. So, if we reach to the first node, it gives address to second node, second node gives address to the third and so on. It is important to store the address of the first node. So that we are declaring one global variable 'first' which will give address of the first node. In the case of empty link list (no node is there in the link list) first pointer will have NULL value.

   In the figure give below, 100, 200, 300, 400 etc. are the address of the nodes. These are the assumed address to explain the example. In reality, when malloc() function is called, operating system will allot the memory space to node

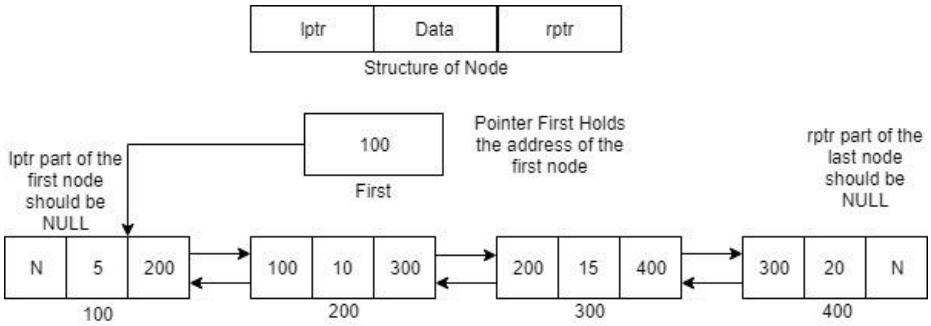
from the available free space. That can vary machine to machine and time to time.



2. Circular Link List: It is same as a singly link list except the next part of the last node is storing the address of the first node. So that, we can move to the first node from the last node. The figure given below, represents circular link list.



3. Doubly Linked List: In the link lists discussed above use the common data structure of the node, having two parts data and next. Next part of the node stores the address of the next node. Nodes in these link lists do not preserves the address of the previous node. In the doubly link list node encapsulate three parts: [1] **lptr**: which stores address of the previous node, [2] **data**: which stores data element entered by the user and [3] **rptr**: which stores the address of the next node. That means, doubly link list preserves two (double) addresses, one for the next node and second for the previous node. Singly link list allows traversal in only one direction (forward only), whereas doubly linked list allows traversal in both the directions (reverse and forward both). The figure given below represents doubly linked list.



In the doubly linked list, we node to modify structure node. Instead of having 2 elements in the singly linked list (data and next pointer), now we will place 3 elements in the node, that is (lptr which stores the address of the left node, data and rptr which stores the address of the right node). So, the structure node can be defined as,

```
struct node
{
   struct node *lptr, *rptr;
   int data;
} * first =NULL;
```

Program to implement doubly linked list is given below: