DEPARTMENT OF COMPUTER ENGINEERING

4TH SEMESTER

SUBJECT: DATA STRUCTURE USING C

SUBJECT CODE:180842

SUBJECT: DATA STRUCTURE USING C
SEM: 4th (Computer Engineering)

UNIT-1

Fundamental Notations

1.1 Problem solving Concepts: Top Down and Bottom up Design

Programming refers to the method of creating a sequence of instructions to enable the computer to perform a task. It is done by developing logic and then writing instructions in a programming language. A program can be written using various programming practices available. A **programming practice** refers to the way of writing a program and is used along with coding style guidelines. Some of the commonly used programming practices include top-down programming, bottom-up programming and structured programming,

1.1.1 Top Down Programming

Top-down programming focuses on the use of modules. It is therefore also known as modular programming. The program is broken up into small modules so that it is easy to trace a particular segment of code in the software program. The modules at the top level are those that perform general tasks and proceed to other modules to perform a particular task. Each module is based on the functionality of its functions and procedures. In this approach, programming begins from the top level of hierarchy and progresses towards the lower levels. The implementation of modules starts with the main module. After the implementation of the main module, the subordinate modules are implemented and the process follows in this way. In top-down programming, there is a risk of implementing data structures as the modules are dependent on each other and they nave to share one or more functions and procedures. In this way, the functions and procedures are globally visible. In addition to modules, the top-down programming uses sequences and the nested levels of commands.

1.1.2 Bottom-up Programming

Bottom-up programming refers to the style of programming where an application is constructed with the description of modules. The description begins at the bottom of the hierarchy of modules and progresses through higher levels until it reaches the top. Bottom-up programming is just the opposite of top-down programming. Here, the program modules are more general and reusable than top-down programming.

It is easier to construct functions in bottom-up manner. This is because bottom-up programming requires a way of passing complicated arguments between functions.

1.1.3 Structured Programming

Structured programming is concerned with the structures used in a computer program. Generally, structures of computer program comprise decisions, sequences, and loops. The **decision structures** are used for conditional execution of statements (for example, 'if statement). The **sequence structures** are used for the sequentially executed statements. The **loop structures** are used for performing some repetitive tasks in the program.

Structured programming forces a logical structure in the program to be written in an efficient and understandable manner. The purpose of structured programming is to make the software code easy to modify when required. Structured programming focuses on reducing the following statements from the program.

1. 'GOTO' statements.
2. 'Break' or 'Continue' outside the loops.
3. Multiple exit points to a function, procedure, or subroutine. For example, multiple 'Return' statements should not be used.
4. Multiple entry points to a function, procedure, or a subroutine.

Structured programming generally makes use of top-down design because program structure is divided into separate subsections. A defined function or set of similar functions is kept separately. Due to this separation of functions, they are easily loaded in the memory. In addition, these functions can be reused in one or more programs. Each module is tested individually. After testing, they are integrated with other modules to achieve an overall program structure. Note that a key characteristic of a structured statement is the presence of single entry and single exit point. This characteristic implies that during execution, a structured statement starts from one defined point and terminates at another defined point.

1.2 Data Type, Variable and Constants

1.2.1 Data and Data Item

Data are simply collection of facts and figures. Data are values or set of values. A data item refers to a single unit of values. Data items that are divided into sub items are group items;those that are not are called elementary items. For example, a student's name may be divided into three sub items – [first name, middle name and last name] but the ID of a student would normally be treated as a single item. In the above example (ID, Age, Gender, First, Middle, Last, Street, Area) are elementary data items, whereas (Name, Address) are group data items.

1.2.2 Data Type

Data type is a classification identifying one of various types of data, such as floating-point, integer, or Boolean, that determines the possible values for that type; the operations that can be done on values of that type; and the way values of that type can be stored. It is of two types: Primitive and non-primitive data type. Primitive data type is the basic data type that is provided by the programming language with built-in support. This data type is native to the language and is supported by machine directly while non-primitive data type is derived from primitive data type. For example- array, structure etc.

1.2.3 Variable

It is a symbolic name given to some known or unknown quantity or information, for the purposeof allowing the name to be used independently of the information it represents. A variable name in computer source code is usually associated with a data storage location and thus also its contents and these may change during the course of program execution.

1.2.4 Record

Collection of related data items is known as record. The elements of records are usually called fields or members. Records are distinguished from arrays by the fact that their number of fields is typically fixed, each field has a name, and that each field may have a different type.

1.2.5 Constant

A constant is a value or an identifier whose value cannot be altered in a program. For example: 1, 2.5, "C programming is easy", etc.
As mentioned, an identifier also can be defined as a constant.

```
const double PI = 3.14
```

Here, PI is a constant. Basically what it means is that, PI and 3.14 is same for this program.

Below are the different types of constants you can use in C.

1. Integer constants

An integer constant is a numeric constant (associated with number) without any fractional or exponential part. There are three types of integer constants in C programming:

- decimal constant(base 10)
- octal constant(base 8)
- hexadecimal constant(base 16)

For example:

```
Decimal constants: 0, -9, 22 etc

Octal constants: 021, 077, 033 etc

Hexadecimal constants: 0x7f, 0x2a, 0x521 etc
```

In C programming, octal constant starts with a 0 and hexadecimal constant starts with a 0x.

2. Floating-point constants

A floating point constant is a numeric constant that has either a fractional form or an exponent form. For example:

```
-2.0

0.0000234

-0.22E-5
```

Note: E-5 = 10^{-5}

3. Character constants

A character constant is a constant which uses single quotation around characters. For example: 'a', 'l', 'm', 'F'

4. Escape Sequences

Sometimes, it is necessary to use characters which cannot be typed or has special meaning in C programming. For example: newline (enter), tab, question mark etc. In order to use these characters, escape sequence is used.

1.3 Pointers

A **pointer** is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before using it to store any variable address. The general form of a pointer variable declaration is −

```
type *var-name;
```

Here, **type** is the pointer's base type; it must be a valid C data type and **var-name** is the name of the pointer variable. The asterisk * used to declare a pointer is the same asterisk used for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Take a look at some of the valid pointer declarations −

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch     /* pointer to a character */
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

There are a few important operations, like **(a)** define a pointer variable, **(b)** assign the address of a variable to a pointer and **(c)** finally access the value at the address available in the pointer variable. This is done by using unary operator ***** that returns the value of the variable located at the address specified by its operand. The following example makes use of these operations −

```
#include<stdio.h>

int main (){

intvar=20;/* actual variable declaration */

int*ip;/* pointer variable declaration */

ip=&var;/* store address of var in pointer variable*/

printf("Address of var variable: %x\n",&var);

/* address stored in pointer variable */

printf("Address stored in ip variable: %x\n",ip);

/* access the value using the pointer */

printf("Value of *ip variable: %d\n",*ip);

return0;

}
```

When the above code is compiled and executed, it produces the following result −

```
Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c
Value of *ip variable: 20
```

1.4 DATA STRUCTURE

In computer science, a data structure is a particular way of storing and organizing data in acomputer's memory so that it can be used efficiently. Data may be organized in many differentways; the logical or mathematical model of a particular organization of data is called a datastructure. The choice of a particular data model depends on the two considerations first; it mustbe rich enough in structure to mirror the actual relationships of the data in the real world. On theother hand, the structure should be simple enough that one can effectively process the datawhenever necessary.

1.4.1 Need of data structure

- It gives different level of organization data.
- It tells how data can be stored and accessed in its elementary level.
- Provide operation on group of data, such as adding an item, looking up highest priorityitem.
- Provide a means to manage huge amount of data efficiently.
- Provide fast searching and sorting of data.

1.4.2 Selecting a data structure

Selection of suitable data structure involve following steps –

- Analyze the problem to determine the resource constraints a solution must meet.
- Determine basic operation that must be supported. Quantify resource constraint for eachoperation
- Select the data structure that best meets these requirements.

Each data structure has cost and benefits. Rarely is one data structure better than other inall situations. A data structure require :

- Space for each item it stores
- Time to perform each basic operation
- Programming effort.

Each problem has constraints on available time and space. Best data structure for the taskrequires careful analysis of problem characteristics.

1.4.3 Type of data structure

1.4.3.1 Static data structure

A data structure whose organizational characteristics are invariant throughout its lifetime. Suchstructures are well supported by high-level languages and familiar examples are arrays andrecords. The prime features of static structures are

(a) None of the structural information need be stored explicitly within the elements – it is oftenheld in a distinct logical/physical header;

(b) The elements of an allocated structure are physically contiguous, held in a single segment ofmemory;

(c) All descriptive information, other than the physical location of the allocated structure, isdetermined by the structure definition;
(d) Relationships between elements do not change during the lifetime of the structure.

1.4.3.2 Dynamic data structure

A data structure whose organizational characteristics may change during its lifetime. Theadaptability afforded by such structures, e.g. linked lists, is often at the expense of decreasedefficiency in accessing elements of the structure. Two main features distinguish dynamicstructures from static data structures. Firstly, it is no longer possible to infer all structural information from a header; each data element will have to contain information relating itlogically to other elements of the structure. Secondly, using a single block of contiguous storageis often not appropriate, and hence it is necessary to provide some storage management schemeat run-time.

1.4.3.3 Linear Data Structure

A data structure is said to be linear if its elements form any sequence. There are basically twoways of representing such linear structure in memory.

a) One way is to have the linear relationships between the elements represented by means ofsequential memory location. These linear structures are called arrays**.**

b) The other way is to have the linear relationship between the elements represented by means ofpointers or links. These linear structures are called linked lists**.**

The common examples of linear data structure are arrays, queues, stacks and linked lists.

1.4.3.4 Non-linear Data Structure

This structure is mainly used to represent data containing a hierarchical relationship betweenelements. E.g. graphs, family trees and table of contents.

1.5 BRIEF DESCRIPTION OF DATA STRUCTURES

1.5.1 Array

The simplest type of data structure is a linear (or one dimensional) array. A list of a finitenumber *n* of similar data referenced respectively by a set of *n* consecutive numbers, usually 1, 2,3 . . . . *n*. if we choose the name **A** for the array, then the elements of **A** are denoted bysubscript notation**A** 1, **A** 2, **A** 3 . . . . **A** nor by the parenthesis notation

A (1), A (2), A (3) . . . . . . A (n)

or by the bracket notation

A [1], A [2], A [3] . . . . . . A [n]

Example:

A linear array **A[8]** consisting of numbers is pictured in following figure.

1.5.2 Linked List

A linked list or one way list is a linear collection of data elements, called nodes, where the linearorder is given by means of pointers. Each node is divided into two parts:

The first part contains the information of the element/node

The second part contains the address of the next node (link /next pointer field) in the list.

There is a special pointer Start/List contains the address of first node in the list. If this specialpointer contains null, means that List is empty.

Example:

1.5.3 Tree

Data frequently contain a hierarchical relationship between various elements. The data structurewhich reflects this relationship is called a rooted tree graph or, simply, a tree.

1.5.4 Graph

Data sometimes contains a relationship between pairs of elements which is not necessarilyhierarchical in nature, e.g. an airline flights only between the cities connected by lines. This datastructure is called Graph.

1.5.5 Queue

A queue, also called FIFO system, is a linear list in which deletions can take place only at oneend of the list, the Font of the list and insertion can take place only at the other end Rear.

1.5.6 Stack

It is an ordered group of homogeneous items of elements. Elements are added to and removedfrom the top of the stack (the most recently added items are at the top of the stack). The lastelement to be added is the first to be removed (LIFO: Last In, First Out).

1.6 DATA STRUCTURES OPERATIONS

The data appearing in our data structures are processed by means of certain operations. In fact,the particular data structure that one chooses for a given situation depends largely in thefrequency with which specific operations are performed.

The following four operations play a major role in this text:

Traversing: accessing each record/node exactly once so that certain items in the recordmay be processed. (This accessing and processing is sometimes called "visiting" therecord.)

Searching: Finding the location of the desired node with a given key value, or findingthe locations of all such nodes which satisfy one or more conditions.

Inserting: Adding a new node/record to the structure.

Deleting: Removing a node/record from the structure.

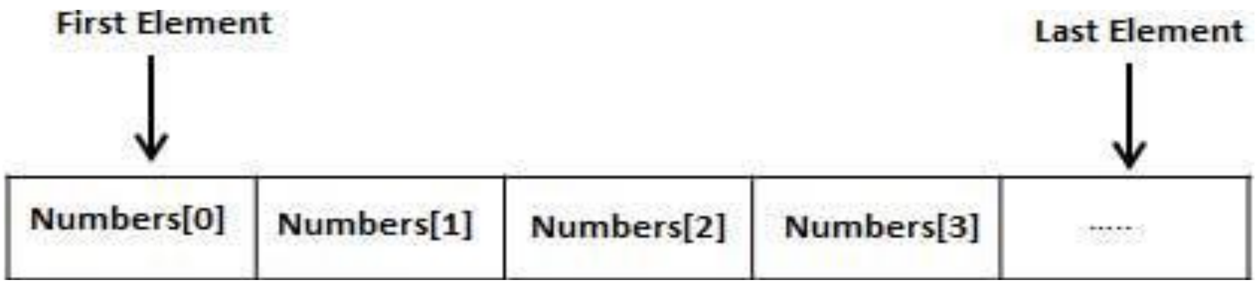
Sorting: Arranging elememts in either ascending or descending order.

UNIT-2

Arrays

2.1 Concept of Arrays

Array is a data structure which can hold a fix number of items and these items should be of the same type. Most of the data structures make use of arrays to implement their algorithms. Instead of declaring individual variables, such as number0, number1, ..., and number99, we declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.
All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



The array may be categorized as :

- One dimensional array
- Two dimensional array
- Multidimensional array

2.1.1 One Dimensional Array

A one-dimensional array (or single dimension array) is a type of linear array. Accessing its elements involves a single subscript which can either represent a row or column index.

As an example consider the C declaration

`intanArrayName[10];`

Syntax :datatypeArrayname[sizeofArray];

In the given example the array can contain 10 elements of any value available to the `int` type. In C, the array element indices are 0-9 in this case. For example, the expressions `anArrayName[0]` and `anArrayName[9]` are the first and last elements respectively.

One Dimensional Arrays can be initialized as follows:

Examples –

```
// A character array in C
char arr1[] = {'g', 'e', 'e', 'k', 's'};


// An Integer array in C
int arr2[] = {10, 20, 30, 40, 50};


// Item at i'th index in array is typically accessed
// as "arr[i]".  For example arr1[0] gives us 'g'
// and arr2[3] gives us 40.
```

As we know now 1-d array are linear array in which elements are stored in the successive memory locations. The element at which the first element is stored in memory is called its **base address**. Now consider the following example :

int arr[5];

Element	34	78	98	45	56
Memory Address	arr[0] = 100	arr[1] = ?	arr[2] = ?	arr[3] = ?	arr[4] = ?

Here we have defined an array of five elements of integer type whose first element is at base address 100. i.e, the element arr[0] is stored at base address 100. Now for calculating the starting address of the next element i.e. of a[1], we can use the following formula :

Base Address (B)+ No. of bytes occupied by element (C) * index of the element (i)

/* Here C is constant integer and vary according to the data type of the array, for e.g. for integer the value of C will be 2 bytes, since an integer occupies 2 bytes of memory. */

Now, we can calculate the starting address of second element of the array as :

arr[1] = 100 + 2 * 1 = 102/*Thus starting address of second element of array is 102 */

Similarly other addresses can be calculated in the same manner as :

arr[2] = 100 + 2 * 2 = 104

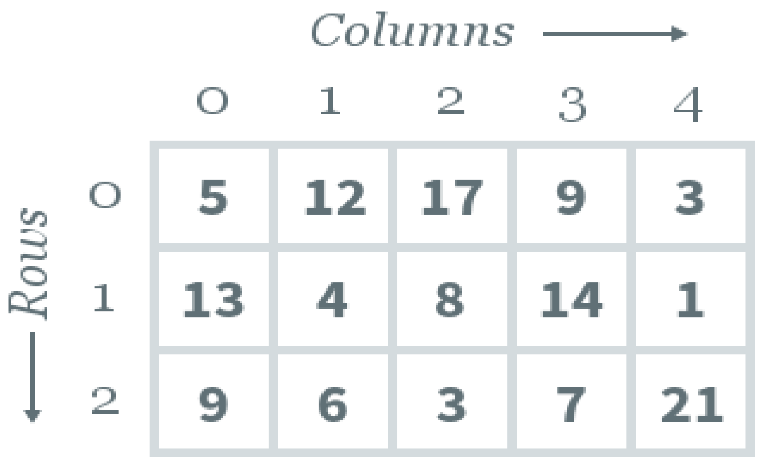
arr[3] = 100 + 2 * 3 = 106

arr[4] = 100 + 2 * 4 = 108

2.1.2 Two Dimensional Array

An array of one dimensional arrays is known as 2-D array. The two dimensional (2D) array in C programming is also known as matrix. A matrix can be represented as a table of rows and columns.

Consider following 2-D array, which is of the size 3×5. For an array of size N×M, the rows and columns are numbered from 0 to N−1 and columns are numbered from 0 to M−1, respectively. Any element of the array can be accessed by arr[i][j] where 0≤i<N and 0≤j<M. For example, in the following array, the value stored at arr[1][3] is 14.



2D Array of size 3 x 5

2D array declaration:

To declare a 2D array, you must specify the following:
Row-size: Defines the number of rows
Column-size: Defines the number of columns
Type of array: Defines the type of elements to be stored in the array i.e. either a number, character, or other such datatype. A sample form of declaration is as follows:

```
typearr[row_size][column_size]
```

A sample C array is declared as follows:

```
intarr[3][5];
```

2D array initialization:

An array can either be initialized during or after declaration. The format of initializing an array during declaration is as follows:

```
typearr[row_size][column_size]={{elements},{elements}...}
```

An example is given below:

```
intarr[3][5]={{5,12,17,9,3},{13,4,8,14,1},{9,6,3,7,21}};
```

Initializing an array after declaration can be done by assigning values to each cell of 2D array, as follows.

```
typearr[row_size][column_size]
arr[i][j]=14
```

An example of initializing an array after declaration by assigning values to each cell of a 2D array is as follows:

```
intarr[3][5];
arr[0][0]=5;
arr[1][3]=14;
```

Processing 2D arrays:

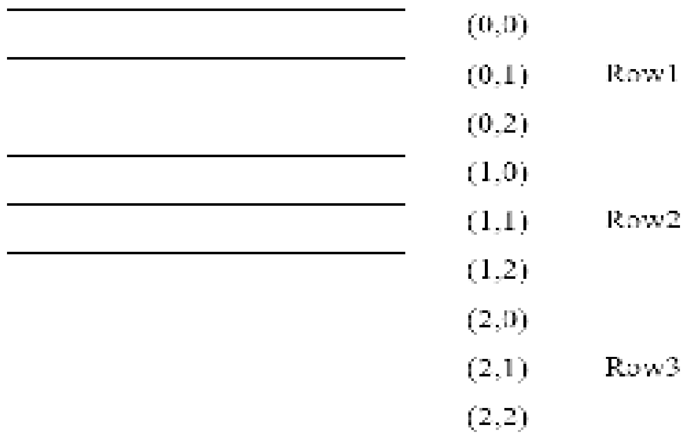
The most basic form of processing is to loop over the array and print all its elements, which can be done as follows:

```
typearr[row_size][column_size]={{elements},{elements}...}
for i from0 to row_size
for j from0 to column_size
printarr[i][j]
```

Representation of two dimensional arrays in memory

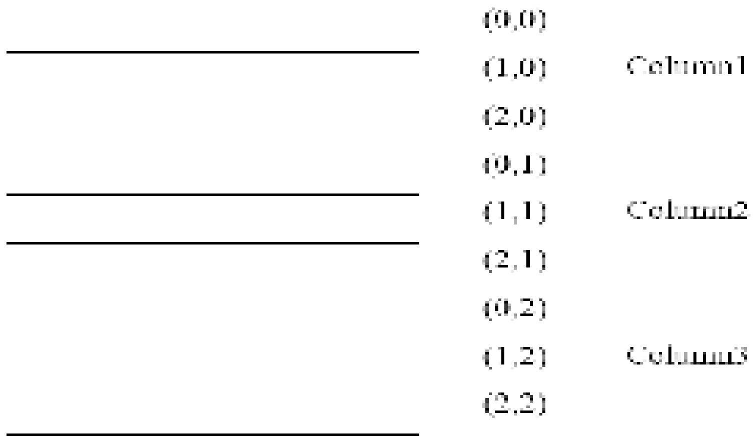
A two dimensional 'm x n' Array A is the collection of m X n elements. Programming language stores the two dimensional array in one dimensional memory in either of two ways-

Row Major Order: First row of the array occupies the first set of memory locations reserved for the array; Second row occupies the next set, and so forth.



To determine element address A[i,j]:
Location (A[i,j]) =Base Address + (N x (I - 1)) + (j - 1)
For example:
Given an array [1…5,1…7] of integers. Calculate address of element T[4,6], where BA=900.
Sol) I = 4 , J = 6
M= 5 , N= 7
Location (T [4,6]) = BA + (7 x (4-1)) + (6-1)
= 900+ (7 x 3) +5
= 900+ 21+5
= 926

Column Major Order: Order elements of first column stored linearly and then comes elements of next column

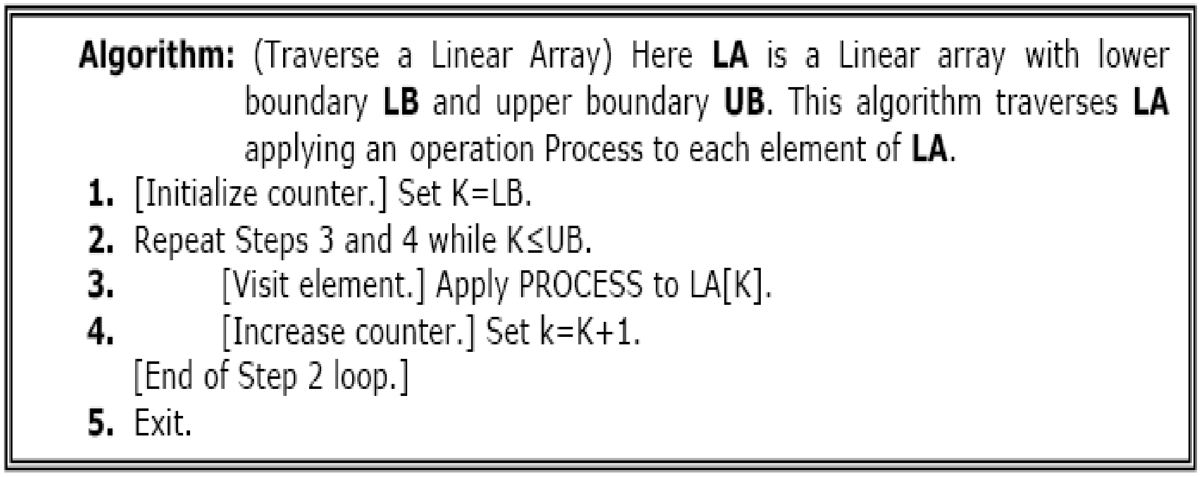


To determine element address A[i,j]:
Location (A[i,j]) =Base Address + (M x (j - 1)) + (i - 1)
For example:
Given an array [1…6,1…8] of integers. Calculate address element T[5,7], where BA=300
Sol) I = 5 , J = 7
M= 6 , N= 8
Location (T [4,6]) = BA + (6 x (7-1)) + (5-1)
= 300+ (6 x 6) +4

= 300+ 36+4

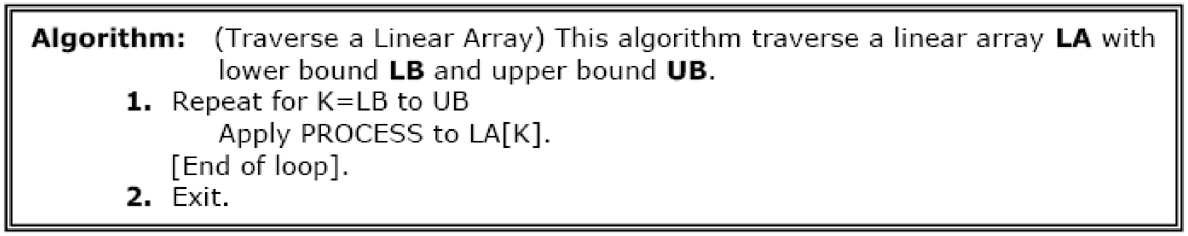
= 340

2.2 Operations on array
a) Traversing: means to visit all the elements of the array in an operation is called traversing.
b) Insertion: means to put values into an array
c) Deletion / Remove: to delete a value from an array.
d) Sorting: Re-arrangement of values in an array in a specific order (Ascending or Descending) is called sorting.
e) Searching: The process of finding the location of a particular element in an array is called searching.
a) Traversing in Linear Array:
It means processing or visiting each element in the array exactly once; Let **‘A’** is an array stored in the computer’s memory. If we want to display the contents of ***‘A’***, it has to be traversed i.e. by accessing and processing each element of ***‘A’*** exactly once.



```
Algorithm: (Traverse a Linear Array) Here LA is a Linear array with lower
           boundary LB and upper boundary UB. This algorithm traverses LA
           applying an operation Process to each element of LA.
1. [Initialize counter.] Set K=LB.
2. Repeat Steps 3 and 4 while K≤UB.
3.         [Visit element.] Apply PROCESS to LA[K].
4.         [Increase counter.] Set k=K+1.
   [End of Step 2 loop.]
5. Exit.
```

The alternate algorithm for traversing (using for loop) is :



```
Algorithm:   (Traverse a Linear Array) This algorithm traverse a linear array LA with
             lower bound LB and upper bound UB.
        1. Repeat for K=LB to UB
               Apply PROCESS to LA[K].
           [End of loop].
        2. Exit.
```

This program will traverse each element of the array to calculate the sum and then calculate& print the average of the following array of integers.
(4, 3, 7, -1, 7, 2, 0, 4, 2, 13)

b)Insertion:

Insert operation is to insert one or more data elements into an array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array.

Here, we see a practical implementation of insertion operation, where we add data at the end of the array −

Algorithm

Let **LA** be a Linear Array (unordered) with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm where ITEM is inserted into the Kth position of LA −

```
1. Start
2. Set J = N
3. Set N = N+1
4. Repeat steps 5 and 6 while J >= K
5. Set LA[J+1] = LA[J]
```

```
6. Set J = J-1
7. Set LA[K] = ITEM
8. Stop
```

Example

Following is the implementation of the above algorithm −

```
#include<stdio.h>


main(){

int LA[]={1,3,5,7,8};

int item =10, k =3, n =5;

int i =0, j = n;


printf("The original array elements are :\n");


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

printf("LA[%d] = %d \n", i, LA[i]);

}


   n = n +1;


while( j >= k){

LA[j+1]= LA[j];

      j = j -1;

}


   LA[k]= item;
```

```
            printf("The array elements after insertion :\n");


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

printf("LA[%d] = %d \n", i, LA[i]);

}

}
```

When we compile and execute the above program, it produces the following result −

Output

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
The array elements after insertion :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 10
LA[4] = 7
LA[5] = 8
```

c) Deletion Operation

Deletion refers to removing an existing element from the array and re-organizing all elements of an array.

Algorithm

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm to delete an element available at the Kth position of LA.

```
1. Start
2. Set J = K
3. Repeat steps 4 and 5 while J < N
4. Set LA[J] = LA[J + 1]
5. Set J = J+1
6. Set N = N-1
7. Stop
```

Example

Following is the implementation of the above algorithm −

```
#include<stdio.h>

main(){

int LA[]={1,3,5,7,8};

int k =3, n =5;

int i, j;


printf("The original array elements are :\n");


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

printf("LA[%d] = %d \n", i, LA[i]);

}


   j = k;


while( j < n){

LA[j-1]= LA[j];

      j = j +1;

}


   n = n -1;


printf("The array elements after deletion :\n");


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

```
printf("LA[%d] = %d \n", i, LA[i]);

}

}
```

When we compile and execute the above program, it produces the following result −

Output

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
The array elements after deletion :
LA[0] = 1
LA[1] = 3
LA[2] = 7
LA[3] = 8
```

d) Search Operation

You can perform a search for an array element based on its value or its index.

Algorithm

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm to find an element with a value of ITEM using sequential search.

```
1. Start
2. Set J = 0
3. Repeat steps 4 and 5 while J < N
4. IF LA[J] is equal ITEM THEN GOTO STEP 6
5. Set J = J +1
6. PRINT J, ITEM
7. Stop
```

Example

Following is the implementation of the above algorithm −

```
#include<stdio.h>

main(){

int LA[]={1,3,5,7,8};
int item =5, n =5;

int i =0, j =0;


printf("The original array elements are :\n");


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

printf("LA[%d] = %d \n", i, LA[i]);

}


while( j < n){

if( LA[j]== item ){

break;

}


    j = j +1;

}


printf("Found element %d at position %d\n", item, j+1);

}
```

When we compile and execute the above program, it produces the following result −

Output

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
Found element 5 at position 3
```

e) Update Operation

Update operation refers to updating an existing element from the array at a given index.

Algorithm

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm to update an element available at the Kth position of LA.

```
1. Start
2. Set LA[K-1] = ITEM
3. Stop
```

Example

Following is the implementation of the above algorithm −

```
#include<stdio.h>

main(){

int LA[]={1,3,5,7,8};

int k =3, n =5, item =10;

int i, j;


printf("The original array elements are :\n");


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

printf("LA[%d] = %d \n", i, LA[i]);

}


LA[k-1]= item;


printf("The array elements after updation :\n");


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

```
printf("LA[%d] = %d \n", i, LA[i]);

}

}
```

When we compile and execute the above program, it produces the following result −

Output

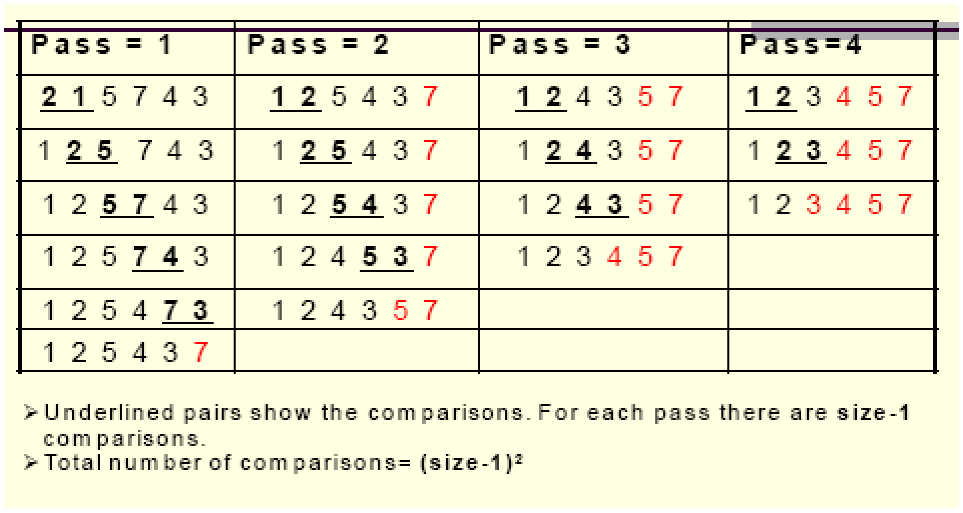
```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
The array elements after updation :
LA[0] = 1
LA[1] = 3
LA[2] = 10
LA[3] = 7
LA[4] = 8
```

f) Sorting:

Sorting an array is the ordering the array elements in ***ascending*** (increasing from min to max) or ***descending***(decreasing from max to min) order.

Bubble Sort:

The technique we use is called *"Bubble Sort"* because the bigger value gradually bubbles theirway up to the top of array like air bubble rising in water, while the small values sink to thebottom of array. This technique is to make several passes through the array. On each pass,successive pairs of elements are compared. If a pair is in increasing order (or the values areidentical), we leave the values as they are. If a pair is in decreasing order, their values areswapped in the array.



Pass = 1	Pass = 2	Pass = 3	Pass=4
2 1 5 7 4 3	**1 2** 5 4 3 7	**1 2** 4 3 5 7	**1 2** 3 4 5 7
1 **2 5** 7 4 3	1 **2 5** 4 3 7	1 **2 4** 3 5 7	1 **2 3** 4 5 7
1 2 **5 7** 4 3	1 2 **5 4** 3 7	1 2 **4 3** 5 7	1 2 3 4 5 7
1 2 5 **7 4** 3	1 2 4 **5 3** 7	1 2 3 4 5 7	
1 2 5 4 **7 3**	1 2 4 3 5 7		
1 2 5 4 3 7			

➢ Underlined pairs show the comparisons. For each pass there are size-1 comparisons.

➢ Total number of comparisons= $(\text{size-1})^2$

```
Algorithm:  (Bubble Sort) BUBBLE (DATA, N)
            Here DATA is an Array with N elements. This algorithm sorts the
            elements in DATA.
    1. for pass=1 to N-1.
    2.    for (i=0; i<= N-Pass; i++)
    3.       If DATA[i]>DATA[i+1], then:
                      Interchange DATA[i] and DATA[i+1].
                [End of If Structure.]
            [End of inner loop.]
        [End of Step 1 outer loop.]
    4. Exit.
```

2.3 Applications of array

Arrays are used to implement mathematical vectors and matrices, as well as other kinds of rectangular tables. Many databases, small and large, consist of (or include) one-dimensional arrays whose elements are records. Arrays are used to implement other data structures, such as heaps, hash tables, deques, queues, stacks, strings, and VLists. One or more large arrays are sometimes used to emulate in-program dynamic memory allocation, particularly memory pool allocation. Historically, this has sometimes been the only way to allocate "dynamic memory" portably. Arrays can be used to determine partial or complete control flow in programs, as a compact alternative to (otherwise repetitive) multiple IF statements. They are known in this context as control tables and are used in conjunction with a purpose built interpreter whose control flow is altered according to values contained in the array.

2.4 Sparse matrix

Matrix with maximum zero entries is termed as sparse matrix. It can be represented as:

- ➢ Lower triangular matrix: It has non-zero entries on or below diagonal.
- ➢ Upper Triangular matrix: It has non-zero entries on or above diagonal.
- ➢ Tri-diagonal matrix: It has non-zero entries on diagonal and at the places immediately above or below diagonal.

When a sparse matrix is represented with 2-dimensional array, we waste lot of space to represent that matrix. For example, consider a matrix of size 100 X 100 containing only 10 non-zero elements. In this matrix, only 10 spaces are filled with non-zero values and remaining spaces of matrix are filled with zero. That means, totally we allocate 100 X 100 X 2 = 20000 bytes of space to store this integer matrix. And to access these 10 non-zero elements we have to make scanning for 10000 times.

Sparse Matrix Representation

A sparse matrix can be represented by using TWO representations, those are as follows...

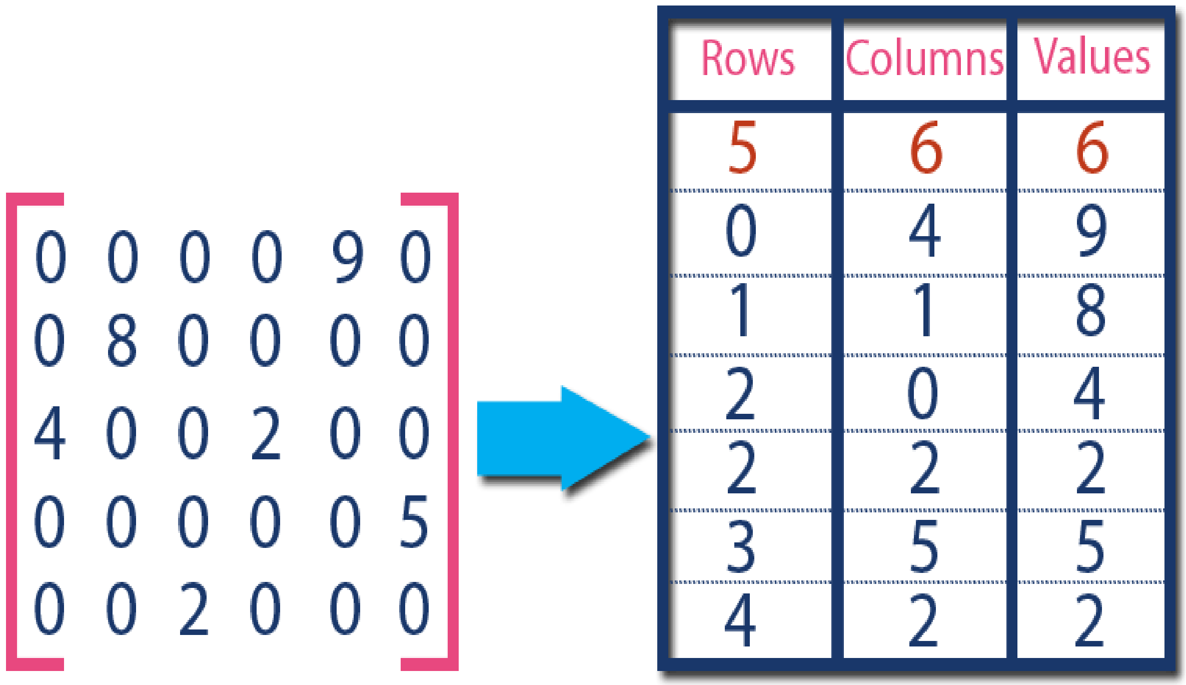
Triplet Representation

Linked Representation

Triplet Representation

In this representation, we consider only non-zero values along with their row and column index values. In this representation, the 0th row stores total rows, total columns and total non-zero values in the matrix.

For example, consider a matrix of size 5 X 6 containing 6 number of non-zero values. This matrix can be represented as shown in the image...



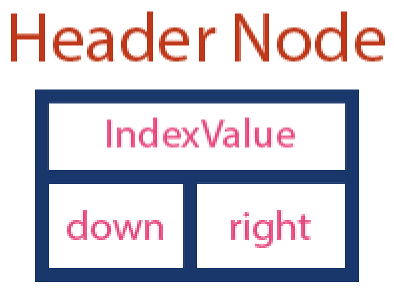
$$\begin{bmatrix} 0 & 0 & 0 & 0 & 9 & 0 \\ 0 & 8 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 2 & 0 & 0 & 0 \end{bmatrix}$$

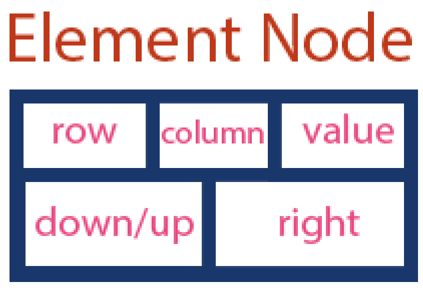
Rows	Columns	Values
5	6	6
0	4	9
1	1	8
2	0	4
2	2	2
3	5	5
4	2	2

In above example matrix, there are only 6 non-zero elements (those are 9, 8, 4, 2, 5 & 2) and matrix size is 5 X 6. We represent this matrix as shown in the above image. Here the first row in the right side table is filled with values 5, 6 & 6 which indicates that it is a sparse matrix with 5 rows, 6 columns & 6 non-zero values. Second row is filled with 0, 4, & 9 which indicates the value in the matrix at 0th row, 4th column is 9. In the same way the remaining non-zero values also follows the similar pattern.

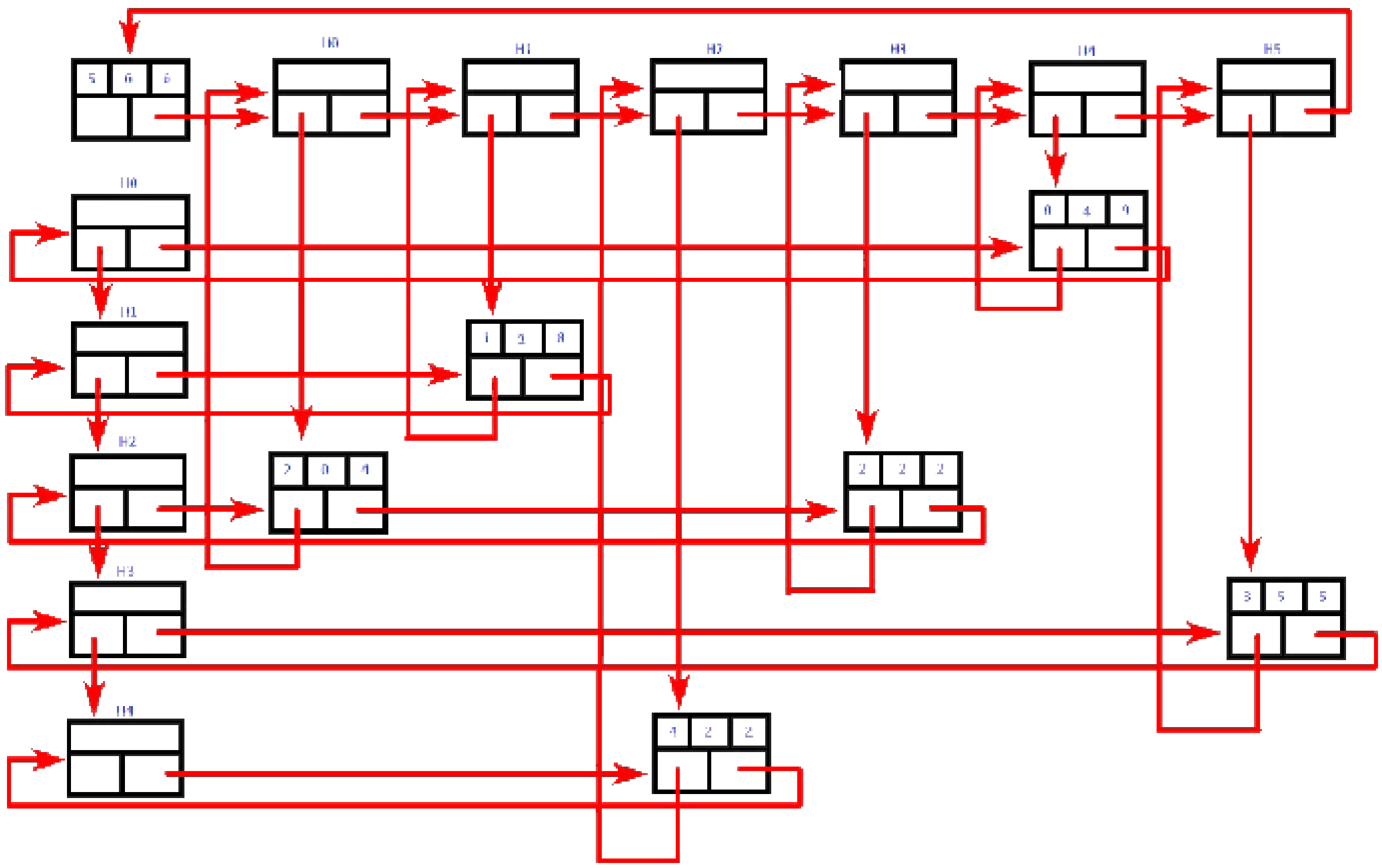
Linked Representation

In linked representation, we use linked list data structure to represent a sparse matrix. In this linked list, we use two different nodes namely header node and element node. Header node consists of three fields and element node consists of five fields as shown in the image...





Consider the above same sparse matrix used in the Triplet representation. This sparse matrix can be represented using linked representation as shown in the below image...



In above representation, H0, H1,...,H5 indicates the header nodes which are used to represent indexes. Remaining nodes are used to represent non-zero elements in the matrix, except the very first node which is used to represent abstract information of the sparse matrix (i.e., It is a matrix of 5 X 6 with 6 non-zero elements).

In this representation, in each row and column, the last node right field points to its respective header node.

UNIT-3

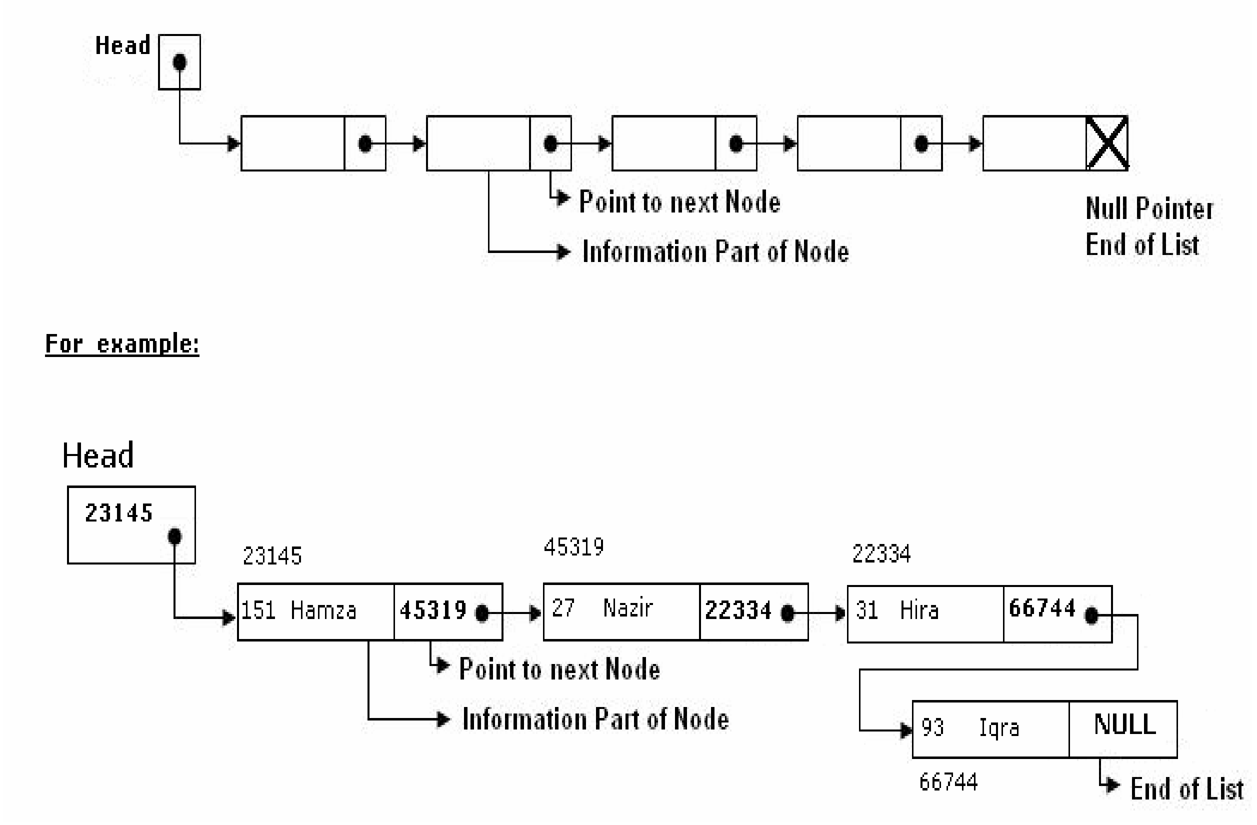
Linked Lists

3.1 Introduction

A linked list or one way list is a linear collection of data elements, called nodes, where the linear order is given by means of "pointers". Each node is divided into two parts.

- The first part contains the information of the element.
- The second part called the link field contains the address of the next node in the list.

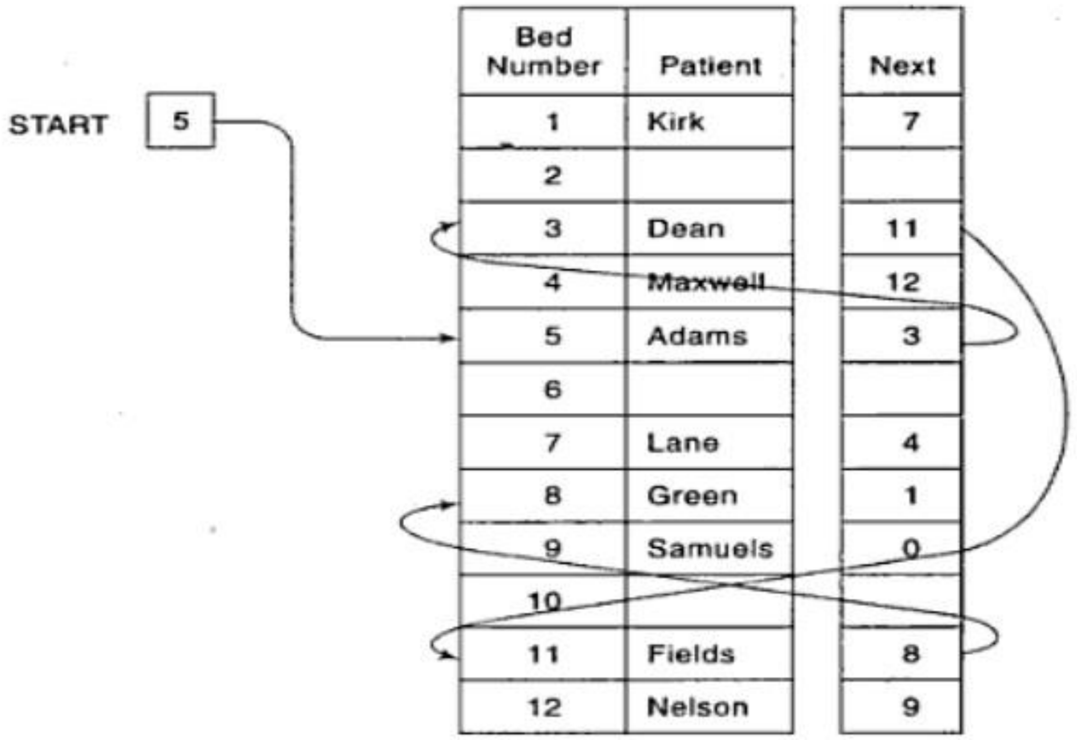
To see this more clearly lets look at an example:



The ***Head*** is a special pointer variable which contains the address of the first node of the list. If there is no node available in the list then ***Head*** contains ***NULL***value that means, List is empty.The left part of the each node represents the information part of the node, which may contain an entire record of data (e.g. ID, name, marks, age etc). the right part represents pointer/link to the next node. The next pointer of the last node is ***null*** pointer signal the end of the list.

3.2 Representation of Linked list in memory

Linked list is maintained in memory by two linear arrays: **INFO** and **LINK** such that **INFO [K]** and **LINK [K]** contain respectively the information part and the next pointer field of node of **LIST**. LIST also requires a variable name such as **START** which contains the location of the beginning of the list and the next pointer denoted by **NULL** which indicates the end of the **LIST**.



Bed Number	Patient	Next
1	Kirk	7
2		
3	Dean	11
4	Maxwell	12
5	Adams	3
6		
7	Lane	4
8	Green	1
9	Samuels	0
10		
11	Fields	8
12	Nelson	9

3.3 Operations on Linked List

There are several operations associated with linked list i.e.

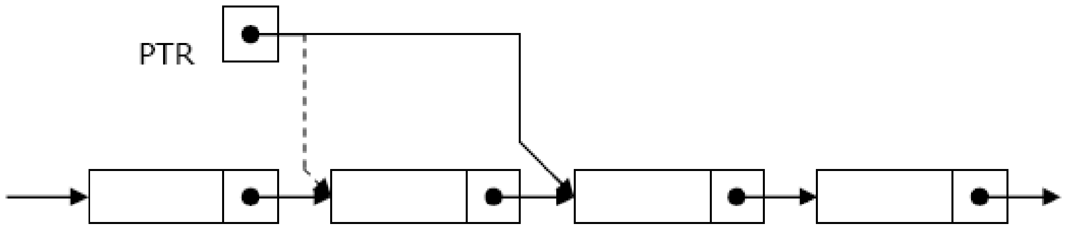
a) Traversing a Linked List

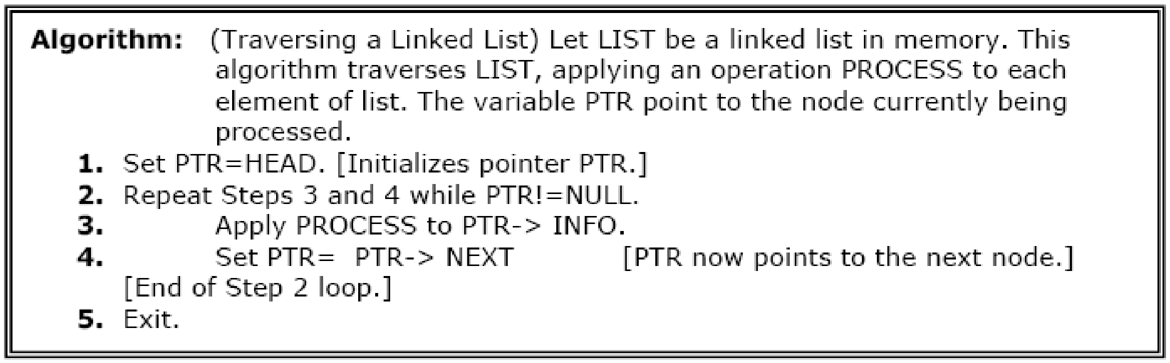
Suppose we want to traverse LIST in order to process each node exactly once. The traversingalgorithm uses a pointer variable PTR which points to the node that is currently being processed.

Accordingly, PTR->NEXT points to the next node to be processed so,

PTR=HEAD [Moves the pointer to the first node of the list]

PTR=PTR->NEXT [Moves the pointer to the next node in the list.]

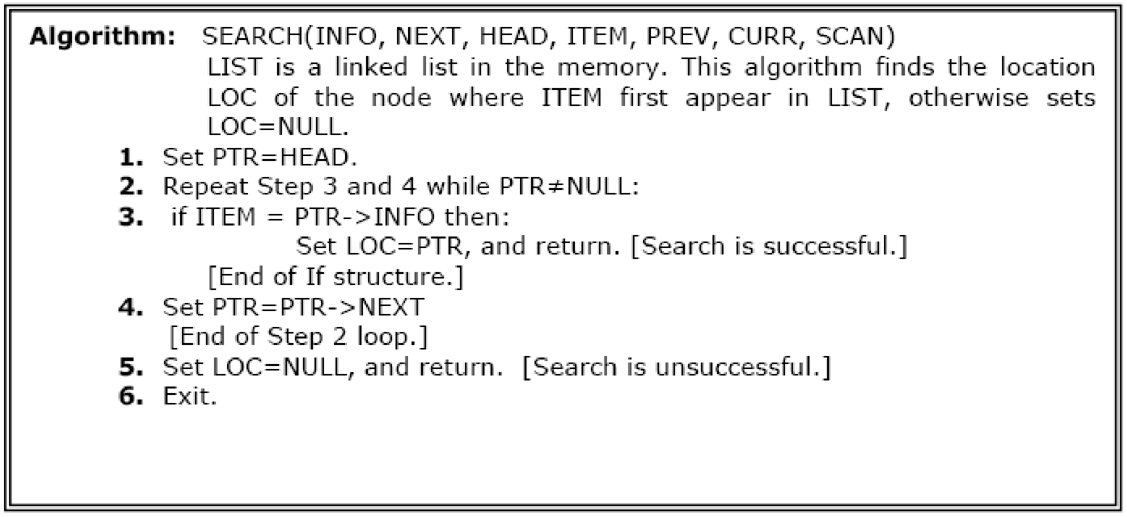




```
Algorithm:  (Traversing a Linked List) Let LIST be a linked list in memory. This
            algorithm traverses LIST, applying an operation PROCESS to each
            element of list. The variable PTR point to the node currently being
            processed.
    1. Set PTR=HEAD. [Initializes pointer PTR.]
    2. Repeat Steps 3 and 4 while PTR!=NULL.
    3.          Apply PROCESS to PTR-> INFO.
    4.          Set PTR=  PTR-> NEXT          [PTR now points to the next node.]
       [End of Step 2 loop.]
    5. Exit.
```

b) Searching a Linked List:

Let list be a linked list in the memory and a specific ITEM of information is given to search. IfITEM is actually a key value and we are searching through a LIST for the record containing ITEM, then ITEM can appear only once in the LIST. Search for wanted ITEM in List can be performed by traversing the list using a pointer variable PTR and comparing ITEM with the contents PTR->INFO of each node, one by one of list.



```
Algorithm:  SEARCH(INFO, NEXT, HEAD, ITEM, PREV, CURR, SCAN)
            LIST is a linked list in the memory. This algorithm finds the location
            LOC of the node where ITEM first appear in LIST, otherwise sets
            LOC=NULL.
    1. Set PTR=HEAD.
    2. Repeat Step 3 and 4 while PTR≠NULL:
    3.  if ITEM = PTR->INFO then:
                Set LOC=PTR, and return. [Search is successful.]
        [End of If structure.]
    4. Set PTR=PTR->NEXT
       [End of Step 2 loop.]
    5. Set LOC=NULL, and return.  [Search is unsuccessful.]
    6. Exit.
```

Searching in sorted list

Algorithm: **SRCHSL (INFO, LINK, START, ITEM, LOC)**

LIST is sorted list (Sorted in ascending order) in memory. This algorithm finds the location LOC of the node where ITEM first appears in LIST or sets LOC=NULL.

```
1. Set PTR:= START
2. Repeat while PTR ≠ NULL
If ITEM >INFO[PTR], then:
Set PTR := LINK[PTR]
Else If ITEM = INFO[PTR], then:
Set LOC := PTR
Return
Else Set LOC:= NULL
Return
[End of If structure]
[End of step 2 Loop]
```

3. Set LOC:= NULL
4. Return

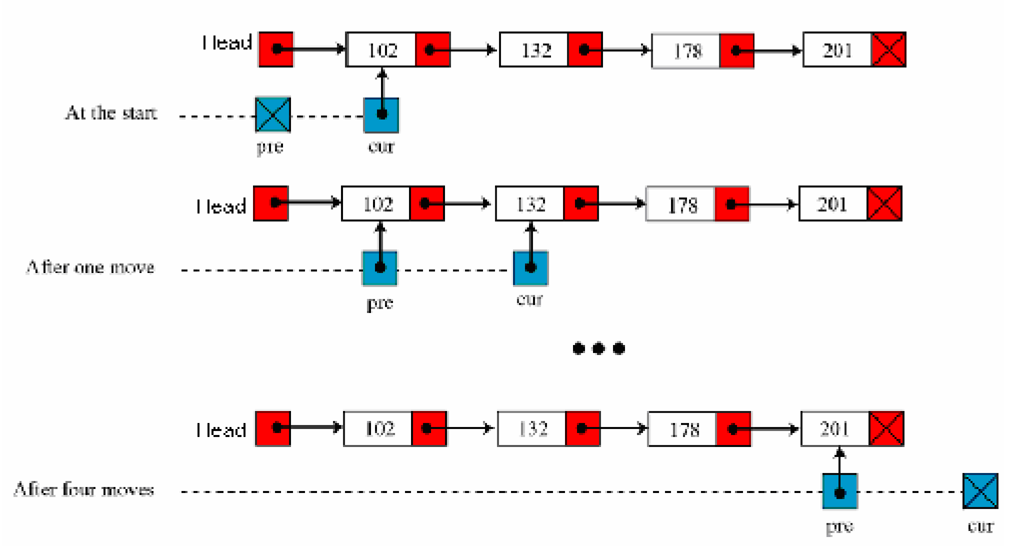
Search Linked List for insertion and deletion of Nodes:

Both insertion and deletion operations need searching the linked list.

- ➢ To add a new node, we must identify the logical predecessor (address of previous node) where the new node is to be inserting.
- ➢ To delete a node, we must identify the location (addresses) of the node to be deleted and its logical predecessor (previous node).

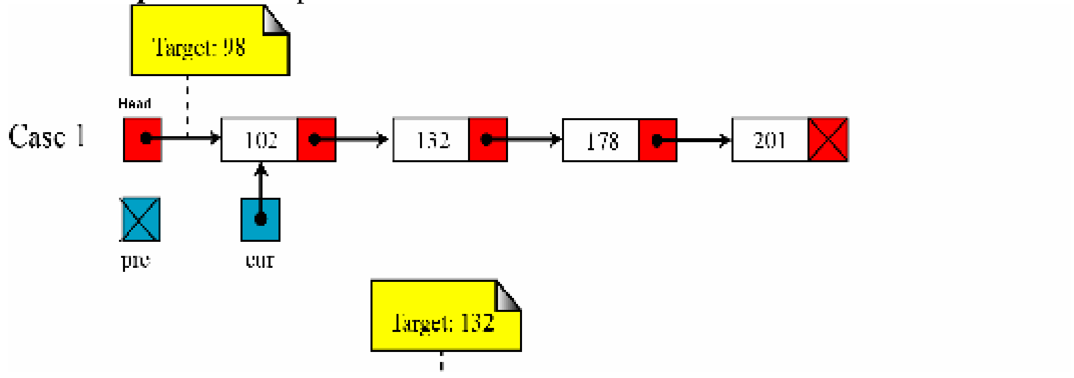
Basic Search Concept

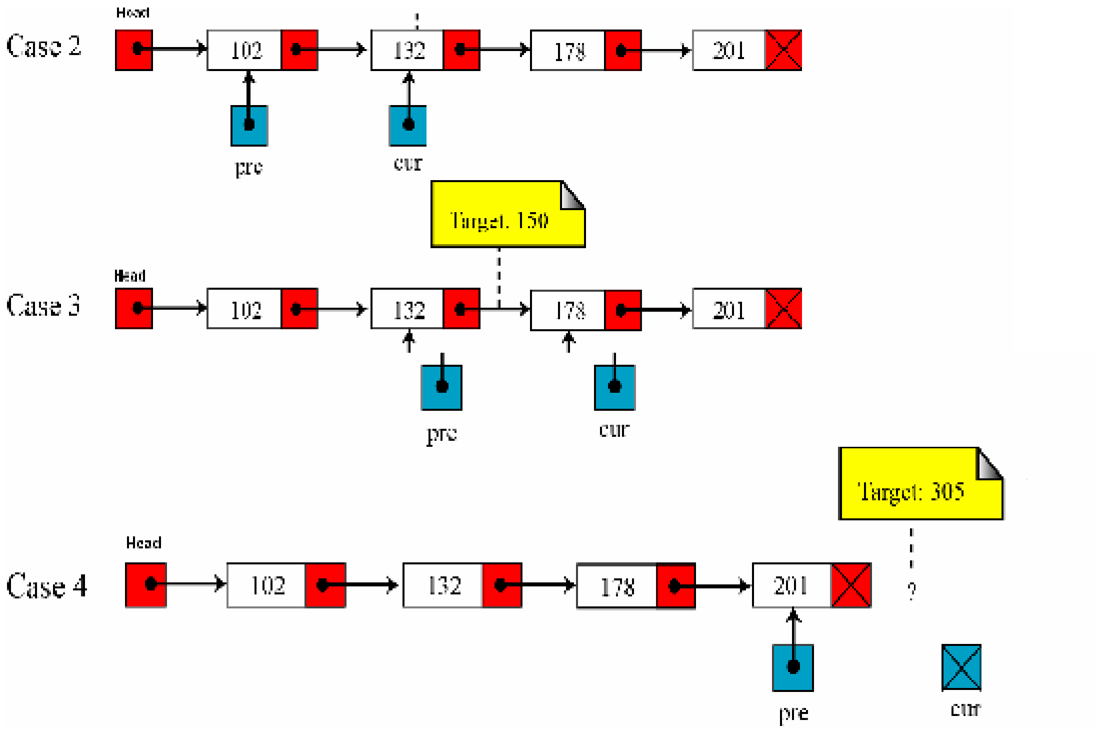
Assume there is a sorted linked list and we wish that after each insertion/deletion this list should always be sorted. Given a target value, the search attempts to locate the requested node in the linked list. Since nodes in a linked list have no names, we use two pointers, **pre** (for previous) and**cur** (for current) nodes. At the beginning of the search, the **pre** pointer is **null** and the **cur** pointer points to the first node (**Head**). The search algorithm moves the two pointers together towards the end of the list. Following Figure shows the movement of these two pointers through the list in an extreme case scenario: when the target value is larger than any value in the list.



Moving of ***pre*** and ***cur*** pointers in searching a linked list

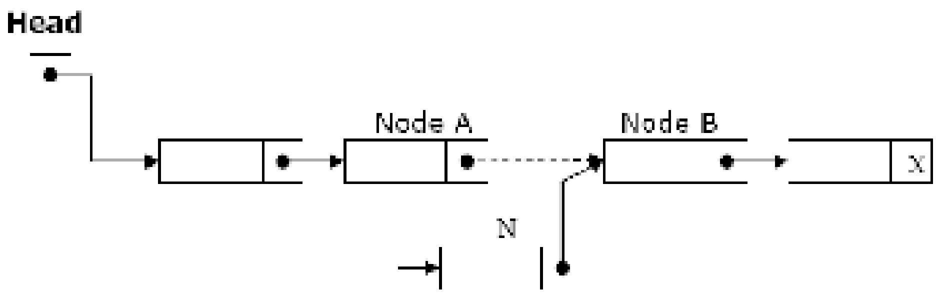
Values of ***pre*** and ***cur*** pointers in different cases





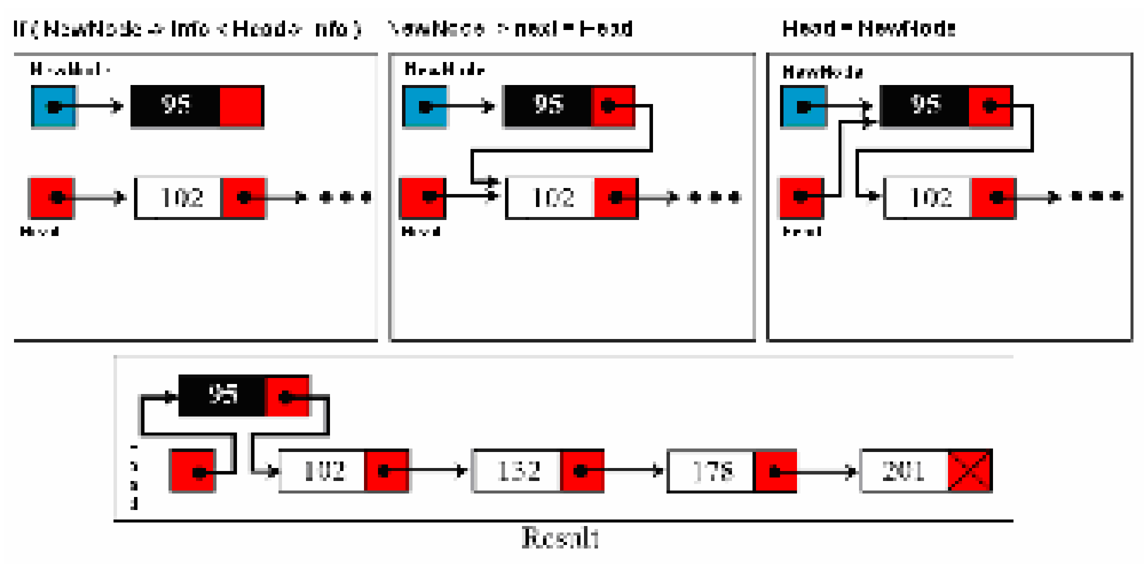
c) Insertion into a Linked List:

If a node N is to be inserted into the list between nodes **A** and **B** in a linked list named LIST. It Can be shown as:



Inserting at the Beginning of a List:

If the linked list is sorted list and new node has the least low value already stored in the list i.e. (***if New->info < Head->info)*** then new node is inserted at the beginning / Top of the list.



```
Algorithm: INSFIRST (INFO, LINK,START,AVAIL,ITEM)
This algorithm inserts ITEM as the first node in the list
Step 1: [OVERFLOW ?] If AVAIL=NULL, then
Write: OVERFLOW
Return
          Step 2: [Remove first node from AVAIL list ]
Set NEW:=AVAIL and AVAIL:=LINK[AVAIL].
Step 3: Set INFO[NEW]:=ITEM [Copies new data into new node]
Step 4: Set LINK[NEW]:= START
[New node now points to original first node]
Step 5: Set START:=NEW [Changes START so it points to new
node]
Step 6: Return
```

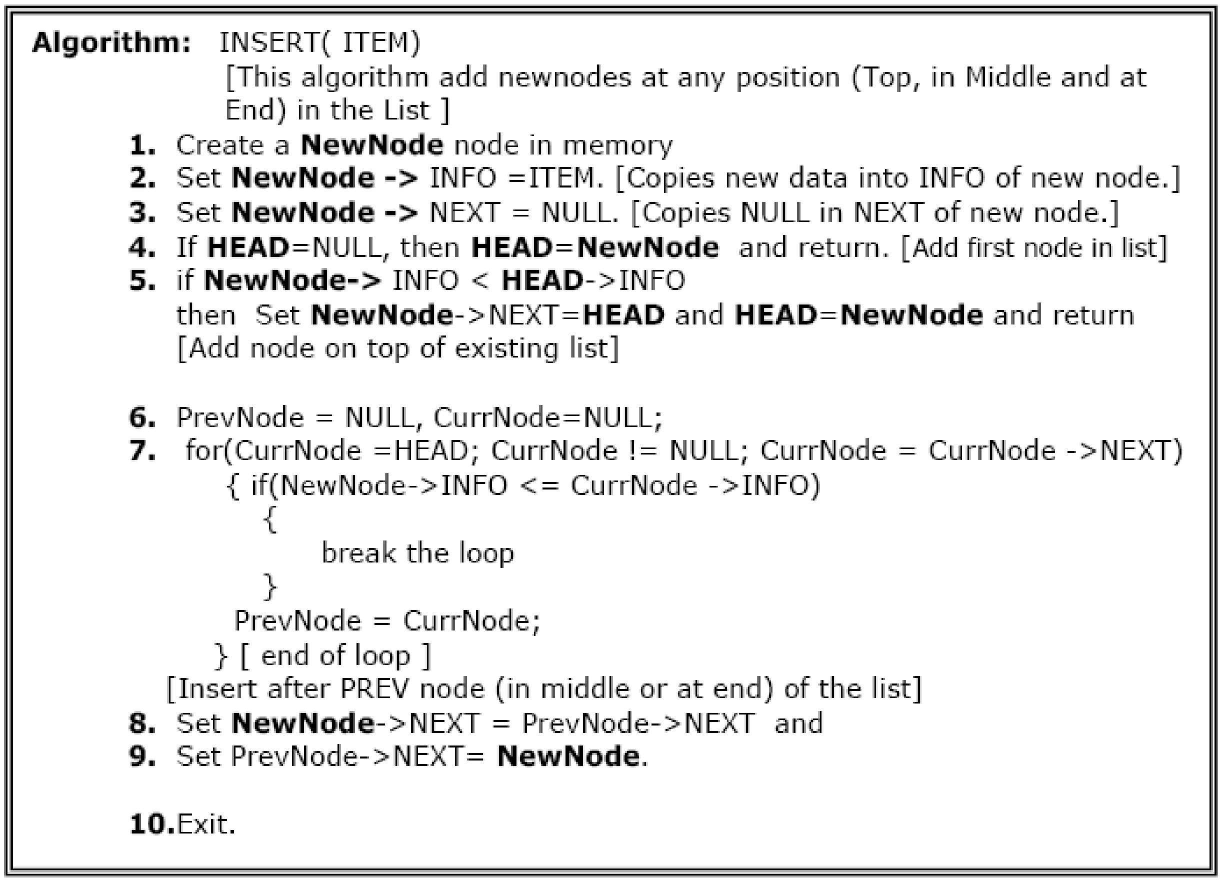
Inserting after a given node

```
Algorithm: INSLOC(INFO, LINK,START,AVAIL, LOC, ITEM)
This algorithm inserts ITEM so that ITEM follows the
node with location LOC or inserts ITEM as the first node
when LOC =NULL
Step 1: [OVERFLOW] If AVAIL=NULL, then:
Write: OVERFLOW
Return
Step 2: [Remove first node from AVAIL list]
Set NEW:=AVAIL and AVAIL:=LINK[AVAIL]
Step 3: Set INFO[NEW]:= ITEM [Copies new data into new node]
Step 4: If LOC=NULL, then:
```

Set LINK[NEW]:=START and START:=NEW
Else:
Set LINK[NEW]:=LINK[LOC] and LINK[LOC]:= NEW
[End of If structure]
Step 5: Return

Inserting a new node in list:

The following algorithm inserts an ITEM into LIST.



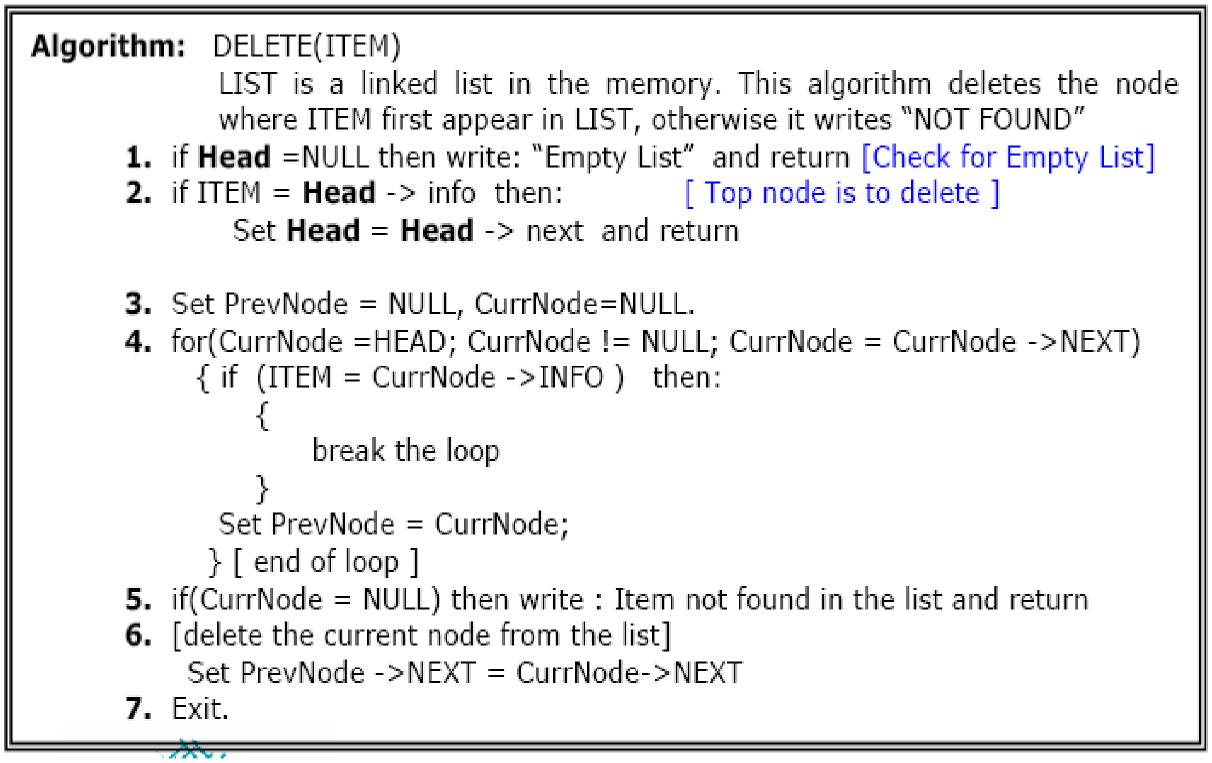
```
Algorithm:  INSERT( ITEM)
            [This algorithm add newnodes at any position (Top, in Middle and at
            End) in the List ]
     1. Create a NewNode node in memory
     2. Set NewNode -> INFO =ITEM. [Copies new data into INFO of new node.]
     3. Set NewNode -> NEXT = NULL. [Copies NULL in NEXT of new node.]
     4. If HEAD=NULL, then HEAD=NewNode  and return. [Add first node in list]
     5. if NewNode-> INFO < HEAD->INFO
        then  Set NewNode->NEXT=HEAD and HEAD=NewNode and return
        [Add node on top of existing list]

     6. PrevNode = NULL, CurrNode=NULL;
     7.  for(CurrNode =HEAD; CurrNode != NULL; CurrNode = CurrNode ->NEXT)
            { if(NewNode->INFO <= CurrNode ->INFO)
                {
                     break the loop
                }
             PrevNode = CurrNode;
            } [ end of loop ]
       [Insert after PREV node (in middle or at end) of the list]
     8. Set NewNode->NEXT = PrevNode->NEXT  and
     9. Set PrevNode->NEXT= NewNode.

     10.Exit.
```

d) Delete a node from list:

The following algorithm deletes a node from any position in the LIST.



```
Algorithm:  DELETE(ITEM)
            LIST is a linked list in the memory. This algorithm deletes the node
            where ITEM first appear in LIST, otherwise it writes "NOT FOUND"
      1. if Head =NULL then write: "Empty List"  and return [Check for Empty List]
      2. if ITEM = Head -> info  then:              [ Top node is to delete ]
             Set Head = Head -> next  and return

      3. Set PrevNode = NULL, CurrNode=NULL.
      4. for(CurrNode =HEAD; CurrNode != NULL; CurrNode = CurrNode ->NEXT)
           { if  (ITEM = CurrNode ->INFO )  then:
                 {
                       break the loop
                 }
             Set PrevNode = CurrNode;
            } [ end of loop ]
      5. if(CurrNode = NULL) then write : Item not found in the list and return
      6. [delete the current node from the list]
           Set PrevNode ->NEXT = CurrNode->NEXT
      7. Exit.
```

e) Concatenating two linear linked lists

Algorithm: Concatenate(INFO,LINK,START1,START2)

This algorithm concatenates two linked lists with start pointers START1 and START2

```
Step 1: Set PTR:=START1
Step 2: Repeat while LINK[PTR]≠NULL:
Set PTR:=LINK[PTR]
[End of Step 2 Loop]
Step 3: Set LINK[PTR]:=START2
Step 4: Return
```

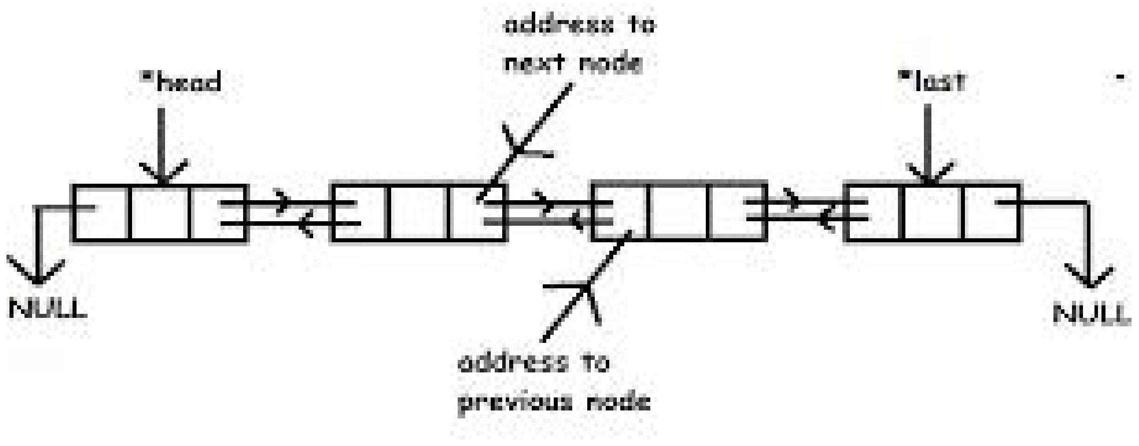
// A Program that exercise the operations on Liked List

```
#include<iostream.h>
#include <malloc.h>
#include <process.h>
struct node
{
int info;
struct node *next;
};
struct node *Head=NULL;
struct node *Prev,*Curr;
voidAddNode(int ITEM)
{
struct node *NewNode;
NewNode = new node;
```

```
// NewNode=(struct node*)malloc(sizeof(struct node));
NewNode->info=ITEM; NewNode->next=NULL;
if(Head==NULL) { Head=NewNode; return; }
if(NewNode->info < Head->info)
{ NewNode->next = Head; Head=NewNode; return;}
Prev=Curr=NULL;
for(Curr = Head ; Curr != NULL ; Curr = Curr ->next)
{
if(NewNode->info <Curr ->info) break;
elsePrev = Curr;
}
NewNode->next = Prev->next;
Prev->next = NewNode;
} // end of AddNode function
voidDeleteNode()
        { intinf;
if(Head==NULL){ cout<< "\n\n empty linked list\n"; return;}
cout<< "\n Put the info to delete: ";
cin>>inf;
if(inf == Head->info) // First / top node to delete
{ Head = Head->next; return;}
Prev = Curr = NULL;
for(Curr = Head ; Curr != NULL ; Curr = Curr ->next )
{
if(Curr ->info == inf) break;
Prev = Curr;
}
if(Curr == NULL)
cout<<inf<< " not found in list \n";
else
{ Prev->next = Curr->next; }
}// end of DeleteNode function
void Traverse()
{
for(Curr = Head; Curr != NULL ; Curr = Curr ->next )
cout<<Curr ->info<<"\t";
} // end of Traverse function
int main()
{ intinf, ch;
while(1)
{ cout<< " \n\n\n\n Linked List Operations\n\n";
cout<< " 1- Add Node \n 2- Delete Node \n";
cout<< " 3- Traverse List \n 4- exit\n";
cout<< "\n\n Your Choice: "; cin>>ch;
switch(ch)
{ case 1: cout<< "\n Put info/value to Add: ";
cin>>inf);
AddNode(inf);
break;
case 2: DeleteNode(); break;
case 3: cout<< "\n Linked List Values:\n";
Traverse(); break;
case 4: exit(0);
} // end of switch
} // end of while loop
return 0;
} // end of main ( ) function
```

3.4 Doubly Linked Lists

A doubly linked list is a list that contains links to next and previous nodes. Unlike singly linked lists where traversal is only one way, doubly linked lists allow traversals in both ways.



Dynamic Implementation of doubly linked list

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
structnode
{
structnode *previous;
intdata;
structnode *next;
}*head, *last;
voidinsert_begning(intvalue)
{
structnode *var,*temp;
var=(structnode *)malloc(sizeof(structnode));
var->data=value;
```

```
if(head==NULL)
{
head=var;
head->previous=NULL;
head->next=NULL;
last=head;
}
else
{
temp=var;
temp->previous=NULL;
temp->next=head;
head->previous=temp;
head=temp;
}
}
voidinsert_end(intvalue)
{
structnode *var,*temp;
var=(structnode *)malloc(sizeof(structnode));
var->data=value;
if(head==NULL)
{
head=var;
head->previous=NULL;
head->next=NULL;
last=head;
}
else
{
last=head;
while(last!=NULL)
{
temp=last;
last=last->next;
}
last=var;
temp->next=last;
last->previous=temp;
last->next=NULL;
}
}
intinsert_after(intvalue, intloc)
{structnode *temp,*var,*temp1;
var=(structnode *)malloc(sizeof(structnode));
var->data=value;
```

```
if(head==NULL)
{
head=var;
head->previous=NULL;
head->next=NULL;
}
else
{
temp=head;
while(temp!=NULL && temp->data!=loc)
{
temp=temp->next;
}
if(temp==NULL)
{
printf("\n%d is not present in list ",loc);
}
else
{
temp1=temp->next;
temp->next=var;
var->previous=temp;
var->next=temp1;
temp1->previous=var;
}
}
last=head;
while(last->next!=NULL)
{
last=last->next;
}
}
intdelete_from_end()
{
structnode *temp;
temp=last;
if(temp->previous==NULL)
{
free(temp);
head=NULL;
last=NULL;
return0;
}
printf("\nData deleted from list is %d \n",last->data);
last=temp->previous;
last->next=NULL;
```

```
free(temp);
return0;
}
intdelete_from_middle(intvalue)
{
structnode *temp,*var,*t, *temp1;
temp=head;
while(temp!=NULL)
{
if(temp->data == value)
{
if(temp->previous==NULL)
{
free(temp);
head=NULL;
last=NULL;
return0;
}
else
{
var->next=temp1;
temp1->previous=var;
free(temp);
return0;
}
}
else
{
var=temp;
temp=temp->next;
temp1=temp->next;
}
}
printf("data deleted from list is %d",value);
}
voiddisplay()
{
structnode *temp;
temp=head;
if(temp==NULL)
{
printf("List is Empty");
}
while(temp!=NULL)
{
printf("-> %d ",temp->data);
temp=temp->next;
}
}
intmain()
{
intvalue, i, loc;
head=NULL;
printf("Select the choice of operation on link list");
printf("\n1.) insert at begning\n2.) insert at at\n3.) insert at middle");
printf("\n4.) delete from end\n5.) reverse the link list\n6.) display list\n7.)
exit
while(1)
{
printf("\n\nenter the choice of operation you want to do ");
scanf("%d",&i);
switch(i)
{
case1:
{
printf("enter the value you want to insert in node ");
scanf("%d",&value);
insert_begning(value);
display();
break;
}
case2:
{
printf("enter the value you want to insert in node at last ");
scanf("%d",&value);
insert_end(value);
display();
break;
}
case3:
{
printf("after which data you want to insert data ");
scanf("%d",&loc);
printf("enter the data you want to insert in list ");
scanf("%d",&value);
insert_after(value,loc);
display();
break;
}
case4:
{
delete_from_end();
```

```
display();
break;
}
case5:
{
printf("enter the value you want to delete");
scanf("%d",value);
delete_from_middle(value);
display();
break;
}
case6 :
{
display();
break;
}
case7 :
{
exit(0);
break;
}
}
}
printf("\n\n%d",last->data);
display();
getch();
}
```

3.5 Circular Linked List

A circular linked list is a linked list in which last element or node of the list points to first node. For non-empty circular linked list, there are no NULL pointers. The memory declarations for representing the circular linked lists are the same as for linear linked lists. All operations performed on linear linked lists can be easily extended to circular linked lists with following exceptions:

• While inserting new node at the end of the list, its next pointer field is made to point to the first node.

• While testing for end of list, we compare the next pointer field with address of the first Node.Circular linked list is usually implemented using **header linked list**. Header linked list is a linked list which always contains a special node called the **header node**, at the beginning of the list. This header node usually contains vital information about the linked list such as number of nodes in lists, whether list is sorted or not etc. Circular header lists are frequently used instead of ordinary linked lists as manyoperations are much easier to state and implement using header listThis comes from the following two properties of circular header linked lists:

• The null pointer is not used, and hence all pointers contain valid addresses

• Every (ordinary) node has a predecessor, so the first node may not require a special case.

Algorithm: **(Traversing a circular header linked list)**
This algorithm traverses a **circular header linked list** with START pointer storing the address of the header node.

```
Step 1: Set PTR:=LINK[START]
Step 2: Repeat while PTR≠START:
        Apply PROCESS to INFO[PTR]
        Set PTR:=LINK[PTR]
        [End of Loop]
Step 3: Return
```

Searching a circular header linked list

Algorithm: SRCHHL(INFO,LINK,START,ITEM,LOC)
This algorithm searches a circular header linked list

```
Step 1: Set PTR:=LINK[START]
Step 2: Repeat while INFO[PTR]≠ITEM and PTR≠START:
        Set PTR:=LINK[PTR]
        [End of Loop]
Step 3: If INFO[PTR]=ITEM, then:
            Set LOC:=PTR
        Else:
            Set LOC:=NULL
        [End of If structure]
Step 4: Return
```

Deletion from a circular header linked list

Algorithm: DELLOCHL(INFO,LINK,START,AVAIL,ITEM)
This algorithm deletes an item from a circular header linked list.

```
Step 1: CALL FINDBHL(INFO,LINK,START,ITEM,LOC,LOCP)
Step 2: If LOC=NULL, then:
            Write: ‘item not in the list’
            Exit
Step 3: Set LINK[LOCP]:=LINK[LOC] [Node deleted]
Step 4: Set LINK[LOC]:=AVAIL and AVAIL:=LOC
        [Memory retuned to Avail list]
Step 5: Return
```

Searching in circular list

Algorithm: FINDBHL(NFO,LINK,START,ITEM,LOC,LOCP)
This algorithm finds the location of the node to be deleted and the location of the node preceding the node to be deleted

```
Step 1: Set SAVE:=START and PTR:=LINK[START]
Step 2: Repeat while INFO[PTR]≠ITEM and PTR≠START
        Set SAVE:=PTR and PTR:=LINK[PTR]
        [End of Loop]
```

```
Step 3: If INFO[PTR]=ITEM, then:
Set LOC:=PTR and LOCP:=SAVE
Else:
Set LOC:=NULL and LOCP:=SAVE
[End of If Structure]
Step 4: Return
```

Insertion in a circular header linked list

```
Algorithm: INSRT(INFO,LINK,START,AVAIL,ITEM,LOC)
This algorithm inserts item in a circular header linked list
after the location LOC
Step 1:If AVAIL=NULL, then
Write: ‘OVERFLOW’
Exit
Step 2: Set NEW:=AVAIL and AVAIL:=LINK[AVAIL]
Step 3: Set INFO[NEW]:=ITEM
Step 4: Set LINK[NEW]:=LINK[LOC]
Set LINK[LOC]:=NEW
Step 5: Return
```

Insertion in a sorted circular header linked list

```
Algorithm: INSERT(INFO,LINK,START,AVAIL,ITEM)
This algorithm inserts an element in a sorted circular header
linked list
Step 1: CALL FINDA(INFO,LINK,START,ITEM,LOC)
Step 2: If AVAIL=NULL, then
Write: ‘OVERFLOW’
Return
Step 3: Set NEW:=AVAIL and AVAIL:=LINK[AVAIL]
Step 4: Set INFO[NEW]:=ITEM
Step 5: Set LINK[NEW]:=LINK[LOC]
Set LINK[LOC]:=NEW
Step 6: Return
Algorithm: FINDA(INFO,LINK,ITEM,LOC,START)
This algorithm finds the location LOC after which to
insert
Step 1: Set PTR:=START
Step 2: Set SAVE:=PTR and PTR:=LINK[PTR]
Step 3: Repeat while PTR≠START
If INFO[PTR]>ITEM, then
Set LOC:=SAVE
Return
Set SAVE:=PTR and PTR:=LINK[PTR]
[End of Loop]
Step 4: Set LOC:=SAVE
Step 5: Return
```

3.6 Applications of Linked Lists:

- Linked Lists can be used to implement Stacks , Queues.
- Linked Lists can also be used to implement Graphs. (Adjacency list representation of Graph).
- Implementing Hash Tables :- Each Bucket of the hash table can itself be a linked list. (Open chain hashing).
- Undo functionality in Photoshop or Word . Linked list of states.
- A polynomial can be represented in an array or in a linked list by simply storing the coefficient and exponent of each term.
- However, for any polynomial operation , such as addition or multiplication of polynomials , linked list representation is more easier to deal with.
- Linked lists are useful for dynamic memory allocation.
- The real life application where the circular linked list is used is our Personal Computers, where multiple applications are running.
- All the running applications are kept in a circular linked list and the OS gives a fixed time slot to all for running. The Operating System keeps on iterating over the linked list until all the applications are completed.

3.7 Comparison of Linked List and Array

Comparison between array and linked list are summarized in following table –

S.No	Array	Linked List
1.	List of elements stored in contiguous memory location i.e. all elements linked physically.	List of elements need not stored in contiguous memory location i.e. all elements will be linked logically.
2.	Contiguous memory required for complete list that will be large requirements.	Contiguous memory required for single node of list that will be small requirements.
3.	List is static in nature i.e. created at compile time mostly.	List is dynamic i.e. created and manipulated at time of execution .
4.	List can't grow and shrink	List can grow and shrink dynamically
5.	Memory allotted to single item of list can't free	Memory allotted to single node of list can also be free.
6.	Random access of Ith element is possible through a[I]	Sequential access to Ith element i.e. all previous I -1 node have to traverse before.
7.	Traversal easy since any elements can access dynamically and randomly.	Traversal is done node by node hence not as good as in array.
8.	Searching can be linear and if sorted than in array we can also apply binary search of logarithm time complexity.	Searching operation must be linear in case of sorted list also. Time complexity is proportional to list length..
9.	Insertion Deletion is costly since require shifting of many items.	Insertion Deletion is performed by simply pointer exchange. Only set of pointer assignment statements can perform insertion deletion operation.