heterogeneous [different] data elements.

Example

```
struct student
{
    int rollno;
    char name[20];
    float per;
};
```

ita Structure

Data may be organized in many different ways. The logical or mathematical method of a particular organization of data is called a data structure.

Data structure is a method of organizing, manipulating, modifying and handling of different types of data item in computers memory.

The possible way in which data items are logically related is defined as data structure.

In data structure we not only describe a set of objects but also the way how they are related.

Data structure describe a set of operations which may legally be applied to elements of data.

Example.

For integer we can perform

arithmatic operations such as addition, subtraction, division, multiplication, modul and comparision operations like less than greater than, less than or equal to, greater than or equal to.

Therefore data structure is also define as,

"A set of domain D, set of functions F and set of axioms A, then the tripple [D, F, A] denotes data structure."

[Note: Axioms is nothing but truth or real value].

The set of axioms describe semantics [related meaning] of operations.

Usually the tripple {D, F, A} is referred to as an "Abstract Data Type". It is called abstract because the axioms do not imply [involve] a form of representation.

Importance of Data Structure

- It is easy to understand and analize when the data is organized into data structure, because in data structure modules are divided into submodules i.e known as modularity and due to modularity data becames easy to understand and analize.
- It reduces complexity of data when it is in large amount.
- Solving the problem of time consumption with faster access of data.
- Easy to understand throughly the

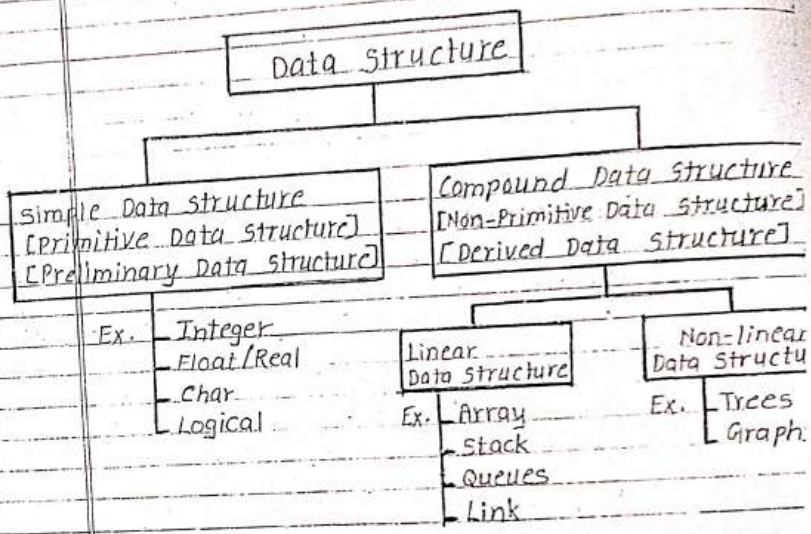
relationship between the data elements that are relevent to the solution of the problem.
- To decide on the operations that must be performed on the logically related data items.
- To decide what problem solving language can best aid in the solution of the problem by allowing the user to express in a 'natural' manner the operation programmer wishes to perform on the data.
- Problems can be solved more easily because a large set of tools is available.

Requirements of Data Structure

- Identify and develop useful mathematical entities and operations and to determine what classes of problems can be solved by using these entities and operations.
- To determine representations for those abstract entities and to implement the abstract operations on these concrete representations.
- The structure should be simple enough that one can effectively process data when necessary.

* Types of Data Structure

Data structure are classified as shown in following figure.



- Data structures are classified into two categories
  I) Primitive data structure
  II) Non-Primitive data structure

I) Primitive data structure

Primitive da
structure are nothing but preliminary data type that allows to use numb

and non-numerical values [alphabetical values]
Examples : int, float, char, Logical

i) int :
int type data structure allow to use only integer data which must not have a decimal point and it could be either positive or negative.
Example 426 , -764 etc

ii) float :
float type of data structure allow to use only Real number or floating point number that must have a decimal point, and it could be either positive or negative.
Example 325.34 , -32.96

iii) char :
char type of data structure allow to use a character constant which is either a single alphabet, a single digit or a single special symbol enclosed within single inverted commas
Example 'A' , '3' , '*'

iv) Logical :
Logical type of data structure allow to use a logical values 0 or 1 and true or false.

II) Non-Primitive data structure

Non-Primitive data structures are made up of primitive data types.

Non-Primitive data structures are classified into two types depending on arrangement of data elements.

1) Linear data structure
2) Non-linear data structure

1) Linear data structure

Linear data structure arrange the data elements in linear that means sequential manner.
Example Array, stack, queue, Linked list

i) Array

- Array is a collection of finite number [n] of homogeneous data elements.
- The element of the array are referenced respectively by an index set consisting of n consecutive numbers.
- The elements of the array are stored respectively in succes memory location.
- Consider array name A then elements of A are denoted subscript notation

$a_1, a_2, a_3, \ldots, a_n$

or 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].
- The number K in A[K] is called subscript and A[K] is called a subscripted variable.

ii) Stack

A stack also called a Last-in First-Out [LIFO] system, is a linear list in which insertion and deletion can take place only at one end, called the top.

iii) Queue

A queue also called a First-in First-Out [FIFO] system, is a linear list in which deletions can take place only at one end of the list i.e. "Front" of the list, and insertions can take place only at the other end of the list, i.e. "Rear" of the list.

iv) Linked List

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 pointer i.e. each node is divided into two parts
a) The first part contains the information of the element.
b) The second part, called the link field or nextpointer field contains the address of the next node in the list.

2) Non-linear data structure

Non-linear data structre arrange the data element not in linear manner means not in sequential manner.
Example Tree, Graph

i) Tree

Tree is a non-linear data structu
This structure is mainly used to represent data containing a hierarchi relationship between elements.
Example family tree, tables of conten

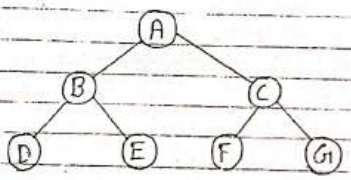


Fig. Tree structure

ii) Graph

The graph is a non-linear structure. Data sometimes contain

relationship between pair of elements which is not hierarchical in nature is called a graph

Example

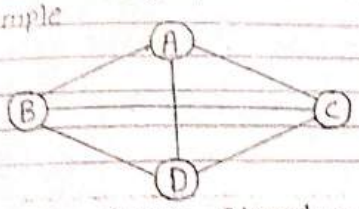


Fig. Graph Structure

Operations on Data Structure

The data appearing in data structures are processed by following operations.

i) Traversing :
Accessing each record exactly once so that certain items in the record may be processed

ii) Searching :
Finding the location of the record with a given key value, or finding the locations of all records which satisfy one or more conditions.

iii) Inserting :
Adding a new record to the structure

iv) Deleting :
Removing the record from the structure.

v) Sorting :
Arranging the records in some logical order.

vi) Merging :
Combining the records in two different sorted files into a single sorted file.

UNIT-I : Chaper 2

Arrays

- Array is a collection of similar data items, it share common variable name and differentiate by their locations.
- An array is a group of related data items called as elements of array.
- The elements of an array are situated at some distance apart from name called as Index.
- Index starts with zero and ends with the number one less than the size of array.
- Example

```
int SALARY[10];
```

This indicate declaration of array SALARY of 10 employees. This complete set of value can be refer to as an array. and individual values are called as elements.

- Array can be of any type as per requirement [i.e. int, char, float] but all elements in an array must be of the same data type.
- Elements of an array can be accessed through Index values from 0 to (n-1) where, n is the size of array.
- Following figure shows memory representation of array SALARY[10] where.
  SALARY[0] is first element of array.
  SALARY[1] is second element of array.

SALARY[2] is third element of array.

SALARY[9] is last element of array.

SALARY

SALARY[0]	
SALARY[1]	
SALARY[2]	
SALARY[3]	
SALARY[9]	

- The value written in square bracket after array name is called index or subscript
- The elements of an array are linearly arranged in memory therefore array is called as linear data structure.
- Following operations can be perform on any linear data structure.

i) Traversing : Processing each element.
ii) Searching : Finding the location of the element.
iii) Inserting : Adding a new element
iv) Deleting : Removing an element
v) Sorting : Arranging the elements in some type of order.
vi) Merging : Combining two list into a single list.

Linear Array

- A linear array is a list of a finite number n of homogeneous [similar] data elements [i.e. data elements of the same type]. such that
  (i) The elements of the array are referenced respectively by an index set consisting of n consecutive numbers.
  (ii) The elements of the array are stored respectively in successive memory locations.

- The number n of elements is called the length or size of the array.
- Length or size of the array can be obtained from the index set by the formula

$$\text{Length} = UB - LB + 1$$

Where,
UB is the largest index, called the upper bound of the array.
LB is the smallest index called the lower bound of the array

- The elements of an array A may be denoted by the subscript notation

$A_1, A_2, A_3, \ldots, A_n$

or by the parenthesis notation

$A(1), A(2), \ldots, A(N)$

or by the bracket notation

$A[1], A[2], A[3], \ldots, A[N]$

- Regardless of the notation, the number K in A[K] is called a subscript or an index and A[K] is called a subscripted variable.
- Subscript allow to access any element referenced by its relative position in A.
- Memory allocation to array is in sequential form.
- Some programming languages [i.e FORTRAN and Pascal] allocate memory space for arrays statically i.e during Program Complication, hence the size of the array is fixed during program execution.
- Some programming languages allow one to read an integer n and then declare an array with n elements. Such programming languages are said to allocate memory dynamically.

Representation of linear array in memory

- Let LA be a linear array in the memory of the computer.
- Memory of the computer is simply a sequence of addressed locations as shown in following figure.

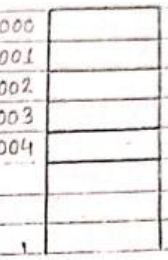


Fig Computer Memory

- The elements of LA are stored in successive memory cells. So computer does not need to keep track of the address of every element of LA, but needs to keep track only of the address of the first element of LA denoted by Base(LA) and also called the base address of LA.
- By using the base address of LA i.e Base(LA) the computer calculates the address of any element of LA by using following formula.

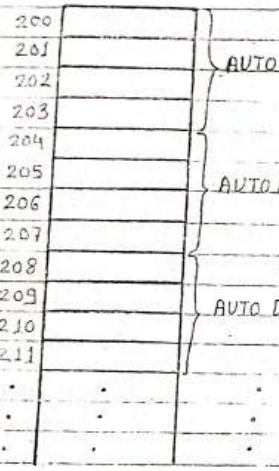
$$LOC(LA[K]) = Base(LA) + w(K - lower\ bound)$$

where,

w is the number of words per memory cell for the array LA. The time to calculate LOC(LA[K]) is essentially the same for any value of K.

- Example

Consider the array AUTO which records the number of automobiles sold each year from 1932 through 1984, as shown in following figure where Base[AUTO] = 200 and w = 4 words per memory cell for AUTO



Base[AUTO] = 200
Loc(AUTO[1932]) = 200
Loc(AUTO[1933]) = 204
Loc(AUTO[1934]) = 208

- The address of the array element for the year K = 1965 can be obtained by formula as

Loc(LA[K]) = Base(LA) + w(K - lower bound)

∴ Loc(AUTO[1965]) = Base(AUTO) + w(1965 - lower bound)
Loc(AUTO[1965]) = 200 + 4(1965 - 1932)
= 200 + 4(33)
= 200 + 132
Loc(AUTO[1965]) = 332

- A collection A of data elements is said to be indexed if any element of A i.e A_K can be located and processed in a time i.e independent of K.
- The above example shows that linear array can be indexed. This is very important property of linear array.

Traversing Linear Arrays

- Let A be a collection of data elements stored in the memory of the computer.
- Suppose we want to print the contents of each element of A or suppose we want to count the number of elements of A with a given property. This can be accomplished by traversing.
- Traversing means accessing and processing [means visiting] each element exactly once
- The following algorithm traverses a linear array LA. The simplicity of the algorithm comes from the fact that LA is a linear structure.

Algorithm [Traversing a Linear Array]

Here
LA = Linear array with lower bound and upper bound
LB = Lower bound
UB = Upper bound
PROCESS = Operation perform on each element of LA.

This algorithm traverses LA applying an operation PROCESS to each element of LA.

Step 1 : [Initialize counter]
Set K := LB

Step 2 : Repeat steps 3 and 4 while K ≤ UB

Step 3 : [Visit element]
Apply PROCESS to LA[K].

Step 4 : [Increase counter]
Set K := K+1
[End of step 2 loop]

Step 5 : Exit

Program

```
/* Program for traversing linear array */
#include<stdio.h>
#include<conio.h>
void main()
{
    int LA[5] = {1,2,3,4,5};
    int K, LB=0, UB=4;
    clrscr();
    while(K<=UB)
    {
        printf("%d \n", LA[K]);
        K = K+1;
    }
    getch();
}
```

Output

1
2
3
4
5

- We also state an alternative form of the algorithm which uses a for loop instead of while loop.

Algorithm [Traversing a linear Array]

Here,
LA = Linear array with lower bound and upper bound
LB = Lower bound
UB = Upper bound
PROCESS = Operation perform on each element of LA.

This algorithm traverses a linear array LA with lower bound and upper bound by applying an operation PROCESS on each element of LA.

Step 1 : Repeat for K=LB to UB
Apply PROCESS to LA[K].
[End of loop]

Step 2 : Exit

Program

```
/* Program for traversing linear array */
#include<stdio.h>
#include<conio.h>
void main()
{
    int LA[5] = {1,2,3,4,5};
    int K, LB=0, UB=4;
    clrscr();
    for(K=LB; K<=UB; K++)
    {
        printf("%d\n", LA[K]);
    }
    getch();
}
```

Output

1
2
3
4
5

Q) Write a program in C to traverse a linear array of 10 integers to find maximum of them

```
/* Program to find maximum from array */
#include<stdio.h>
#include<conio.h>
void main()
{
    int A[10], i, max;
    clrscr();
    printf("\n Enter 10 array elements \n");
    for(i=0; i<10; i++)
    {
```

```
        scanf("%d", &A[i]);
    }
    max = A[0];
    for(i=0; i<10; i++)
    {
        if(max < A[i])
        {
            max = A[i];
        }
    }
    printf("\n Maximum number = %d", max);
    getch();
}
```

Output

Enter 10 array elements
11
24
10
9
25
30
46
5
50
15

Maximum number = 50

Q Write a program in C to traverse a linear array of 10 integers to find minimum of them

```
/* Program to find minimum from array */
#include<stdio.h>
#include<conio.h>
void main()
{
    int A[10], i, min;
    clrscr();
    printf("\n Enter 10 array elements \n");
    for(i=0; i<10; i++)
    {
        scanf("%d", &A[i]);
    }
    min = A[0];
    for(i=0; i<10; i++)
    {
        if(min > A[i])
        {
            min = A[i];
        }
    }
    printf("\n Minimum number = %d", min);
    getch();
}
```

Output

Enter 10 array elements
11
24
10
9
25
30
46
5
50
15

Minimum number = 5

Write a program in C to perform addition of array elements

```
/* Program to perform addition of array
   element */
#include<stdio.h>
#include<conio.h>
void main()
{
    int A[5], i, sum=0;
    clrscr();
    printf("\n Enter 5 array elements \n");
    for (i=0; i<5; i++)
    {
        scanf("%d", &A[i]);
    }
    for(i=0; i<5; i++)
    {
        sum = sum + A[i];
    }
    printf("\n Addition = %d", sum);
    getch();
}
```

output

```
Enter 5 array elements
1
2
3
4
5

Addition = 15
```

Insertion operation of an array

- "Inserting" refers to the operation of adding another element to the collection of array.
- Inserting an element at the "end" of a linear array can be easily done provided the memory space allocated for the array is large enough to accommodate the additional element.
- To insert an element in the middle of the array, half of the elements must be moved downward to new locations to accommodate the new element and keep the order of the other elements.

Ex. Suppose NAME is an 8 element linear array, and suppose five names are in the array, as shown in fig(a). Observe that the names are listed alphabetically, and suppose we want to keep the array names alphabetical at all times. Suppose 'ISHA' is added to the array, then POOJA and RADHA must each be moved downward on location, as shown in fig(a)

	NAME
1	AASHA
2	BHAVANA
3	DISHA
4	POOJA
5	RADHA
6	
7	
8	

	NAME
1	AASHA
2	BHAVANA
3	DISHA
4	ISHA
5	POOJA
6	RADHA
7	
8	

fig.(a)

Algorithm [Inserting element into Linear Array]

INSERT(LA,N,K,ITEM)

Here,
LA = Linear Array
N = Total number of elements in array
K = Position integer such that K≤N
ITEM = Element that will be insert in LA

This algorithm inserts an element ITEM into the Kth position in LA.

Step 1: [Initialize counter.]
Set I:=N

Step 2: Repeat Steps 3 and 4
while I ≥ K

Step 3: [Move Ith element downward.]
Set LA[I+1] := LA[I]

Step 4: [Decrease counter.]
Set I:=I-1
[End of Step 2 loop.]

Step 5: [Insert element.] Set LA[K] := ITEM

Step 6: [Reset N] Set N:=N+1

Step 7: Exit.

Program

3. Write a program in C to insert an element in the linear list of 10 numbers.

or

Write a program to implement insertion operation in linear array.

```
/* Program for insertion of an element
   in array */
#include <stdio.h>
#include <conio.h>
void main()
{
   int LA[10], i, j, N, ITEM, K;
   clrscr();
   printf("\n How many elements you want
           to store in array = ");
   scanf("%d", &N);
   printf("\n Enter array elements \n");
   for(i=0; i<N; i++)
   {
        scanf("%d", &LA[i]);
   }
   printf("\n Enter ITEM and position to
           insert = \n");
   scanf("%d %d", &ITEM, &K);
   K=K-1;
   for(j=N ; j>=K ; j--)
   {
        LA[j+1] = LA[j];
   }
   LA[K]=ITEM;
   N=N+1;
   printf("\n Array Elements after
           insertion \n");
   for(i=0; i<N; i++)
   {
        printf("%d\n", LA[i]);
   }
   getch();
}
```

Output

```
How many elements you want to store in array = 5
Enter array elements
10
20
30
40
50
Enter ITEM and position to insert =
35
4
Array elements after insertion
10
20
30
35
40
50
```

Deletion Operation of an array

- "Deleting" refers to the operation of removing one of the elements of array.
- Deleting an element at the "end" of an array present no difficulties.
- Deleting an element somewhere in the middle of the array would require that each subsequent element be moved one location upward in order to "fill up" the array.
- Example

Suppose NAME is an 8-element linear array and suppose six elements are in the array as shown in fig(a). Suppose DISHA is removed from the array. Then the three names ISHA, POOJA, RADHA must each be moved upward one location as shown in fig(b).

	NAME
1	AASHA
2	BHAVANA
3	DISHA
4	ISHA
5	POOJA
6	RADHA
7	
8	

fig.(a)

	NAME
1	AASHA
2	BHAVANA
3	ISHA
4	POOJA
5	RADHA
6	
7	
8	

fig.(b)

- Such movement of data would be very expensive if thousands of names were in the array

Algorithm [Deleting from a Linear Array]

DELETE (LA, N, K, ITEM)

Here,
LA = Linear array
N = Number of elements present in array
K = Position integer such that $K \leq N$.
ITEM = Element that will be delete from linear array LA.

This algorithm deletes the Kth element from LA

Step 1 : Set ITEM := LA[K]

Step 2 : Repeat for J = K to N-1 ;
[Move J+1 st element upward.]
Set LA[J] := LA[J+1]
[End of loop.]

Step 3 : [Reset the number N of elements in LA.]
Set N := N-1

Step 4 : Exit.

[Note : If many insertions and deletions are to be made in a collection of data elements, then a linear array may not be the most efficient way of storing the data.]

Program

Write a program in C to delete an element from an Array.

```
/* Program for deletion of an element
   from array */
#include <stdio.h>
#include <conio.h>
void main()
{
    int LA[10], i, j, N, ITEM, K;
    clrscr();
    printf("\n How many elements you want
            to store in array = ");
    scanf("%d", &N);
    printf("\n Enter Array elements \n");
    for(i=0; i<N; i++)
    {
        scanf("%d", &LA[i]);
    }
    printf("\n Enter position of item
            which you want to delete = ");
    scanf("%d", &K);
    K=K-1;
    ITEM = LA[K];
    for(j=K; j<N-1; j++)
    {
        LA[j] = LA[j+1];
    }
    N=N-1;
    printf("\n Array after deletion = \n");
    for(i=0; i<N; i++)
    {
        printf("%d \n", LA[i]);
    }
    getch();
}
```

Output

```
How many elements you want to store in array=5
Enter Array Elements
10
20
30
40
50

Enter position of item which you want to
delete = 3

Array after deletion
10
20
40
50
```

Searching

- Let DATA be a collection of data elements in memory, and suppose a specific ITEM of information is given.
- Searching refers to the operation of finding the location LOC of ITEM in DATA, or printing some message that ITEM does not appear there.
- The search is said to be successful if ITEM does appear in DATA and unsuccessful otherwise.
- There are two types of searching
  i) Linear search
  ii) Binary search

i) Linear Search

Linear search algorithm is very simple and also called as sequential search, as it search ITEM in DATA linearly [sequentially] one by one element starting from first to the last, when ITEM found indicate its location LOC

ii) Binary Search

Binary search algorithm require DATA array must be sorted in increasing numerical order. Binary search could not linearly search ITEM in DATA, by comparing all elements of DATA for equality one by one but this algorithm compare ITEM with the middle element of the list. Then decide whether to search previous part or next part of list DATA.

- For small number of elements linear search is better but for large amount of elements binary search is efficient.

Linear search	Binary search
→ 25 → 100 → 35 → 300 → 250	10 11 ← 20 ← 25 → 36 350 450 500 600
250 number is present at 5th position.	20 number is present on third position.

Sort - Arrange

Sorting : Bubble Sort

- Sorting means arranging the elements in some type of order [i.e. asending or descending order].
- Let A be a list of n numbers
- Sorting A refers to the operation of rearranging the elements of A so they are in increasing order i.e so that.
  A[1] < A[2] < A[3] < . . . . < A[N]
- For example.
  Suppose A originally is the list
  8, 4, 19, 2, 7, 13, 5, 16
  After sorting, A is the list
  2, 4, 5, 7, 8, 13, 16, 19
- Sorting may refers to rearranging numerical data in ascending or descending order and nonnumerical data in alphabetical order or reverse of alphabetical order.
- Bubble sort is very simple sorting algorithm. explain as follow.

Bubble Sort

- Suppose the list of numbers A[1], A[2], ... A[N] is in memory.
- The bubble sort algorithm works as follow.

Step 1 : Compare A[1] and A[2] and arrange them in the desired order. So that A[1] < A[2].
Then compare A[2] and A[3] and arrange them so that A[2] < A[3].
Then compare A[3] and A[4] and arrange them. so that A[3] < A[4].
Continue until we compare A[N-1] with A[N] and arrange them so that A[N-1] < A[N].

Step 1 involves n-1 comparisons. When Step 1 is completed. A[N] will contain the largest element.

Step 2 : Repeat Step 1 with one less comparison; i.e. now we stop after we compare and possibly arrange A[N-2] and A[N-1]

Step 2 involves N-2 comparisons and when Step 2 is completed. the second largest element will occupy A[N-1].

Step 3 : Repeat Step 1 with two fewer comparisons; that is, we stop after we compare and possibly rearrange A[N-3] and [N-2]

.
.
.
.

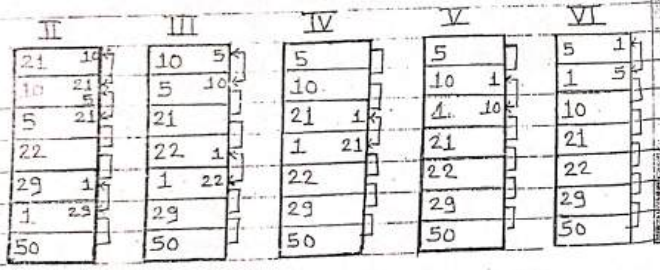
Step N-1 : Compare A[1] with A[2] and arrange them so that A[1] < A[2].
After n-1 steps, the list will be sorted in increasing order.

The process of sequentially traversing through all or part of a list is frequently called a "pass". So each of the above steps is called a pass.

- Thus bubble sort algorithm requires n-1 passes, where n is the number of input items.

Example

Suppose an array A contain 7 elements as 22, 21, 10, 5, 29, 50, 1

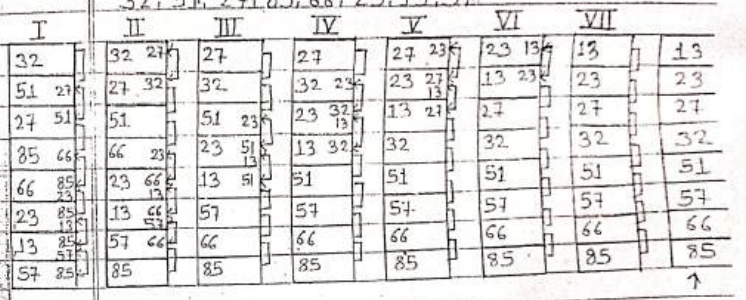


Array after sorting

In this example total 7 elements are present in array therefore require 7-1 = 6 passes.

- Apply Bubble sort technique on linear array with following elements

32, 51, 27, 85, 66, 23, 13, 57



Array After Bubble sorting

- The algorithm for Bubble sort is as follow.

Algorithm [Bubble Sort]

BUBBLE (DATA, N)

Here DATA is an array with N elements. This algorithm sorts the elements in DATA.

Step 1 : Repeat Steps 2 and 3
for K=0 to N-1

Step 2 : Set PTR := 0
[Initializes pass pointer PTR]

Step 3 : Repeat while PTR ≤ N-K-1 :
[Execute pass.]
(a) If DATA[PTR] > DATA[PTR+1], then:
Interchange DATA[PTR] and
DATA[PTR+1].
[End of If structure.]
(b) Set PTR := PTR+1.
[End of Internal loop.]
[End of Step 1 outer loop.]

Step 4 : Exit.

Complexity of the Bubble Sort Algorithm

- The time for a sorting algorithm is measured in terms of the number of comparisons.
- The number f(n) of comparisons in the bubble sort is easily computed.
- Specifically, there are n-1 comparisons during the first pass, which places the largest element in the last position; there are n-2 comparisons in the second step, which places the second largest element

- Thus,

$$f(n) = (n-1) + (n-2) + \cdots + 2 + 1$$
$$= \frac{n(n-1)}{2}$$
$$= \frac{n^2}{2} + O(n)$$
$$= O(n^2)$$

In other words, the time required to execute the bubble sort algorithm is proportional to n^2, where n is the number of input elements.

Program

```
/* Program for Bubble Sort */
#include<stdio.h>
#include<conio.h>
void main()
{
    int N, K, PTR, DATA[10], I, i;
    clrscr();
    printf("\n How many elements you
            want to sorted = ");
    scanf("%d", &N);
    printf("\n Enter Array Elements \n");
    for(i=0; i<N; i++)
    {
        scanf("%d", &DATA[i]);
```

```
printf("\n The original array is \n");
for(i=0; i<N; i++)
{
    printf("\n%d", DATA[i]);
}

for(K=0; K<N; K++)
{
    PTR=0;
    while(PTR < (N-1))
    {
        if(DATA[PTR] > DATA[PTR+1])
        {
            T = DATA[PTR];
            DATA[PTR] = DATA[PTR+1];
            DATA[PTR+1] = T;
        }
        PTR++;
    }
}

printf("\n Array After Sorting \n");
for(i=0; i<N; i++)
{
    printf("\n%d", DATA[i]);
}
getch();
}
```

Output

```
How many elements you want to sorted = 5
Enter Array Elements
44
22
33
11
55

The original array
44
22
33
11
55

Array After Sorting
11
22
33
44
55
```

Multidimensional Arrays : 2D and M-D Arrays

Array is a collection of finite number of homogeneous [similar] data elements.

There are 3 types of array

i) One-dimensional Array :

Arrays where elements are referenced by a single subscript is called as one-dimensional array

Ex. int A[5];

ii) Two-dimensional Array :

Arrays where elements are referenced by two subscripts called as two-dimensional array.

Example int A[3][3];

iii) Multidimensional Array :

Arrays where elements are referenced by three or more subscripts are called as multidimensional arrays

Example. int A[2][3][1];

Mateices & Tables in maths & busi in Applica

* Two-Dimensional Arrays

- A two dimensional array is a collection of similar data elements that can be accessed with two dimensions.
- A two-dimensional $m \times n$ array A is a collection of $m.n$ data elements such that each element is specified by a pair of integers [such as J,K], called subscripts, with the property that

$1 \leq J \leq m$ and $1 \leq K \leq n$

- The element of two-dimensional array A with first subscript J and second subscript K will be denoted by

A_{JK} or $A[J,K]$

- Two dimensional arrays are called matrices in mathematics and tables in business applications.
- Two-dimensional arrays are sometimes called matrix arrays.
- There is a standard way of drawing a two-dimensional $m \times n$ array A where the elements of A form a rectangular array with m rows and n columns and where the element A[J,K] appears in row J and column K. [A row is a horizontal list of elements, and a column is a vertical list of elements.]

- Example

Following array A has 3 rows and 4 columns. means array A[3,4] or A[3][4] is as shown below.

Columns

Rows	col1	col2	col3	col4
row1	A[1,1]	A[1,2]	A[1,3]	A[1,4]
row2	A[2,1]	A[2,2]	A[2,3]	A[2,4]
row3	A[3,1]	A[3,2]	A[3,3]	A[3,4]

Two-dimensional 3X4 Array A

- A is a two-dimensional mxn array, whose first dimension contains the index set 1,.....,m, with lower bound 1 and upper bound m, and the second dimension of A contains the index set 1,2,.....,n, with lower bound 1 and upper bound n.
- The length of a dimension is the number of integers in its index set.
- The pair of lengths mxn is called the size of the array.
- The length of a given dimension can be obtained from the formula.

Length = Upper bound - Lower bound +1

i.e. : Length = UB - LB + 1

Array with LB=1 is regular other than this is non regular.

- Example.

Suppose DATA is a two-dimensional 4X8 array with elements of the real type. It can be declare in various languages as

FORTRAN : REAL DATA(4,8)
PL/1 : DECLARE DATA(4,8) FLOAT;
Pascal : VAR DATA:ARRAY[1...4,1...8] OF REAL;
C : FLOAT DATA[4][8];

* Representation of Two-Dimensional Array in Memory

- Let A be a two-dimensional mxn array.
- A is a rectangular array of elements with m rows and n columns.
- The array will be represented in memory by a block of m*n [m.n] sequential memory locations.
- The programming language will store the array A either
  (1) Column by column, is called column-major order
  (2) row by row, is called row-major order.
- Example

Suppose Array A is two dimensional array of 3*4 size as.

		Col 0	Col 1	Col 2	Col 3
	row 0	1	2	3	5
A =	row 1	1	3	2	4
	row 2	3	2	4	1

3X4

- This is represented in computers memory in column-major order or row-major order as shown in following figure.

Column-major order

1	(0,0)	Col 0
1	(1,0)	
3	(2,0)	
2	(0,1)	Col 1
3	(1,1)	
2	(2,1)	
3	(0,2)	Col 2
2	(1,2)	
4	(2,2)	
5	(0,3)	Col 3
4	(1,3)	
1	(2,3)	

(a)

Row-major order

1	(0,0)	row 0
2	(0,1)	
3	(0,2)	
5	(0,3)	
1	(1,0)	row 1
3	(1,1)	
2	(1,2)	
4	(1,3)	
3	(2,0)	row 2
2	(2,1)	
4	(2,2)	
1	(2,3)	

(b)

- Fig (a) represent matrix A in column-major order and fig(b) represent matrix A in row-major order.
- For a linear array LA, the computer does not keep track of the address of every element of the array, but does keep track of Base(LA) i.e the address of the first element of LA.
- For any two-dimensional mxn array A, i.e. the computer keeps track of Base(A) - i.e the address of the first element A[1,1] of A and compute the address of any element Loc(A[I,K]) of array A[I,K] by using following formula:

(i) Column-major order

$$Loc(A[I,K]) = Base(A) + w[M(K-1) + (I-1)]$$

ii) Row-major order

$$Loc(A[I,K]) = Base(A) + w[N(I-1) + (K-1)]$$

Where, w is the number of words per memory location for the array A.
M = Total number of rows
N = Total number of columns
Base(A) = Address of A[1,1].
I and K = variables use to identify position of element.

- Example

Consider 25X4 matrix array SCORE. Suppose Base(SCORE) = 200 and w = 4 then the address of element at 12th row and 3rd column can be calculated as

loc(SCORE[12,3]) = ?

I = 12
K = 3

i) In column-major order

LOC(SCORE[J,K]) = Base(SCORE) + w[M(K-1)+(J-1)]
LOC(SCORE[12,3]) = Base(SCORE) + w[M(3-1)+(12-1)]
= 200 + 4[25(2)+11]
= 200 + 4[50+11]
= 200 + 4[61]
= 200 + 244
LOC(SCORF[12,3]) = 444

ii) In Row-major order

LOC(SCORE[J,K]) = Base(SCORE) + w[N(J-1)+(K-1)]
LOC(SCORE[12,3]) = Base(SCORE) + w[N(12-1)+(3-1)]
= 200 + 4[4(11)+2]
= 200 + 4[44+2]
= 200 + 4[46]
= 200 + 184
LOC(SCORF[12,3]) = 384

Q.) Write a program in C to perform transpose of a matrix and print both original and transpose of matrix in matrix form.

```
/* Program for transpose of matrix */
#include<stdio.h>
#include<conio.h>
void main()
{
    int J,K, A[5][5], m,n;
    clrscr();
    printf("Enter number of rows = ");
    scanf("%d",&m);
    printf("\n Enter number of columns = ");
    scanf("%d",&n);
    printf("\n Enter Elements of matrix \n");
    for(J=0; J<m; J++)
    {
        for(K=0; K<n; K++)
        {
            scanf("%d", &A[J][K]);
        }
    }
    printf("\n The Original matrix is \n");
    for(J=0; J<m; J++)
    {
        for(K=0; K<n; K++)
        {
            printf("\t%d", A[J][K]);
        }
```

```
            printf("\n");
        }
    printf("\n Transpose of matrix is \n");
    for(J=0;J<m; J++)
    {
        for(K=0; K<n; K++)
        {
            printf("\t%d", A[K][J]);
        }
        printf("\n");
    }
    getch();
}
```

Output

```
Enter number of rows = 2
Enter number of columns = 2
Enter Elements of matrix
11
22
33
44
The original matrix is
        11      22
        33      44

Transpose of matrix is
        11      33
        22      44
```

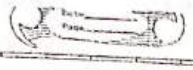
Q.) Write a program in C to perform addition of two matrices

```
/* Program to perform addition of two
   matrices */
#include<stdio.h>
#include<conio.h>
void main()
{
    int J,K, m, n, A[5][5], B[5][5], C[5][5];
    clrscr();
    printf("\n Enter number of rows = ");
    scanf("%d", &m);
    printf("\n Enter number of columns = ");
    scanf("%d", &n);
    printf("\n Enter Elements of first
            matrix A \n");
    for(J=0;J<m ;J++)
    {
        for(K=0; K<n; K++)
        {
            scanf("%d", &A[J][K]);
        }
    }
    printf("\n Enter Elements of second
            matrix B \n");
    for(J=0;J<m;J++)
    {
        for (K=0; K<n; K++)
        {
```

```
            scanf("%d", &B[I][K]);
        }
    }

    printf("\n First matrix A is \n");
    for(I=0; I<m; I++)
    {
        for(K=0; K<n; K++)
        {
            printf("\t %d", A[I][K]);
        }
        printf("\n");
    }
    printf("\n Second matrix B is \n");
    for(I=0; I<m; I++)
    {
        for(K=0; K<n; K++)
        {
            printf("\t %d", B[I][K]);
        }
        printf("\n");
    }

    /* Now perform addition */
    for(I=0; I<m; I++)
    {
        for(K=0; K<n; K++)
        {
            C[I][K] = A[I][K] - B[I][K];
        }
    }

    printf("\n Addition of matrices A and B
                is \n");
    for(I=0; I<m; I++)
    {
        for(K=0; K<n; K++)
        {
            printf("\t %d", C[I][K]);
        }
        printf("\n");
    }
    getch();
}
```

Output

```
Enter number of rows = 2
Enter number of columns = 2
Enter Elements of first matrix A
1
2
3
4

Enter Elements of second matrix B
1
2
3
4
```



First matrix A is

1 2

3 4

Second matrix B is

1 2

3 4

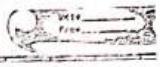
Addition of matrices A and B is

2 4

6 8

Write a program in C to perform subtraction of two matrices

```
/* Program to perform subtraction of two
   matrices */
#include <stdio.h>
#include <conio.h>
void main()
{
    int I,K,m,n, A[5][5], B[5][5], C[5][5];
    clrscr();
    printf("\n Enter number of rows = ");
    scanf("%d", &m);
    printf("\n Enter number of columns = ");
    scanf("%d", &n);
    printf("\n Enter Elements of first
            matrix A \n");
```

```
    for(I=0; I<m; I++)
    {
        for(K=0; K<n; K++)
        {
            scanf("%d", &A[I][K]);
        }
    }
    printf("\n Enter Elements of second
            matrix B \n");
    for(I=0; I<m; I++)
    {
        for(K=0; K<n; K++)
        {
            scanf("%d", &B[I][K]);
        }
    }
    printf("\n First matrix A is \n");
    for(I=0; I<m; I++)
    {
        for(K=0; K<m; K++)
        {
            printf("\t%d", A[I][K]);
        }
        printf("\n");
    }
    printf("\n Second matrix B is \n");
    for(I=0; I<m; I++)
    {
        for(K=0; K<n; K++)
        {
            printf("\t%d", B[I][K]);
```

```
        }
        printf("\n");
    }

    /* Now perform subtraction */
    for(I=0; I<m; I++)
    {
        for(K=0; K<n; K++)
        {
            C[I][K] = A[I][K] - B[I][K];
        }
    }
    printf("\n Subtraction of matrices is \n");
    for(I=0; I<m; I++)
    {
        for(K=0; K<n; K++)
        {
            printf("\t%d", C[I][K]);
        }
        printf("\n");
    }
    getch();
}
```

Output

```
Enter number of rows = 2
Enter number of columns = 2
Enter Elements of first matrix A
5
4
3
2
Enter Elements of second matrix B
1
1
1
1
first matrix A is
        5       4
        3       2

Second matrix B is
        1       1
        1       1

Subtraction of matrices is
        4       3
        2       1
```

Write a program to perform multiplication of two matrices

```
/*Program to perform multiplication of two
  matrices */
#include<stdio.h>
#include<conio.h>
void main();
{
    int A[5][5], B[5][5], C[5][5], J,K,T, m,n;
    clrscr();
    printf("\n Enter number of rows = ");
    scanf("%d", &m);
    printf("\n Enter number of columns = ");
    scanf("%d", &n);
    printf("\n Enter Elements of first
                matrix A \n");
    for(J=0; J<m; J++)
    {
        for(K=0; K<n; K++)
        {
            scanf("%d", &A[J][K]);
        }
    }
    printf("\n Enter Elements of second
                matrix B \n");
    for(J=0; J<m; J++)
    {
        for(K=0; K<n; K++)
        {
            scanf("%d", &B[J][K]);
        }
    }

    printf("\n First matrix A is \n");
    for(J=0; J<m; J++)
    {
        for(K=0; K<n; K++)
        {
            printf("\t %d", A[J][K]);
        }
        printf("\n");
    }
    printf("\n Second matrix B is \n");
    for(J=0; J<m; J++)
    {
        for(K=0; K<n; K++)
        {
            printf("\t %d", B[J][K]);
        }
        printf("\n");
    }

    /* Now perform multiplication */
    for(J=0; J<m; J++)
    {
        for(K=0; K<n; K++)
        {
            C[J][K] = 0;
            for(T=0; T<m; T++)
            {
```

```
                C[I][K] = C[I][K] + A[I][J] *
                          B[J][K];
            }
        }
    }
    printf("\n Multiplication of two matrices is:\n");
    for(I=0; I<m; I++)
    {
        for(K=0; K<n; K++)
        {
            printf("\t%d", C[I][K]);
        }
        printf("\n");
    }
    getch();
}
```

```
First matrix A is
        1       2
        3       4

Second matrix B is
        1       2
        3       4

Multiplication of two matrices is
        7       10
        15      22
```

Output

```
Enter number of rows = 2
Enter number of columns = 2
Enter Elements of first matrix A
1
2
3
4
Enter Elements of second matrix B
1
2
3
4
```

काही वाक्यांत सारखा

Multidimensional Array

- General multidimensional arrays are defined analogously. That means an n dimensional $m_1 \times m_2 \times m_3 \times \ldots \times m_n$
- Array B is a collection of $m_1, m_2, m_3, \ldots, m_n$ data elements in which each element is specified by a list of n integer such as $K_1, K_2, K_3, \ldots, K_n$ called as subscripts, with the priority that

$$1 \le K_1 \le m_1$$
$$1 \le K_2 \le m_2$$
$$1 \le K_3 \le m_3$$
$$\vdots$$
$$1 \le K_n \le m_n$$

- Multidimensional arrays are the arrays in which each element is referred by more than two subscripts.
- In many large computers it may have 7 subscripts array.
- The element of B with subscript $K_1, K_2, K_3, \ldots, K_n$ will be denoted as.

$B_{K_1, K_2, K_3, \ldots, K_n}$

or

$B[K_1, K_2, K_3, \ldots, K_n]$

or

$B[K_1][K_2][K_3]\ldots[K_n]$

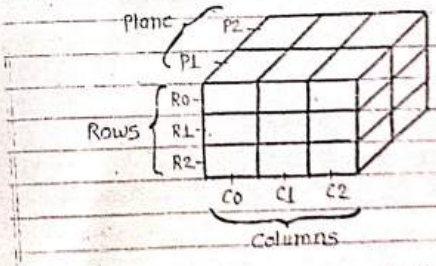
- Multidimensional array can be stored in a sequential memory locations and may be stored in row-major order or column-major order.
- Example : consider in C language

```
        Plane Rows Columns
          ↑    ↑    ↑
int   B [2] [3] [3];
↑     ↑
Array type  Array name
```

Here,

First subscript/dimension specifies plane number.
Second subscript/dimension specifies number of rows.
Third subscript/dimension specifies number of columns.

- This array B can be store logically as given below



- In this example integer array B has 2 planes, 3rows and 3 columns. The

elements of multidimensional array B can be access as

B[0,0,0]	B[0,0,1]	B[0,0,2]
B[0,1,0]	B[0,1,1]	B[0,1,2]
B[0,2,0]	B[0,2,1]	B[0,2,2]
B[1,0,0]	B[1,0,1]	B[1,0,2]
B[1,1,0]	B[1,1,1]	B[1,1,2]
B[1,2,0]	B[1,2,1]	B[1,2,2]

Representation of multidimensional array in memory

- The defination of general multidimensional arrays also permits lower bound other than 1.
- Let B be such an n dimensional array. As before the index set for each dimension of B consists of the consecutive integers from the lower bound to the upper bound of the dimension.
- The length L of dimension i of B is the number of elements in the index set and Li can be calculated as before from.

$$L_i = \text{Upper bound}_i - \text{lower bound}_i + 1$$

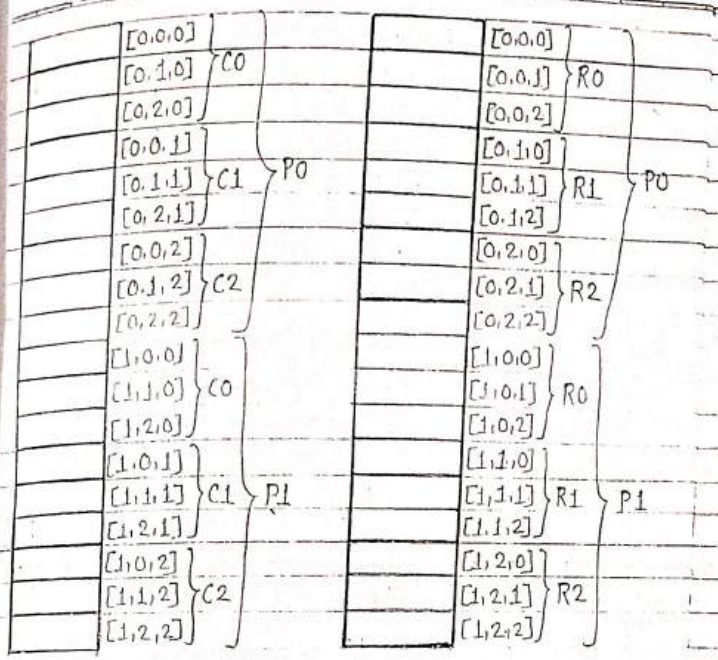
i.e. $\boxed{L_i = UB_i - LB_i + 1}$

- For a given subscript K_i, the effective index E_i of L_i is the number of indices preceding K_i in the index set, and E_i can be calculated from.

$$E_i = K_i - \text{lower bound}_i$$

i.e. $\boxed{E_i = K_i - LB_i}$

- The elements of multidimensional array are represent in memory by two ways
  1) Column-major order
  2) Row-major order
- Suppose multidimensional array B[2][3][3] is represented as shown in following figure.
- fig(a) shows column-major order of multidimensional array B[2][3][3]
- fig(b) shows row-major order of multidimensional array B[2][3][3]



(a) Column - major order

(b) Row - major order.

Then the address $LOC(B[K_1, K_2, \ldots, K_n])$ of an arbitrary element of B can be obtained by the formula:

i) for column - major order

$$LOC(B[K_1, K_2, \ldots, K_n]) = Base(B) + w[(\ldots((E_n L_{n-1} + E_{n-1}) L_{n-2} + \ldots + E_3) L_2 + E_2) L_1 + E_1]$$

ii) For Row major order

$$LOC(B[K_1, K_2, \ldots, K_n]) = Base(B) + w[(\ldots((E_1 L_2 + E_2) L_3 + E_3) L_4 + \ldots + E_{N-1}) L_N + E_N]$$

Where,

Base(B) denotes the address of the first element of B, and w denotes the number of words per memory location.

- Example

Suppose a three-dimensional array MAZE is declared as

MAZE(2:8, -4:1, 6:10)

Then length of the three dimensions of MAZE are respectively.

$L_1 = 8 - 2 + 1 = 7$

$L_2 = 1 - (-4) + 1 = 6$

$L_3 = 10 - 6 + 1 = 5$

Accordingly MAZE contains

$L_1 \cdot L_2 \cdot L_3 = 7 \cdot 6 \cdot 5$

$= 210$ elements.

Suppose Base(MAZE) = 200

$w = 4$

then address of MAZE[5, -1, 8] element of MAZE is obtained as follow

The effective indices of the

subscripts are respectively,

$E_1 = 5-2 = 3$

$E_2 = -1-(-4) = 3$

$E_3 = 8-6 = 2$

i) Using the row-major order we have

$E_1 \cdot L_2 = 3 \cdot 6 = 18$

$E_1 L_2 + E_2 = 18+3 = 21$

$(E_1 L_2 + E_2) L_3 = 21 \cdot 3 = 105$

$[E_1 L_2 + E_2) L_3 + E_3 = 105 + 2 = 107$

Therefore

$LOC(MAZE[5,-1,8]) = 200 + 4(107)$

$= 200 + 428$

$= 628$

ii) Using column-major order we have

$E_3 \cdot L_2 = 2 \cdot 6 = 12$

$E_3 \cdot L_2 + E_2 = 12+3 = 15$

$(E_3 \cdot L_2 + E_2) L_1 = 15 \cdot 7 = 105$

$(E_3 \cdot L_2 + E_2) L_1 + E_1 = 105 + 3 = 107$

Therefore

$LOC(MAZE[5,-1,8]) = 200 + 4(107)$

$= 200 + 428$

$= 628$

UNIT-II : Linked List

Introduction

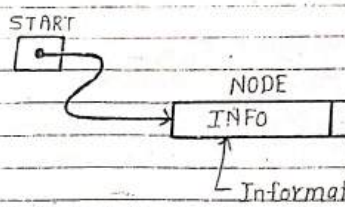
- In many applications we need to crosses data organized into list.
- This data processing involves storing and processing data by adding elements deleting elements, modifying list of elements and arrange the elements in required order.
- One way to store data is by means of linear list that means array.
- Linear list with array representation represent linear relationship between the data elements of an array is reflected by the physical relationship of the data in memory. This makes it easy to compute the address of an element in an array.
- But array have certain disadvantages i.e.
  i) It is expensive to insert and delete elements in an array.
  ii) Array occupies a block of memory space, when additional space is required then we can not double or triple the size of an array, thus array are called dense lists and are said to be static data structure.

- Linked list provides another way to store a list in memory that each element in the list contain a field called a link or

pointer which contains the address of the next element in the list.
- Thus successive elements in the list need not occupy adjacent space in memory. This will make it easier to insert and delete elements in the list.

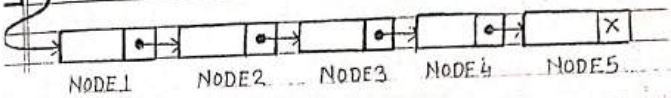
Linked Lists

- A Linked List is also called as one-way list or singly linked list.
- Linked 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.
  i) The first part contains the information of the element.
  ii) The second part called the link field or nextpointer field contains the address of the next node in the list.
- Following diagram shows a node, it contain information field and address of next field called link.



Fig(a). Node that contain information field and nextpointer field

- Following fig.(b) shows a linked list with 5 nodes. This is logical representation of linked list.



Fig(b). Linked list with 5 nodes.

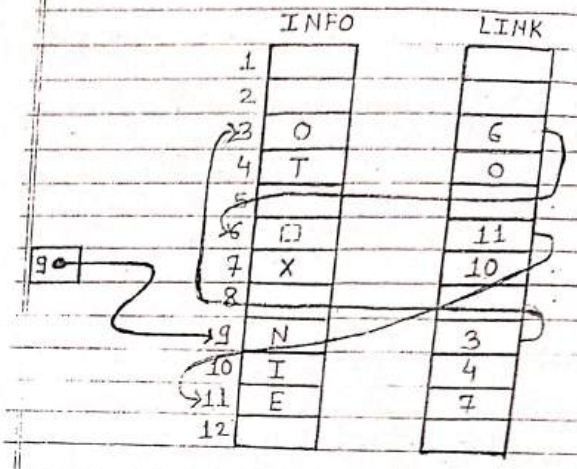
- Each node is pictured with two parts
  i) The left part represents the information part of the node, which may contain an entire record of data items.
  ii) The right part represents the nextpointer field of the node.
- The arrow from the nextpointer field of one node to another node indicate the link to next node.
- The pointer of the last node contains a special value, called the null pointer, which is any invalid address and it is ending of the linked list.
- In actual practice 0 [zero] or negative number is used for the null pointer.
- The null pointer denote by 'X' in the fig(b). signals the end of the list.
- The linked list also contains a list pointer variable called START or NAME.

which contains the address of the first node in the list. Hence there is an arrow drawn from START to the first node.

- If the list has no nodes, such a list is called the null list or empty list and is denoted by the null pointer in the variable START.

Representation of Linked Lists in memory

- Let LIST be a Linked List.
- The linked list LIST will be maintained in memory with two linear arrays INFO & LINK
- INFO[K] contain the information part of a node of LIST.
- LINK[K] contain the nextpointer field of a node of LIST.
- LIST also require a variable name such as START which contains the location of the beginning of the list.
- The nextpointer section denotes by NULL indicates the end of the list.
- Subscripts of the arrays INFO and LINK will usually be positive and NULL = 0
- Following example shows a linked list that represents the character string NO EXIT in the linear arrays INFO and LINK.
START holds address of first node of the LIST that is 9



Where,

START = 9 ; INFO[9] = N	1st character
LINK[9] = 3 ; INFO[3] = O	2nd character
LINK[3] = 6 ; INFO[6] = [] [Blank]	3rd character
LINK[6] = 11 ; INFO[11] = E	4th character
LINK[11] = 7 ; INFO[7] = X	5th character
LINK[7] = 10 ; INFO[10] = I	6th character
LINK[10] = 4 ; INFO[4] = T	7th character
LINK[4] = 0 i.e. NULL so the LIST has ended.	

It represent NO EXIT as a character string.

aversing a Linked List

et LIST be a linked list in memory stored in linear arrays INFO and LINK with START pointing to the first element and NULL indicating the end of LIST.

Suppose we want to traverse LIST in order to process each node exactly once.

Traversing algorithm uses a pointer variable PTR which points to the node that is currently being processed.

Therefore, LINK[PTR] points to the next node to be processed. Thus

PTR := LINK[PTR]

moves the pointer to the next node in the list. shown in following figure.

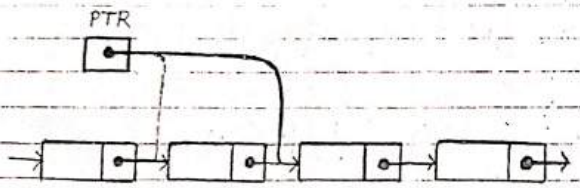


Fig. PTR := LINK[PTR]

The algorithm for traversing a linked list is as follow.

Algorithm : Traversing a Linked List

Let
LIST = Linked list in memory.
This algorithm traverses LIST, applying an operation PROCESS to each element of LIST.
The variable PTR points to the node currently being processed.

Step 1 : Set PTR := START
[Initializes pointer PTR]

Step 2 : Repeat steps 3 and 4 while PTR ≠ NULL

Step 3 : Apply PROCESS to INFO[PTR]

Step 4 : Set PTR := LINK[PTR]
[PTR now points to the next node]
[End of step 2 loop.]

Step 5 : Exit

Q) Write a program in C to create and print the linked list to demonstrate traversing a Linked list.

```
/* Program to create and print the linked
   list */
#include<stdio.h>
#include<conio.h>
struct NODE
{
    int INFO;
    struct NODE *LINK;
};
void display(struct NODE *PTR);

void main()
{
    int i;
    struct NODE LIST[5];
    clrscr();
    printf("\n Enter 5 integer numbers \n");
    for(i=0;i<5;i++)
    {
        scanf("%d", &LIST[i].INFO);
        LIST[i].LINK = &LIST[i+1];
    }
    LIST[4].LINK = 0;
    printf("\n Linked list is \n");
    display(&LIST[0]);
    getch();
}

void display(struct NODE *PTR)
{
    while(PTR !=0)
    {
        printf("\n %u \t %d \t %u",
                &PTR -> INFO,
                PTR -> INFO,
                PTR -> LINK);
        PTR = PTR -> LINK;
    }
}
```

Output

```
Enter 5 integer numbers
55
33
66
22
11

Linked list is

65484    55    65488
65488    33    65492
65492    66    65496
65496    22    65500
65500    11    0
```

Searching a Linked List

Let LIST be a Linked list in memory. Suppose a specific ELE of information is given.

Searching a linked list refers to finding the location LOC of the node where ELE first appears in LIST.

If ELE is actually a key value and we are searching through a file for the record containing ELE, then ELE can appear only once in LIST.

The data in the list is either unsorted or sorted hence there are two different algorithms for searching a linked list. i.e.

1) LIST is unsorted
2) LIST is sorted

> LIST is unsorted

- Let LIST be a linked list in memory.
- Suppose the data in LIST are not necessarily sorted.
- Then searches for ELE in LIST by traversing through the list using a pointer variable PTR and comparing element ELE with the contents INFO[PTR] of each node, one by one of LIST.
- Before we update the pointer PTR by

PTR := LINK[PTR]

- we require two tests.

i) First we have to check to see whether we have reached the end of the list i.e. First we check to see whether

PTR = NULL

ii) If not, then we check to see whether

INFO[PTR] = ELE

- The two tests cannot be performed at the same time, since INFO[PTR] is not defined when PTR = NULL.
- Hence first test i.e. PTR = NULL use to control the execution of a loop.
- and second test i.e INFO[PTR] = ELE take place inside the loop.
- The algorithm for searching linked list when LIST is unsorted is as follows.

Algorithm : Searching unsorted LIST

SEARCH(INFO, LINK, START, ELE, LOC)

LIST is a linked list in memory. This algorithm finds the location LOC of the node where ELE first appears in LIST, or sets LOC = NULL

Step 1 : Set PTR := START
Step 2 : Repeat Step 3 while PTR ≠ NULL :

Step 3: If ELE = INFO[PTR], then:
Set LOC := PTR and Exit.
Else:
Set PTR := LINK[PTR].
[PTR now points to the next node.]
[End of If structure.]
[End of Step 2 loop.]

Step 4: [Search is unsuccessful.]
Set LOC := NULL

Step 5: Exit.

Complexity

- The complexity of this algorithm is that,
i) the worst-case running time is proportional to the number n of elements in LIST,
ii) The average-case running time is approximately proportional to n/2 [with the condition that ELE appears once in LIST but with equal probability in any node of LIST.]

Insertion into a Linked list

- Let LIST be a linked list with successive nodes A and B, as shown in following fig (a).

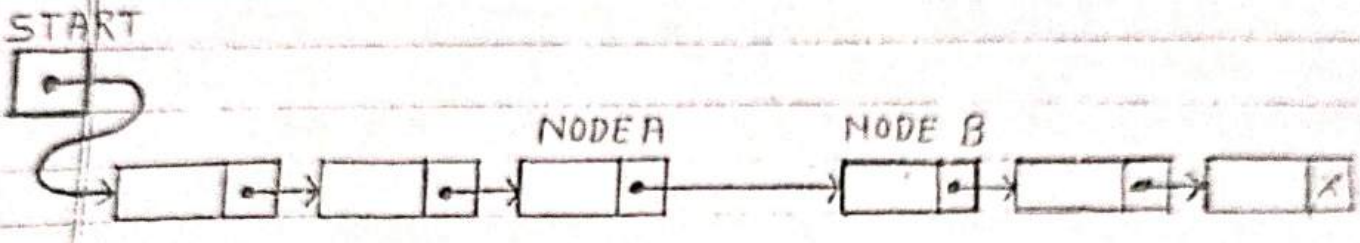


Fig (a). Before insertion

- Suppose a node N is to be inserted into the list between A and B. then as shown in following fig (b). NODE A now points to the new NODE N, and NODE N points to NODE B.

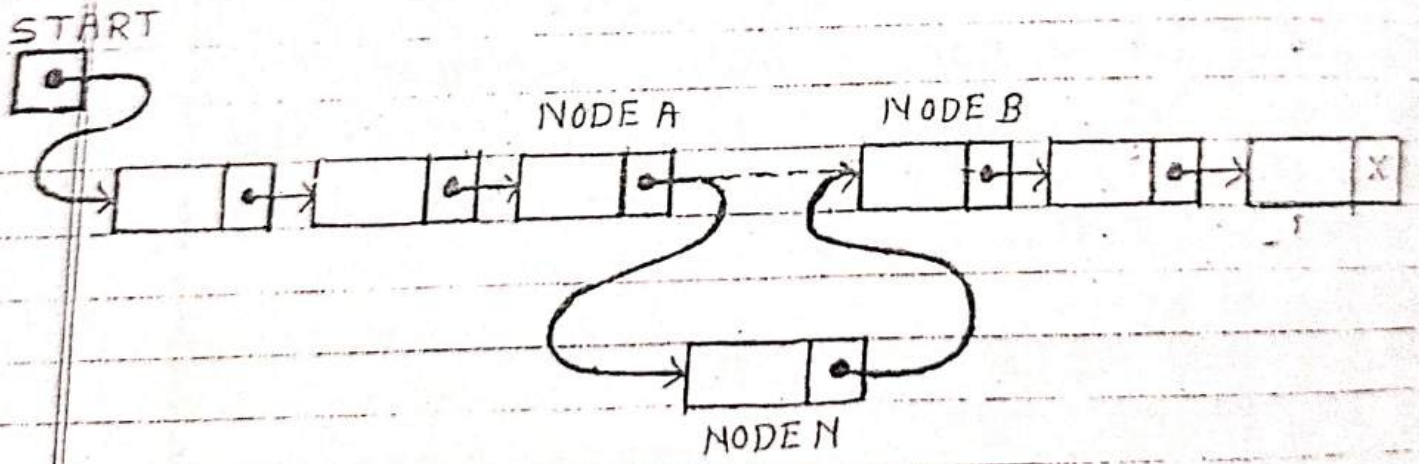
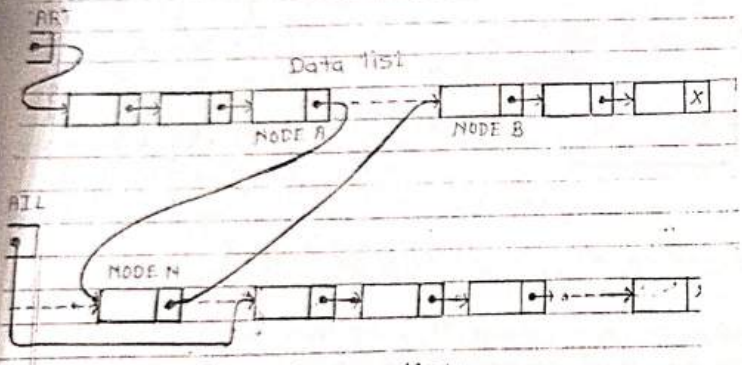


Fig (b) After insertion

- Suppose Linked list maintained in memory in the form

LIST (INFO, LINK, START, AVAIL)

where AVAIL is the pointer variable points to the LIST of available nodes.
- The above fig(b) does not specify the memory space for the new NODE N will come from the AVAIL list.
- For easier processing, the first node in the AVAIL list will be used for the new NODE N.
- Thus a more exact schematic diagram of such an insertion is shown in following fig(c)



Free-storage list

fig (c).

- Here three activities are performed
i) The nextpointer field of NODE A now points to the new NODE N, to which AVAIL previously pointed.

ii) AVAIL now points to the second node in the free pool, to which NODE N previously pointed.

iii) The nextpointer field of NODE N now points to NODE B, to which NODE A previously pointed.

- There are two special cases.
i) If the new NODE N is the first node in the list, then START will point to N.

ii) If the new NODE N is the last node in the list then N will contain the null pointer.

Insertion Algorithms

There are three situations to insert nodes to linked list that corresponds to the position to insert nodes.

I) Insert a node at the beginning of the list.

II) Insert a node after the node with a given location.

III) Insert node into a sorted list.

- These are three algorithms that assure the linked list in the memory in the form LIST(INFO, LINK, START, AVAIL) and that the variable ELE contains the new information to be added to the list.
- All these algorithms use node in AVAIL list. All of the algorithms will include the following steps.

Step 1 : Checking to see if space is available in the AVAIL list. If not i.e. if AVAIL = NULL, then the algorithm will print the message OVERFLOW.

Step 2 : Removing the first node from the AVAIL list, using the variable NEW to keep track of the location of the new node.

this step can be implemented by the pair of assignment means

NEW := AVAIL and

AVAIL := LINK[AVAIL]

Step 3 : Copying new information into the new node. that means

INFO[NEW] := ELE

- This process is shown in following fig(d)

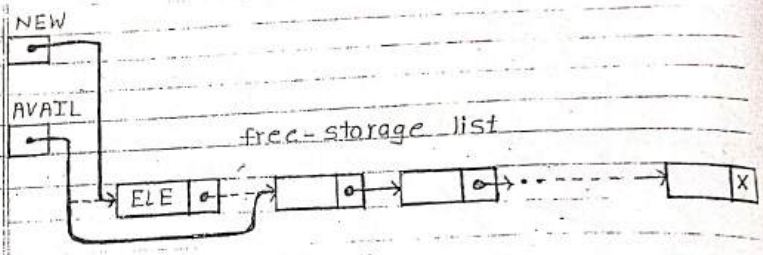


Fig.(d)

I) Inserting a node at the beginning of a list

- Suppose our linked list is not necessarily sorted and there is no reason to insert a new node in any special place in the list
- The the easiest place to insert the node is at the beginning of the list.
- Algorithm for inserting a node at the beginning of a list is as follow.

Algorithm 1: [Inserting a node at beginning

INSFIRST (INFO, LINK, START, AVAIL, ELE)

This algorithm inserts ELE as the first node in the list.

Step 1 : [OVERFLOW ?]
If AVAIL = NULL then
Write : OVERFLOW , and Exit

Step 2 : [Remove first node from AVAIL list].
Set NEW := AVAIL and
AVAIL := LINK[AVAIL]

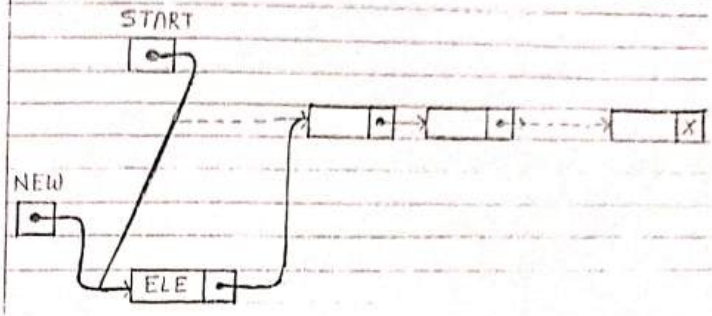
Step 3 : Set INFO[NEW] := ELE
[Copies new data into new node.]

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 the new node]

Step 6 : Exit.

- Following fig(e) shows steps 4 and 5 which are use to insert node at the beginning of the linked list.



Fig(e) Insertion a node at the beginning of a list

II) Insertion a node after a given node

- Suppose we are given the value of LOC where either LOC is the location of a NODE A in a linked list LIST or LOC = NULL.
- The following is an algorithm which inserts ELE into LIST. So that ELE so that ELE follows NODE A or when LOC = NULL so that ELE is the first node.
- Let N denote new node [whose location is NEW].
If LOC = NULL then N is inserted as the first node in LIST.
otherwise NODE N point to NODE B by the assignment
LINK[NEW] := LINK[LOC]
and NODE A point to the new NODE N.
LINK[LOC] := NEW

Algorithm 2 [Inserting a node after a given node]

INSLOC(INFO, LINK, START, AVAIL, LOC, ELE)

This algorithm inserts ELE so that ELE follows the node with location LOC or inserts ELE as the first node when LOC = NULL

Step 1 : [OVERFLOW ?]
If AVAIL = NULL then
Write : OVERFLOW, and Exit.

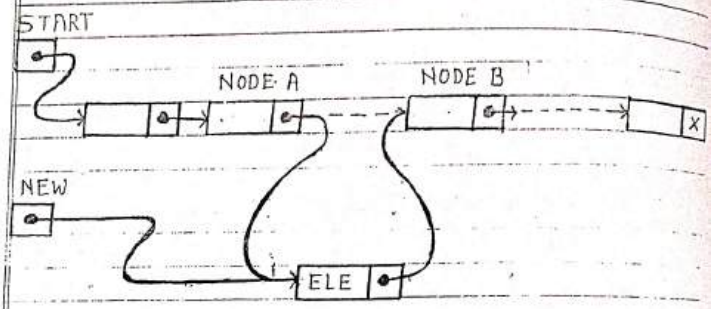
Step 2 : [Remove first node from AVAIL list]
Set NEW := AVAIL and
AVAIL := LINK[AVAIL]

Step 3 : Set INFO[NEW] := ELE
[Copies new data into new node.]

Step 4 : If LOC = NULL then
[Insert as first node]
Set LINK[NEW] := START and
START := NEW
Else [Insert after node with location LOC]
Set LINK[NEW] := LINK[LOC]
and LINK[LOC] := NEW
[End of If structure.]

Step 5 : Exit

- Following Fig(f) shows how to insert node after a given node



Fig(f). Insert a new node after NODE A where LOC = 3

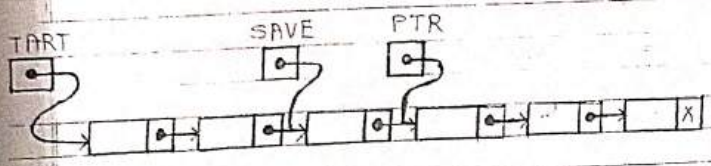
III) Inserting a node into a sorted list

- Suppose ELE is to be inserted into a sorted linked list LIST.
- Then ELE must be inserted between nodes A and B so that
INFO(A) < ELE ≤ INFO(B)

- The procedure finds the location LOC of NODE A, i.e. which finds the location LOC of the last node in LIST whose value is less than ELE.
- Traverse the list, using a pointer variable PTR and comparing ELE with INFO[PTR] at each node.

- While traversing keep track of the location of the preceding node by using a pointer variable SAVE as shown in following Fig(g) where

SAVE := PTR and
PTR := LINK[PTR]



Fig(g). SAVE := PTR and PTR := LINK[PTR]

- The traversing continues as long as INFO[PTR] > ELE or ITEM ≤ INFO[PTR]

- Then PTR points to NODE B so SAVE will contain the location of the NODE A.
- The procedure follows formal statement. The cases where the list is empty or where ITEM < INFO[START] so LOC = NULL are treated seperately, since they do not involve the variable SAVE

Procedure (9)

FINDA(INFO, LINK, START, ELE, LOC)

This procedure finds the location LOC of the last node in a sorted list such that INFO[LOC] < ELE or set LOC = NULL

Step 1: [List empty]
If START = NULL then:
Set LOC := NULL and Return.

Step 2: [Special case?]
If ELE < INFO[START] then:
Set LOC := NULL and return.

Step 3: Set SAVE := START and
PTR := LINK[START]
[Initialize pointers.]

Step 4: Repeat steps 5 and 6
while PTR ≠ NULL

Step 5: If ELE < INFO[PTR] then
Set LOC := SAVE and Return.
[End of If structure]

Step 6: Set SAVE := PTR and
PTR := LINK[PTR]

[Update pointers]
[End of step 4 loop]

Step 7 : Set LOC := SAVE

Step 8 : Return

Algorithm 3 [Insert a node into sorted list]

INSERT (INFO, LINK, START, AVAIL, ELE, LOC)
This algorithm inserts ELE into a sorted linked list.

Step 1 : [Use above procedure (a) to find the location of the node preceding ELE]
Call FINDA (INFO, LINK, START, ELE, LOC)

Step 2 : [OVERFLOW ?]
If AVAIL = NULL then
Write : OVERFLOW, and Exit.

Step 3 : [Remove find node from AVAIL list]
Set NEW := AVAIL and
AVAIL := LINK[AVAIL]

Step 4 : Set INFO[NEW] := ELE
[Copies new data into new node]

Step 5 : If LOC = NULL then
[Insert as first node]
Set LINK[NEW] := START and
START := NEW
Else [Insert after node with location LOC]
Set LINK[NEW] := LINK[LOC]
and LINK[LOC] := NEW
[End of IF structure]

Step 6 : Exit

i) The nextpointer field of node A now points to node B, where node N previously pointed.
ii) The nextpointer field of N now points to the original first node in the free pool, where AVAIL previously pointed.
iii) AVAIL now points to the deleted node N.

- There are two special cases.
i) If the deleted node N is the first node in the list, then START will point to node B.
ii) If the deleted node N is the last node in the list, then node A will contain the NULL pointer.

Deletion Algorithms

- There are two situations to delete node from a linked list.
i) Deletes the node following a given node.
ii) Deletes the node with a given ITEM of information.

- Suppose the linked list is in memory in the form
LIST(INFO, LINK, START, AVAIL).
- All deletion algorithms will return the memory space of the deleted node N to the beginning of the AVAIL list.
- All deletion algorithms use following assignment, where LOC is the location of the deleted node N.
LINK[LOC] := AVAIL and then
AVAIL = LOC
this shows in following fig.(d)

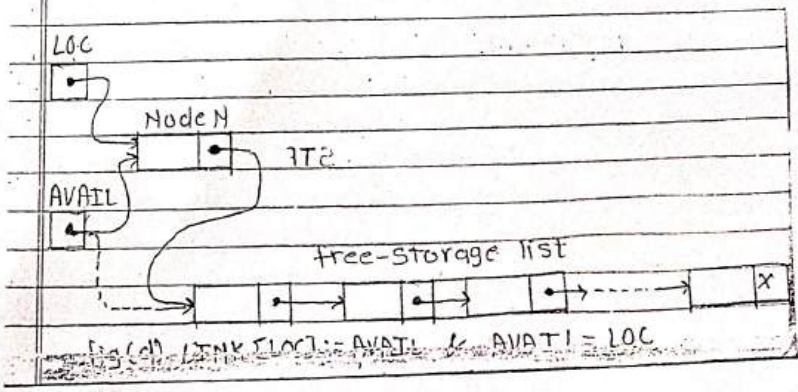


Fig.(d) LINK[LOC] := AVAIL & AVAIL = LOC

- Some algorithms delete either the first node or the last node from the list.
- An algorithm check if there is a node in the list.
  If not i.e. if START = NULL then the algorithm will print the message UNDERFLOW.

ii) Deleting the Node Following a Given Node

- Let LIST be a linked list in memory.
- Suppose we are given the location LOC of a node N in LIST.
- Suppose we are given the location LOCP of the node preceding N or, when N is the first node, we are given LOCP = NULL
- Following algorithm deletes N from the list.

Algorithm 1: [Deleting the Node Following a given node]

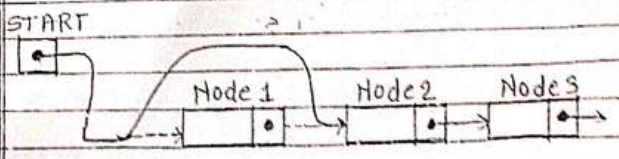
DEL(INFO, LINK, START, AVAIL, LOC, LOCP)
This algorithm deletes the node N with location LOC. LOCP is the location of the node which precedes N or, when N is the first node. LOCP = NULL.

Step 1: If LOCP = NULL then
  Set START := LINK[START]
  [Deletes first node]
  Else:
  Set LINK[LOCP] := LINK[LOC]
  [Deletes node N]
  [End of If structure.]

Step 2: [Return deleted node to the AVAIL list]
  Set LINK[LOC] := AVAIL and AVAIL := LOC.

Step 3: Exit.

- Following fig(c) shows diagram of assignment
  START := LINK[START]
which effectively deletes the first node from the list. This covers the case when N is the first node.



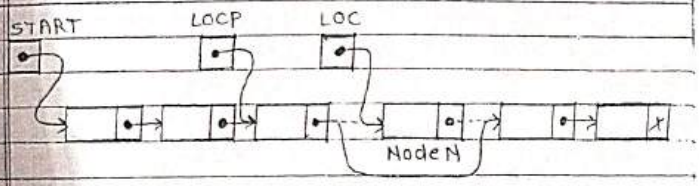
Fig(c). START := LINK[START]

Date: / / Page No.

- following fig(f) shows diagramatic representation of the assignment.

LINK[LOCP] := LINK[LOC]

which effectively deletes the node N when N is not the first node.



fig(f) LINK[LOCP] := LINK[LOC]

- The simplicity of the algorithm comes from the fact that we are already given the location LOCP of the node which precedes node N. we first find LOCP.

ii) Deleting the Node with a Given ITEM of Information.

- Let LIST be a linked list in memory.
- Suppose we are given an ITEM of information and we want to delete from the LIST the first node N which contains ITEM.
- First we use a procedure which finds the location LOC of the node N containing ITEM and location LOCP of the node preceding node N.
- If N is the first node, we set LOCP=NULL and if ITEM does not appear in LIST, we set LOC=NULL.
- Traverse the list, using a pointer variable PTR and comparing ITEM with INFO[PTR] at each node.
- While traversing, keep track of the location of the preceding node by using a pointer variable SAVE, as shown in following fig(g). Thus SAVE and PTR are updated by the assignments.

SAVE := PTR and

PTR := LINK[PTR]

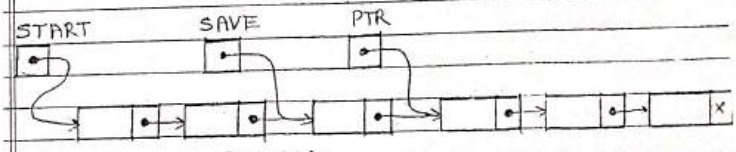


fig.(g).

- The traversing continues as long as INFO[PTR] ≠ ITEM, or traversing stops as soon as ITEM = INFO[PTR].
- Then PTR contains the location LOC of node N and SAVE contains the location LOCP of the node preceding N.
- The case where the list is empty or where INFO[START] = ITEM are treated separately, since they do not involve the variable SAVE.

M T W T F S

Date: / / Page No.

* Types of Linked List

1) Header Linked List

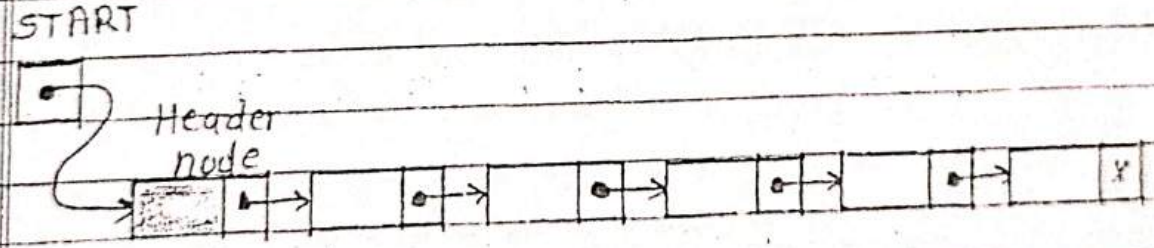
A header linked list is a linked list which always contains a special node, called the header node, at the beginning of the list.

The following are two kinds of widely used header list.

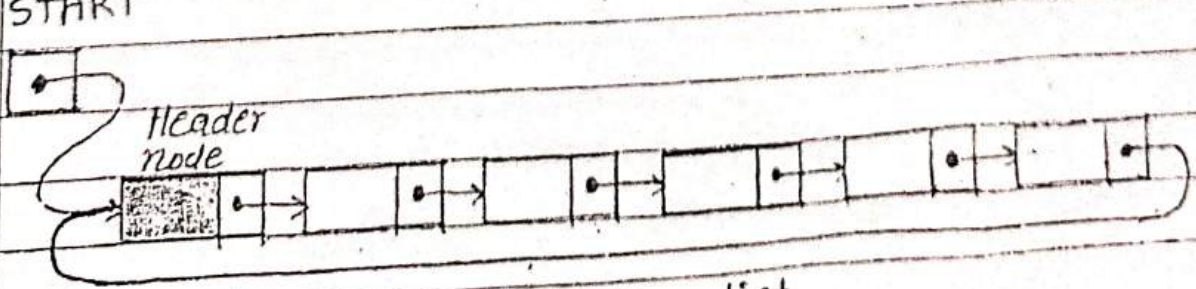
i) A grounded header list is a header list where the last node contains the null pointer.

ii) A circular header list is a header list where the last node points back to the header node.

Following fig. shows these two types of header list.



(a) Grounded header list



(b) Circular header list

- The list pointer START always points to the header node. Accordingly, LINK[START] = NULL indicates that a grounded header list is empty, and LINK[START] = START indicates that a circular header list is empty.

Linked List with header and trailer nodes

A linked list contains both a special node at the beginning of the list and a special trailer node at the end of the list.

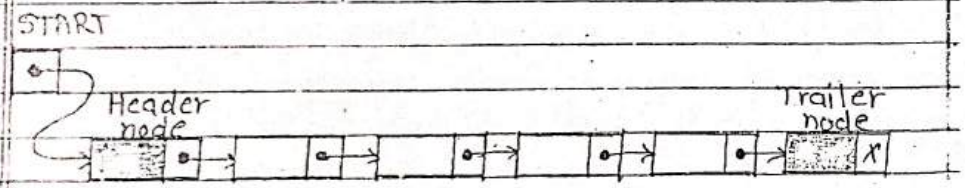


Fig. Linked List with header and trailer nodes

2) Two-way Lists or doubly Linked List-

- Two-way list traversed in two directions:
  i) In the usual forward direction from the beginning of the list to the end.
  ii) or in the backward direction from the end of the list to the beginning.

- A two-way list is a linear collection of data elements, called nodes, where each node N is divided into 3 parts.
  i) An information field INFO which contains the data of N.
  ii) A pointer field FORW which contains the location of the next node in the list.
  iii) A pointer field BACK which contains the location of the preceding node in the list.

- The list also requires two list pointer variables.
  i) FIRST, which points to the first node in the list.
  ii) LAST, which points to the last node in the list.

- Following figure shows two-way list.

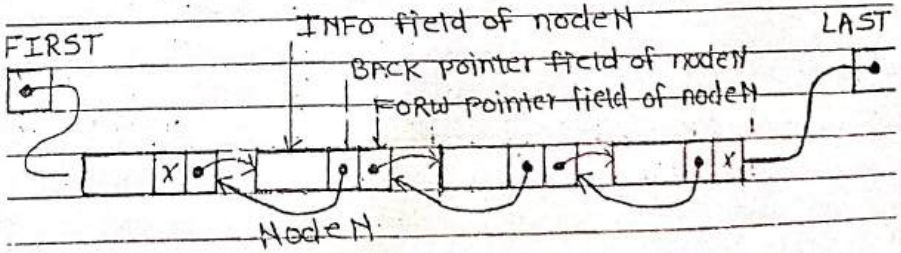


Fig. Two-way list

- The null pointer appears in the FORW field of the last node in the list and also

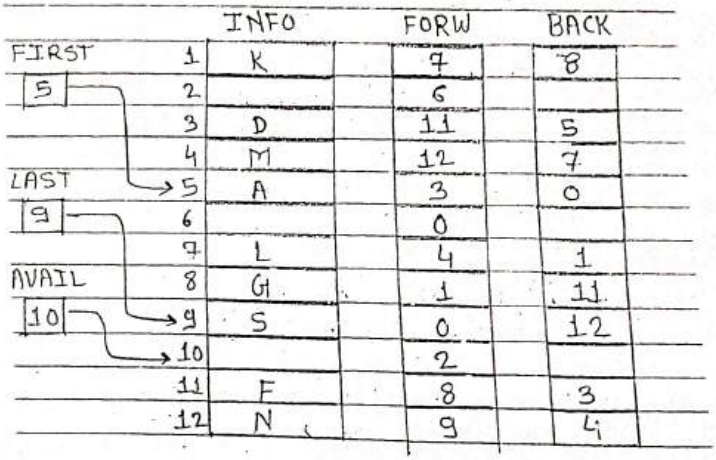
in the BACK field of the first node in the list.
- Using the variable FIRST and the pointer field FORW, we can traverse a two-way list in the forward direction.
- Using the variable LAST and the pointer field BACK, we can traverse the list in the backward direction.
- Suppose LOCA and LOCB are the locations of nodes A and B respectively in a two-way list. Then the way that the pointers FORW and BACK are defined given as follow.

Pointer property :
FORW[LOCA] = LOCB if and only if
BACK[LOCB] = LOCA
means
Statement node B follows nodeA is equivalent to the statement that node A precedes node B

Representation of two-way linked list in memory

- Two-way list may be maintained in memory by means of linear arrays.
- It require two pointer arrays, FORW and BACK.
- It require two list pointer variables FIRST and LAST.
- The AVAIL of available space in the arrays will still be maintained.
- Using FORW as the pointer field since we delete and insert nodes only at the beginning of the AVAIL list.

- *Example*



- Above fig. shows the representation of two-way list in memory which contain two arrays FORW and BACK
where FIRST = 5 LAST = 9 AVAIL = 10
It will create two strings
In forward direction = A D F G K L M N S
In backward direction = S N M L K G F D A

Date: / /

Operations on Two-way Lists

Suppose LIST is a Two-way list in memory. Following operations can perform on two-way list.

1) Traversing
2) Searching
3) Inserting
4) Deleting

I) Traversing

Suppose we want to traverse Two-way list in order to process each node exactly once then we can use following algorithm.

Algorithm : [Traversing a Two-way linked list]
Let LIST be a two-way linked list in memory. This algorithm traverses LIST in forward direction by applying an operation PROCESS to each element of LIST.
The variable PTR points to

Step1 : Set PTR := FIRST
[Initialize pointer PTR]

Page No. / /

Step 2 : Repeat steps 3 and 4 while PTR ≠ NULL

Step 3 : Apply PROCESS to INFO[PTR].

Step 4 : Set PTR := LINK[PTR]
[PTR now points to the next node]
[End of step2 loop]

Step 5 : Exit

- If we want to traverse the Two-way list in backward direction then in above algorithm only change the step 1. ie.
step 1 : set PTR := BACK
[Initialize pointer PTR as BACK

III) Inserting

- Suppose we are given the locations LOCA and LOCB of nodes A and B respectively in LIST.
- Suppose we want to insert a given ITEM of information between nodes A and B.
- Remove the first node N from the AVAIL list, using the variable NEW to keep track

of its location, and then we copy the data ITEM into the node N, i.e. we set

NEW := AVAIL
AVAIL := FORW[AVAIL]
INFO[NEW] := ITEM

-- The node N with contents ITEM is inserted into the list by changing the four pointers

FORW[LOCA] := NEW
FORW[NEW] := LOCB
BACK[LOCB] := NEW
BACK[NEW] := LOCA

shown in following fig.

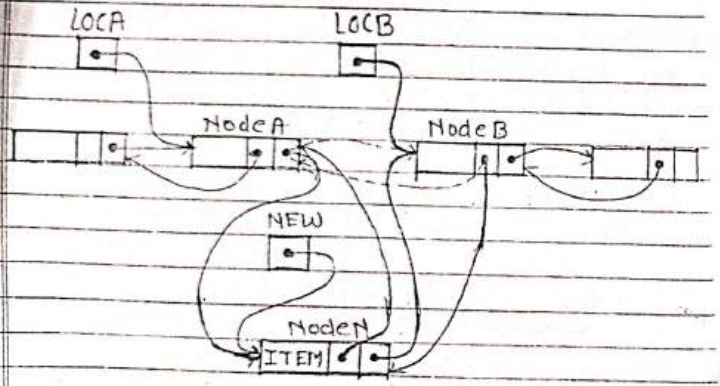


Fig. Inserting node N in two-way linked list.

Algorithm : [Insert a node N into Two-way linked list]

INSTWL (INFO, FORW, BACK, START, AVAIL, LOCA, LOCB, ITEM)

This algorithm will insert node N between node A and node B of two-way linked list

Step 1 : [OVERFLOW ?]
If AVAIL = NULL then
write : OVERFLOW , and Exit

Step 2 : [Remove node from AVAIL list and copy new data into node]
Set NEW := AVAIL
AVAIL := FORW[AVAIL]
INFO[NEW] := ITEM

Step 3 : [Insert node into list]
Set FORW[LOCA] := NEW
FORW[NEW] := LOCB
BACK[LOCB] := NEW
BACK[NEW] := LOCA

Step 4 : Exit

Deleting

- Suppose we are given the location LOC of a node N in LIST.
- Suppose we want to delete N from the list.
- Deleting of node N from two-way list is shown in following figure.

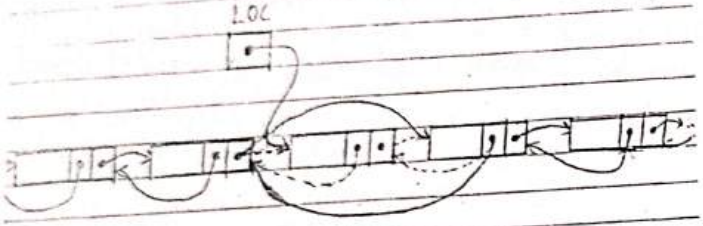


Fig. Deleting Node N

- BACK[LOC] is the location of the node which precede node N.
- FORW[LOC] is the location of the node which follow node N.
- Above fig. shows node N is deleted from the list by changing the following pair of pointers.

FORW[BACK[LOC]] := FORW[LOC]
and
BACK[FORW[LOC]] := BACK[LOC]

- The deleted node N is then returned to the AVAIL list by the assignments

FORW[LOC] := AVAIL and
AVAIL := LOC

Algorithm : [Delete a node from Two-way Linked List]
DELTWL(INFO, FORW, BACK, START, AVAIL, LOC)

Step 1 : [Delete node]
Set FORW[BACK[LOC]] := FORW[LOC]
and
BACK[FORW[LOC]] := BACK[LOC].

Step 2 : [Return node to AVAIL list]
Set FORW[LOC] := AVAIL and
AVAIL := LOC

Step 3 : Exit

Two-way Header Linked List

- The advantage of two-way list and a circular header list may be combined into a circular header list as shown in following fig.

Date: / / Page No.

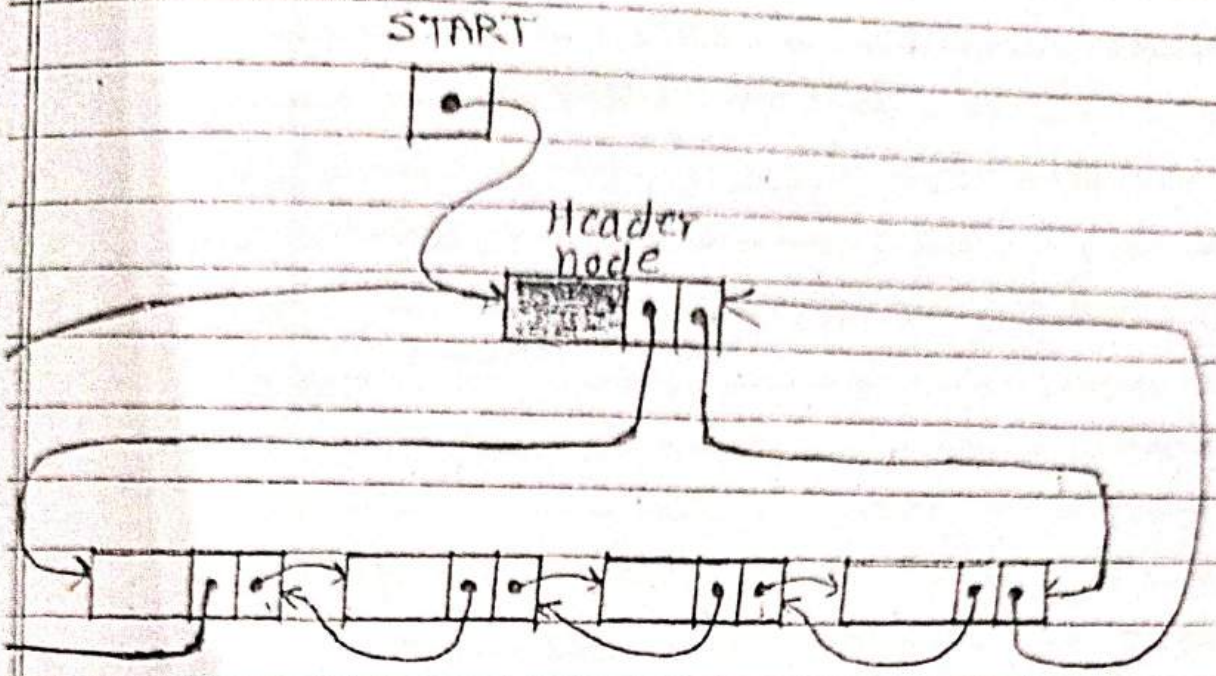


Fig. Two-way Circular Heade list.

- Two-way header linked list requires only one list pointer variable START, which points to the header node. This is because the two pointers in the header node point to the two ends of the list.

- This list is circular because the two end nodes point back to the header node.

UNIT I, II - Chapter 2

Stacks and Queues

In linear lists and linear arrays one can insert and delete elements at any place in the list - i.e. at the beginning, at the end or in the middle.

Stacks and Queues are use to restrict insertions and deletions so that one can insert and delete elements at the beginning or the end of the list, not in the middle.

* Stacks

- A stack is a linear structure in which items are added or removed only at one end
- The last item to be added to a stack is the first item to be removed.
- stack are also called last-in-first-out [LIFO] lists.
- Stack is also called as "piles" and "push-down" lists.

- A stack is a list of elements in which an element may be inserted or deleted only at one end, called the top of the stack.
- The elements are removed from a stack in the reverse order of that in which they were inserted into the stack.

- special terminology is used for two basic operations associated with stacks.
1) "Push" is the term used to insert an element into a stack.
2) "Pop" is the term used to delete an element from a stack.

- Suppose the following 6 elements are pushed, in order, onto an empty stack.
STACK: AAA, BBB, CCC, DDD, EEE, FFF
Following fig. shows 3 ways of picturing a stack

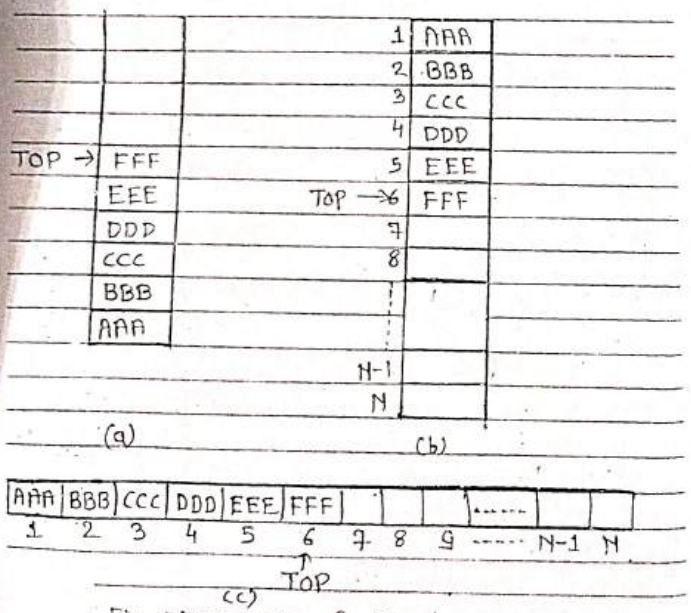
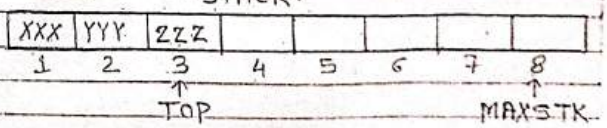


Fig. Diagrams of Stack

- The implication is that the right-most element is the top element.
- The insertion and deletion can occur only at the top of the stack. This means EEE cannot be deleted before FFF is deleted, DDD cannot be deleted before EEE and FFF are deleted.

* Array Representation of stacks

- Stacks are represented in the computers memory by means of linear array.
- Stacks will be maintained by a linear array STACK.
- A pointer variable TOP, which contains the location of the top element of the stack.
- A variable MAXSTK gives the maximum number of elements that can be held by the stack.
- The condition TOP=0 and TOP = NULL will indicate that the stack is empty.
- Following fig shows array representation of stack.



- Since TOP=3, the stack has 3 elements XXX, YYY and ZZZ, and MAXSTK=8, there is room for 5 more items in the stack.
- The operation of adding an item onto a stack is called PUSH.
- In executing the procedure PUSH, one must first test whether there is room in the stack for the new item, if not then we have the condition known as OVERFLOW.
- The operation of deleting an item from a stack is called POP.
- In executing the procedure POP, one must first test whether there is an element in the stack to be deleted, if not, then we have the condition known as UNDERFLOW.
- Following is the procedure for PUSH operation in which after inserting an item into stack TOP will increase by 1.

TOP = TOP + 1

Procedure : [Push operation]

PUSH(STACK, TOP, MAXSTK, ITEM)
This procedure pushes an ITEM onto a stack.

Step 1: [Stack already filled?]
If TOP = MAXSTK then
Print: OVERFLOW and Return

Step 2: Set TOP := TOP + 1
[Increase TOP by 1]

Step 3: Set STACK[TOP] := ITEM
[Inserts ITEM in new TOP position]

Step 5: Return.

Following is the procedure for POP operation in which after deleting an item from stack TOP will decrease by 1.

TOP = TOP - 1

Procedure : (POP operation)

POP(STACK, TOP, ITEM)
This procedure deletes the top element of STACK and assigns it to the variable ITEM.

Step 1: [Stack has an item to be removed?]
If TOP = 0 then
Print: UNDERFLOW and Return

Step 2: Set ITEM := STACK[TOP]
[Assigns TOP element to ITEM]

Step 3: Set TOP := TOP - 1
[Decreases TOP by 1]

Step 4: Return.

Program to demonstrate PUSH and POP operations on stack.

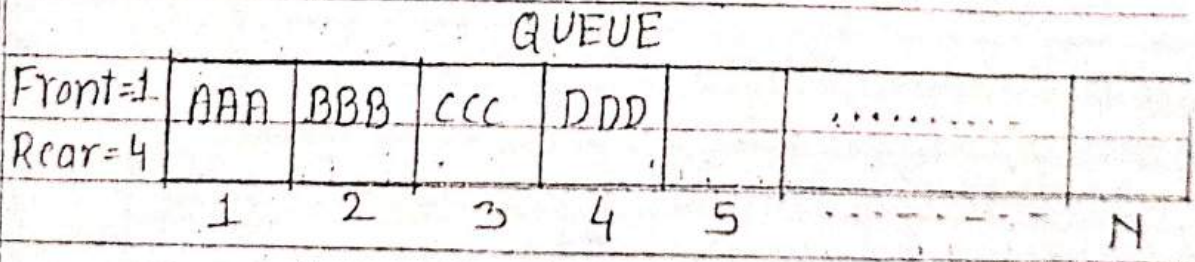
```
/* Program to demonstrate PUSH and POP
   operations on stack */
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#define MAXSIZE 5
void PUSH();
int POP();
void display();
int stack[MAXSIZE];
int TOP = -1;

void main()
{
    int choice, I;
    char c;
    clrscr();
    do
    {
        printf("\n\n Menu:");
        printf("\n 1. PUSH");
        printf("\n 2. POP");
        printf("\n 3. Display");
        printf("\n 4. Exit");
        scanf("\n %d", &choice);
        switch(choice)
        {
            case 1:
                PUSH();
                break;

            case 2:
                printf("\n The
                deleted element is
                %d", I);
                I = POP();
                break;

            Case 3:
                display();
                break;
            case 4:
                printf("\n Exit");
                exit(0);
                break;
        }
    }while(choice!=4);
```

Date: / / Page No.

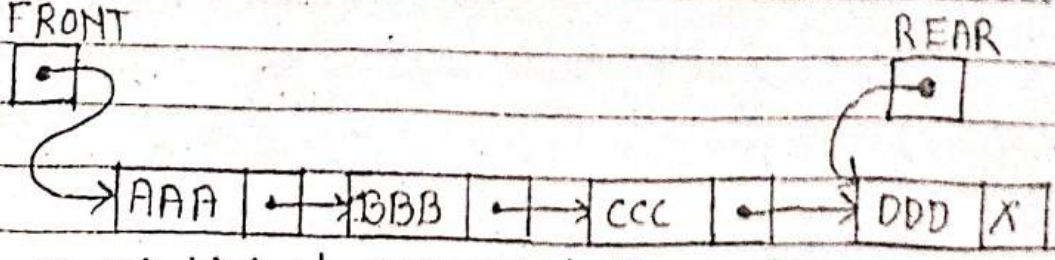
```
void PUSH():
{
    int item;
    if(TOP == MAXSIZE-1)
    {
        printf("\n The Stack is Full\n
                OVERFLOW ");
        getch();
        return;
    }
    else
    {
        printf("\n Enter the element
                to be inserted = ");
        scanf("%d", &item);
        TOP = TOP+1;
        stack[TOP] = item;
    }
}

int POP()
{
    int item;
    if (TOP == -1)
    {
        printf("\n The stack is
                empty i.e. UNDERFLOW");
        getch();
        return;
    }
    else
    {
        item = stack[TOP];
        TOP = TOP-1;
        return(item);
    }
}

void display()
{
    int i;
    if(TOP == -1)
    {
        printf("\n The stack is Empty");
        getch();
        return;
    }
    else
    {
        printf("\n The Elements in stack
                are ");
        for(i=TOP; i>=0; i--)
        {
            printf("\n %d", stack[i]);
        }
    }
}
```

* QUEUES

- A queue is a linear list of elements in which deletions can take place only at one end, called the front, and insertion can take place only at the other end, called the rear.
- The terms "front" and "rear" are used in describing a linear list only when it is implemented as a queue.
- Queue are also called first-in-first-out (FIFO) lists, since the first element in a queue will be the first element out of the queue.
- The order in which elements enter a queue is the order in which they leave.
- Following fig shows the various representations of queue



Fig(a) Array Representation of queue

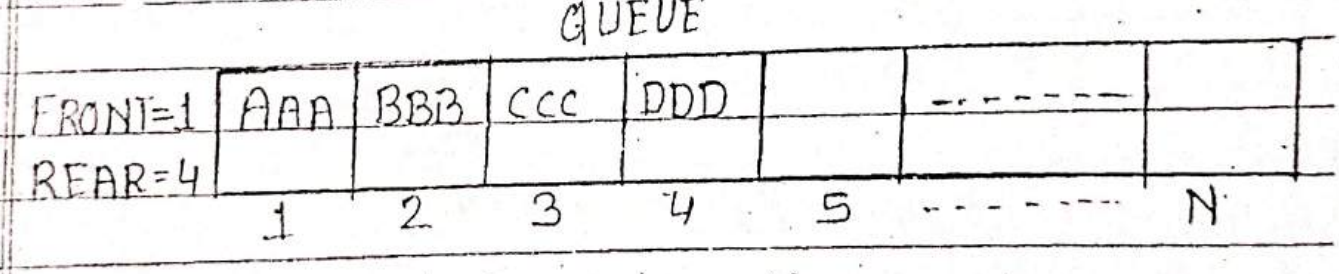


fig(a). Linked representation of queue.

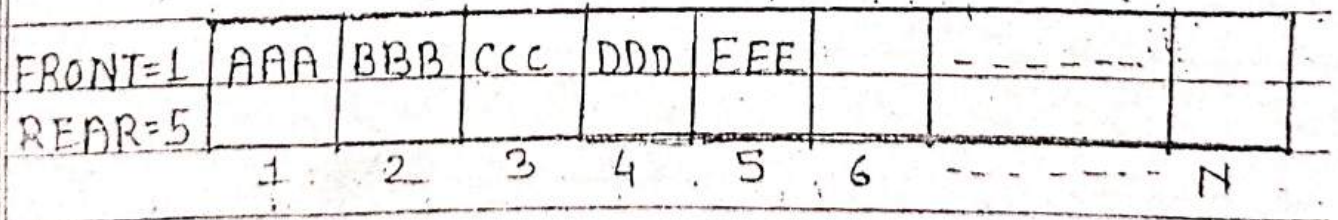
Date: / / Page No.

* Array Representation of Queues

- Queues are represented in the computers memory by means of linear arrays.
- Queues will be maintained by a linear array QUEUE, and two pointer variables FRONT and REAR.
- FRONT containing the location of the front element of the queue.
- REAR containing the location of the rear element of the queue.
- The condition FRONT = NULL will indicate that the queue is empty.
- Following fig. shows the way new elements will be added to the queue.



Queue before insertion.



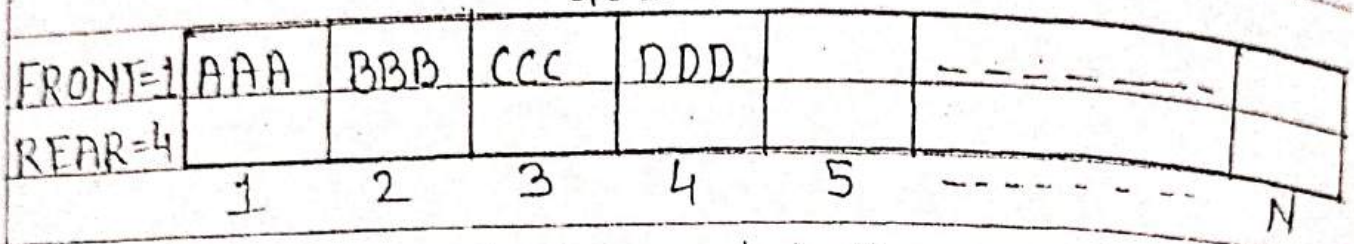
QUEUE after inserting EEE

- Whenever an element is added to the queue, the value of REAR is increased by 1, by assignment

REAR := REAR + 1

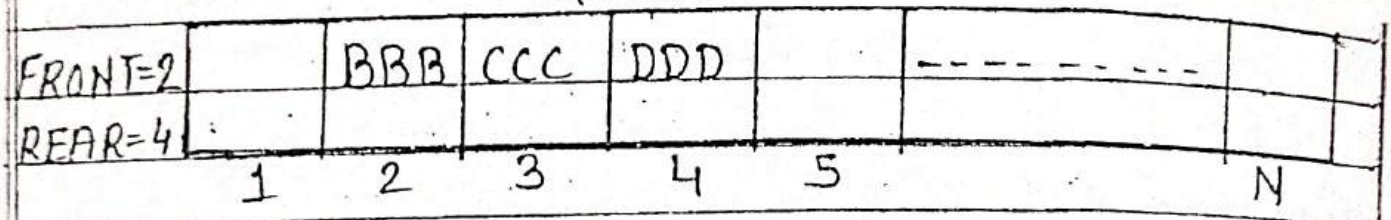
- Following fig. shows the way elements will be deleted from the queue.

QUEUE



QUEUE before deletion.

QUEUE



QUEUE after delition of AAA.

- Whenever an element is deleted from the queue, the value of FRONT is increased by 1, by assignment.

FRONT := FRONT + 1

- Circular Queue

In simple queue representation there are wastage of storage space or memory. If the FRONT pointer never manage to catchup to the REAR pointer.

Actually an arbitrary large amount of memory would be require to

accomodate the elements. This method of performing operation on a queue should be used when queue is empty at certain intervals.

This means after N insertions the REAR element of the queue will occupy QUEUE[N] or in the other words eventually the queue will occupy the last part of the array.]

- Suppose we want to insert an element ITEM into a queue at the time the queue does occupy the last part of the array, i.e. when REAR=N.
- One way to do this is to simply move the entire queue to the beginning of the array, changing FRONT and REAR accordingly, and then inserting ITEM.
- For this assume that the array QUEUE is circular, i.e. that QUEUE[1] comes after QUEUE[N] in the array.
- With this assumption, we insert ITEM into the queue by assigning ITEM to QUEUE[1].
- So instead of increasing REAR to N+1 we reset REAR=1 and then assign QUEUE[REAR] := ITEM
- Following Procedure shows insertion of an element in QUEUE.

Procedure : [Inserting an element into queue]

QINSERT(QUEUE, N, FRONT, REAR, ITEM)
This procedure inserts an element ITEM into a queue.

Step 1: [Queue already filled?]
If FRONT = 1 and REAR = N, or
If FRONT = REAR + 1 then
write: OVERFLOW and return.

Step 2: [Find new value of REAR]
If FRONT := NULL then
[Queue initially empty]
Set FRONT := 1 and
REAR := 1
Else if REAR = N, then:
Set REAR := 1
Else:
Set REAR := REAR + 1
[End of if structure.]

Step 3: Set QUEUE[REAR] := ITEM
[This inserts new element].

Step 4: Return

- If FRONT = N and an element of QUEUE is deleted, we reset FRONT = 1 instead of increasing FRONT to N+1.

Suppose that our queue contains only one element i.e. suppose that:
FRONT = REAR ≠ NULL

and suppose that the element is deleted. Then we assign:
FRONT := NULL and
REAR := NULL
To indicate that the queue is empty.
Following procedure shows deletion of an element from queue.

Procedure : [Deletion of an element from queue.
QDELETE (QUEUE, N, FRONT, REAR, ITEM)
This procedure deletes an element from a queue and assigns it to the variable ITEM.

Step 1 : [Queue already empty?]
If FRONT := NULL then
write : UNDERFLOW, and
Return.

Step 2 : Set ITEM := QUEUE[FRONT]

Step 3 : [Find new value of FRONT]
If FRONT = REAR then
[queue has only one element to start]
Set FRONT := NULL
and REAR := NULL
Else if FRONT = N, then:
Set FRONT := 1
Else
Set FRONT := FRONT + 1.
[End of If structure]

Step 4 : Return.

Program to demonstrate insertion and deletion operation on queue

```
/* Program for insertion and deletion
   operation on queue */
#include<stdio.h>
#include<conio.h>
#define MAXSIZE 5
int QUEUE[MAXSIZE];
int FRONT=-1, REAR=0;
int choice;
char ch;
Void QINSERT();
```

```
void QDELETE();
void display();

void main
{
    clrscr();
    do
    {
        printf("\n\n Menu : ");
        printf("\n 1.Insert \n 2.Delete
                \n 3.display \n 4.Exit");
        printf("\n Enter your choice = ");
        scanf("%d", &choice);
        switch(choice)
        {
            case 1 :
                    QINSERT();
                    break;
            case 2:
                    QDELETE();
                    break;
            case 3:
                    display()
                    break;
            case 4:
                    exit(0);
                    break;
        }
    }while(choice != 4);
}
```

```
Void QINSERT()
{
    int num;
    if(FRONT == (REAR+1) % MAXSIZE)
    {
        printf("\n Queue is full");
        return;
    }
    else
    {
        printf("\n Enter the element to
                be inserted : ");
        scanf("%d", &num);
        if(FRONT == -1)
            FRONT = REAR = 0;
        else
            REAR = (REAR+1) % MAXSIZE
            QUEUE[REAR] = num;
    }
}

void QDELETE()
{
    int num;
    if (FRONT == -1)
    {
        printf("\n Queue is Empty");
        return;
    }
    else
```

```
    {
        num = QUEUE[FRONT];
        printf("\n Deleted element
                is = %d", QUEUE[FRONT]);
        if (FRONT == REAR)
            FRONT = REAR = -1;
        else
            FRONT = (FRONT+1) %
                    MAXSIZE;
    }
}

void display()
{
    int i;
    if (FRONT == -1)
    {
        printf("\n Queue is empty");
        return;
    }
    else
    {
        printf("\n The elements of the
                queue ");
        for(i=FRONT; i<=REAR; i++)
        {
            printf("\n %d", QUEUE[i]);
        }
    }
    if (FRONT > REAR)
    {
        for(i=FRONT; i<MAXSIZE; i++)
        {
            printf("%d", QUEUE[i]);
        }
        for(i=0; i<=REAR; i++)
        {
            printf("%d", QUEUE[i]);
        }
    }
    printf("\n");
}
```

* Linked Representation of queues

- Queues are represented in the computers memory by means of one-way list or singly linked list.
- A linked queue is a queue implemented as a linked list with two pointer variables FRONT and REAR pointing to the nodes which is in the FRONT and REAR of the queue respectively.
- The INFO fields of the list hold the elements of the queue.
- The LINK fields hold pointers to the neighbouring elements in the queue.
- Following fig. shows linked representation of queue

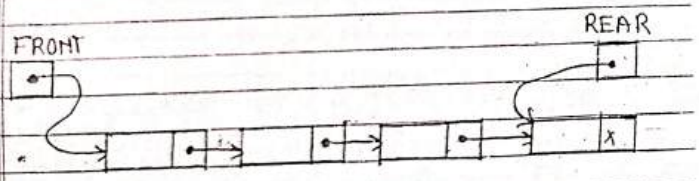


Fig. Linked Representation of Queue

- In the case of insertion into a linked queue, a node borrowed from the AVAIL list and carrying the item to be inserted is added as the last node of the linked list representing the queue.
- The REAR pointer is updated to point to the last node just added to the list.
- Following fig shows insertion of node in linked queue.

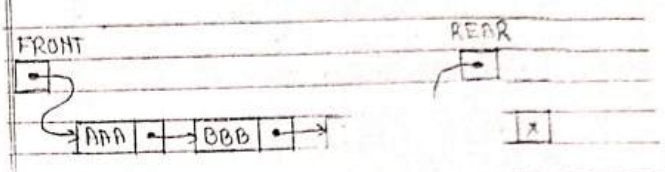


Fig. Queue before insertion

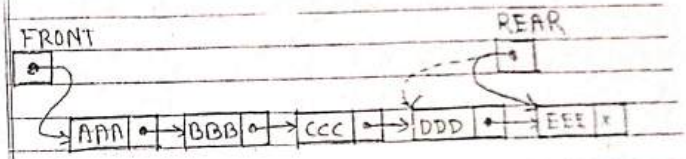


Fig. Linked Queue after inserting EEE

- The linked queue is not limited in capacity and therefore as many as the AVAIL list can provide may be inserted into the queue. This check for the OVERFLOW condition during insertion.
- Following procedure shows insertion of a node in linked queue.

Procedure: (Inserting a node into linked Queue)

LINKQ_INSERT(INFO, LINK, FRONT, REAR, AVAIL, ITEM)
This procedure inserts an ITEM into a linked queue

Step 1: [Available Space?]
If AVAIL = NULL then
write: OVERFLOW and Exit.

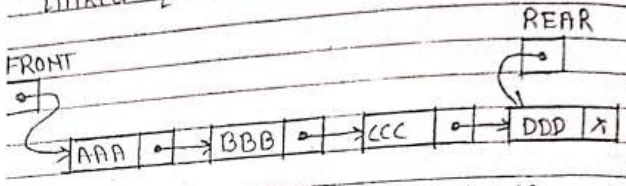
Step 2: [Remove first node from AVAIL list.]
Set NEW := AVAIL and
AVAIL = LINK[AVAIL]

Step 3: Set INFO[NEW] := ITEM and LINK[NEW] := NULL
[Copies ITEM into new node]

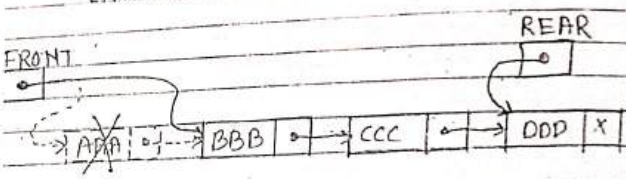
Step 4: If(FRONT = NULL) then
FRONT = REAR = NEW
[If queue is empty then ITEM is the first element in the queue]
else
Set LINK[REAR] := NEW and
REAR := NEW
[REAR points to the new node added to the end of the list]

Step 5: Exit

- In case of deletion, the first node of the list pointed to FRONT is deleted and the FRONT pointer is updated to point to the next node in the list.
- Following fig. shows deletion from a linked queue.



Linked queue before deletion



Linked queue after deleting node AAA

- Following procedure shows the deletion of a node from linked queue.

Procedure : (deleting a node from linked queue)

LINKQ_DELETE(INFO, LINK, FRONT, REAR, AVAIL, ITEM)

This procedure deletes the front element of the linked queue and stores it in ITEM.

Step 1: [Linked queue empty?]
if(FRONT = NULL) then
write : UNDERFLOW and Exit.

Step 2: set TEMP = FRONT
[If linked queue is nonempty, remember FRONT in a temporary variable. TEMP]

Step 3: ITEM = INFO(TEMP)

Step 4: FRONT = LINK(TEMP)
[Reset FRONT to point to the next element in the queue]

Step 5: LINK(TEMP) = AVAIL and
AVAIL = TEMP
[return the deleted node TEMP to the AVAIL list]

Step 6: Exit.

Program to demonstrate insertion and deletion operation on linked queue.

```
/* Program to demonstrate insertion and
   deletion operations on linked queue */
#include<stdio.h>
#include<conio.h>
#include<malloc.h>
struct node
{
    int info;
    struct node *next;
}*front, *rear;

void insert();
void display();
void delet();

void main()
{
    int ch;
    rear=NULL;
    clrscr();
    do
    {
        printf("\n\n Menu:");
        printf("\n 1. Insert \n 2. Delete
                \n 3. Display \n 4. Exit");
        printf("\n Enter your choice= ");
        scanf("%d", &ch);
        switch(ch)
        {
            case 1:
```

M T W T F S

Date: / / Page No.

* Types of Queues

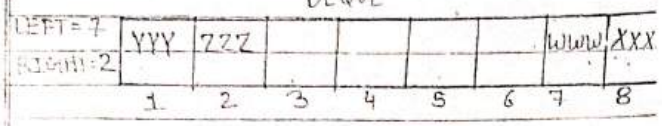
1) DEQUES [Double-ended Queue]

- A deque is a linear list in which elements can be added or removed at either end but not in the middle.
- The deque is a contraction of the name double-ended queue.
- There are various ways of representing deque in a computers memory
- Deque is maintained by a circular array DEQUE with pointer LEFT and RIGHT, which point to the two ends of the deque.
- We assume that the elements extend from the left end to the right end in the array.
- The term 'circular' comes from the fact that we assume that DEQUE[1] comes after DEQUE[N] in the array.
- The condition LEFT = NULL will be used to indicate that a deque is empty.
- Following fig. shows each with 4 elements maintained in an array with N = 8 memory locations.

DEQUE

LEFT=4				AAA	BBB	CCC	DDD	
RIGHT=7								
	1	2	3	4	5	6	7	8

DEQUE



- The procedure which insert and delete elements in deques are given as supplementary problems.
  i) When there is overflow, i.e. when an element is to be inserted into a deque which is already full.
  ii) When there is underflow, that is, when an element is to be deleted from a deque which is empty.

Priority Queues

- A priority queue is a collection of elements such that each element has been assigned a priority.
- There are two types of priority queue

1) Ascending priority queue

An ascending priority queue is a collection of items into which items can be inserted arbitrarily and from which only the smallest item can be removed.

2) Descending priority queue

A descending priority queue is a collection of items into which items can be inserted arbitrarily and from which only the largest item can be removed.

- The order in which elements are deleted and processed comes from the following rules.
  1) An element of higher priority is processed before any element of lower priority.
  2) Two elements with same priority are processed according to the order in which they were added to the queue.

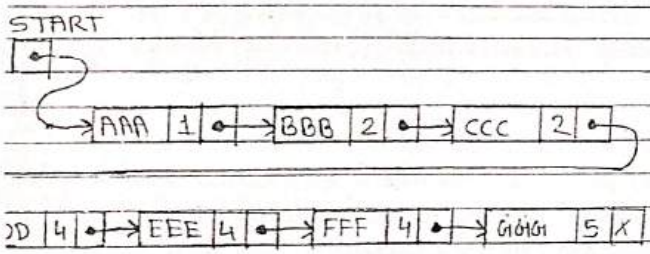
One-Way List Representation of Priority Queue

- Priority queue is maintain in computers memory by means of one-way list or singly linked list.
  1) Each node in the list will contain three items of information:
     An information field INFO,
     a priority number PRN and
     a link number LINK.
  2) A node X precedes a node Y in the list:
     i) When X has higher priority than Y or
     ii) When bothe have the same priority but X was added to the list before

Y. This means that the order in the one-way list corresponds to the order of the priority queue.

- Priority numbers will operate in the usual way.
  1) The lower priority number.
  2) The higher priority number.

- Following diagram shows priority queue with 7 elements.



Fig(a). Priority Queue with 7 elements.

- Following figure shows memory representation of priority queue shown in fig(a).

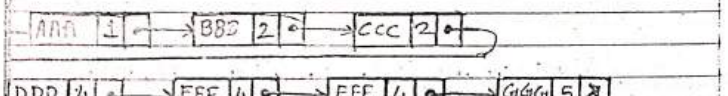
	INFO	PRN	LINK
1	BBB	2	6
2			7
3	DDD	4	4
START 4	EEE	4	9
5 →	AAA	1	1
6	CCC	2	3
AVAIL 7			10
2 → 8	GGG	5	0
9	FFF	4	8
10			11
11			12
12			0

Fig. Memory Representation of priority Queue

- Array representation of priority queue is as follow.

1	AAA
2	BBB
3	CCC
4	DDD
5	EEE
6	FFF
7	GGG
8	
9	
10	
11	

- One way to maintain a priority queue in memory is to use a seperate queue for each level of priority.
- Each such queue will appear in its own circular array and must have its own pair of pointers. FRONT and REAR.
- If each queue is allocated the same amount of space, a two-dimensional array QUEUE can be used instead of the linear arrays.
- Consider a following queue



- The array representation is as follow i.e Two-dimensional array QUEUE

FRONT	REAR		1	2	3	4	5	6
2	2	1	AAA					
1	2	2	BBB	CCC				
0	0	3						
5	1	4	FFF			DDD	EEE	
4	4	5				GGG		

- Here FRONT[K] contain front element of K of QUEUES and REAR[K] contain rear element of row K of QUEUE.
- The row maintains the queue of elements with priority number K.

UNIT - III

Date: / / Page No.

Trees

- Tree structure is mainly used to represent data containing a hierarchical relationship between elements.

Binary Trees

- A binary tree T is defined as a finite set of elements, such that
  1) T is empty [called the null tree or empty tree] or
  2) T contains a distringuished node R, called the root of T, and the remaining nodes of T form an ordered pair of disjoint binary trees T_1 and T_2.

- If T contains a distinguished node R, then the tree T_1 is called left subtree of R and T_2 is called right subtree of R.

- If T_1 is nonempty, then its root is called the left successor of R.
- If T_2 is nonempty, then its root is called the right successor of R.
- A binary tree T is presented by means of following diagram.

Date: / / Page No.

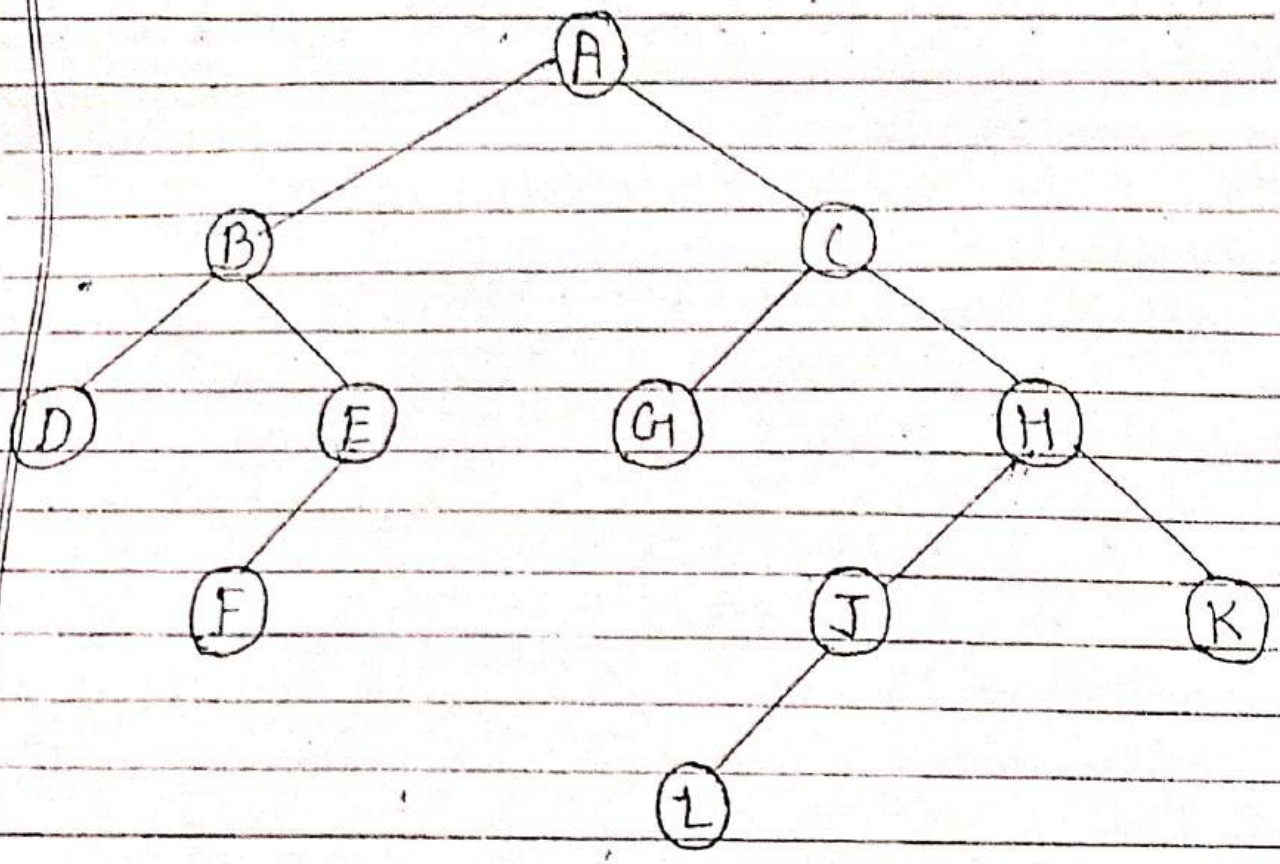


Fig. Binary Tree

Preorder : A B D F E C G H J L K

Inorder : D B F E A G C L J H K

Postorder : D F E B G L J K H C A

Date: / / Page No.

Fibonacci Sequence

- The fabonacci sequence is as follow: (denoted by $F_0, F_1, F_2, \ldots$)

  0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ...

Thus if $F_0 = 0$ and $F_1 = 1$ and each succeeding term is the sum of the two preceding terms.

Ex.

34 + 55 = 89

55 + 89 = 144

- A formal definition is as follow:

(a) If $n = 0$ or $n = 1$ then $F_n = n$

(b) If $n \geq 1$, then $F_n = F_{n-2} + F_{n-1}$

- The definition refers to itself when it uses F_{n-2} and F_{n-1}.

Here:

(a) the base values are 0 and 1

(b) the value of F_n is defined in terms of smaller values of n which are closer to the base values.

Accordingly this function is well defined.

Date: / / Page No.

```
/* Program for fabonacci serics using
   recursion */
#include<stdio.h>
#include<conio.h>
void Fibonacci(int n);
void main()
{
    int k,n;
    long int f;
    clrscr();
    printf(" Enter the range of the
            Fibonacci serics = ");
    scanf("%d", &n);
    printf("\n Fibonacci series : ");
    printf("%d %d", 0, 1);
    Fibonacci(n);
    getch();
}
Void Fibonacci(int n)
{
    static long int first=0, second=1,
                    sum;
    if(n>0)
    {
        sum = first + second;
        first = second;
        second = sum;
        printf("%d ", sum);
        Fibonacci(n-1);
    }
}
```

Output

Enter the range of the fibonacci series 4

Fabonacci series 0 1 1 2 3 5