MODULE 1:

INTRODUCTION TO DATA STRUCTURES

DATA STRUCTURES

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

The choice of a particular data model depends on the two considerations

1. It must be rich enough in structure to mirror the actual relationships of the data in the real world.
2. The structure should be simple enough that one can effectively process the data whenever necessary.

Basic Terminology: Elementary Data Organization:

Data: Data are simply values or sets of values.

Data items: Data items refers to a single unit of values.

Data items that are divided into sub-items are called **Group items**. Ex: An Employee Name may be divided into three subitems- first name, middle name, and last name.

Data items that are not able to divide into sub-items are called **Elementary items.**
Ex: SSN

Entity: An entity is something that has certain attributes or properties which may be assigned values. The values may be either numeric or non-numeric.

Ex:	Attributes-	Names,	Age,	Sex,	SSN
	Values-	Rohland Gail,	34,	F,	134-34-5533

Entities with similar attributes form an **entity set**. Each attribute of an entity set has a range of values, the set of all possible values that could be assigned to the particular attribute.
The term "information" is sometimes used for data with given attributes, of, in other words meaningful or processed data.

Field is a single elementary unit of information representing an attribute of an entity.

Record is the collection of field values of a given entity.

File is the collection of records of the entities in a given entity set.

Each record in a file may contain many field items but the value in a certain field may uniquely determine the record in the file. Such a field K is called a primary key and the values k1, k2, ….. in such a field are called keys or key values.

Records may also be classified according to length.
A file can have fixed-length records or variable-length records.

- In fixed-length records, all the records contain the same data items with the same amount of space assigned to each data item.
- In variable-length records file records may contain different lengths.

Example: Student records have variable lengths, since different students take different numbers of courses. Variable-length records have a minimum and a maximum length.

The above organization of data into fields, records and files may not be complex enough to maintain and efficiently process certain collections of data. For this reason, data are also organized into more complex types of structures.

The study of complex data structures includes the following three steps:

1. Logical or mathematical description of the structure
2. Implementation of the structure on a computer
3. Quantitative analysis of the structure, which includes determining the amount of memory needed to store the structure and the time required to process the structure.

CLASSIFICATION OF DATA STRUCTURES

Data structures are generally classified into

- Primitive data Structures
- Non-primitive data Structures

1. **Primitive data Structures:** Primitive data structures are the fundamental data types which are supported by a programming language. Basic data types such as integer, real, character and Boolean are known as Primitive data Structures. These data types consists of characters that cannot be divided and hence they also called simple data types.

2. **Non- Primitive data Structures:** Non-primitive data structures are those data structures which are created using primitive data structures. Examples of non-primitive data structures is the processing of complex numbers, linked lists, stacks, trees, and graphs.

Based on the structure and arrangement of data, non-primitive data structures is further classified into

1. Linear Data Structure

2. Non-linear Data Structure

1. Linear Data Structure:

A data structure is said to be linear if its elements form a sequence or a linear list. There are basically two ways of representing such linear structure in memory.

1. One wayis to have the linear relationships between the elements represented by means of sequential memory location. These linear structures are called arrays.
2. The other way is to have the linear relationship between the elements represented by means of pointers or links. These linear structures are called linked lists.

The common examples of linear data structure are Arrays, Queues, Stacks, Linked lists

2. Non-linear Data Structure:

A data structure is said to be non-linear if the data are not arranged in sequence or a linear. The insertion and deletion of data is not possible in linear fashion. This structure is mainly used to represent data containing a hierarchical relationship between elements. Trees and graphs are the examples of non-linear datastructure.

Arrays:

The simplest type of data structure is a linear (or one dimensional) array. A list of a finite number *n* of similar data referenced respectively by a set of *n* consecutive numbers, usually 1, 2, 3 .....*n*. if **A** is chosen the name for the array, then the elements of **A** are denoted by subscript notation $a_1, a_2, a_3 \ldots\ldots\ldots a_n$

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]

Example 1: A linear array STUDENT consisting of the names of six students is pictured in below figure. Here STUDENT [1] denotes John Brown, STUDENT [2] denotes Sandra Gold, and so on.

	STUDENT
1	John Brown
2	Sandra Gold
3	Tom Jones
4	June Kelly
5	Mary Reed
6	Alan Smith

Linear arrays are called one-dimensional arrays because each element in such an array is referenced by one subscript. A two-dimensional array is a collection of similar data elements where each element is referenced by two subscripts.

Example 2: A chain of 28 stores, each store having 4 departments, may list its weekly sales as in below fig. Such data can be stored in the computer using a two-dimensional array in which the first subscript denotes the store and the second subscript the department. If SALES is the name given to the array, then

SALES [1, 1] = 2872, SALES [1, 2] - 805, SALES [1, 3] = 3211,…., SALES [28, 4] = 982

Store \ Dept.	1	2	3	4
1	2872	805	3211	1560
2	2196	1223	2525	1744
3	3257	1017	3686	1951
. . .	. . .	. . .	. . .	. . .
28	2618	931	2333	982

Trees

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

Some of the basic properties of tree are explained by means of examples

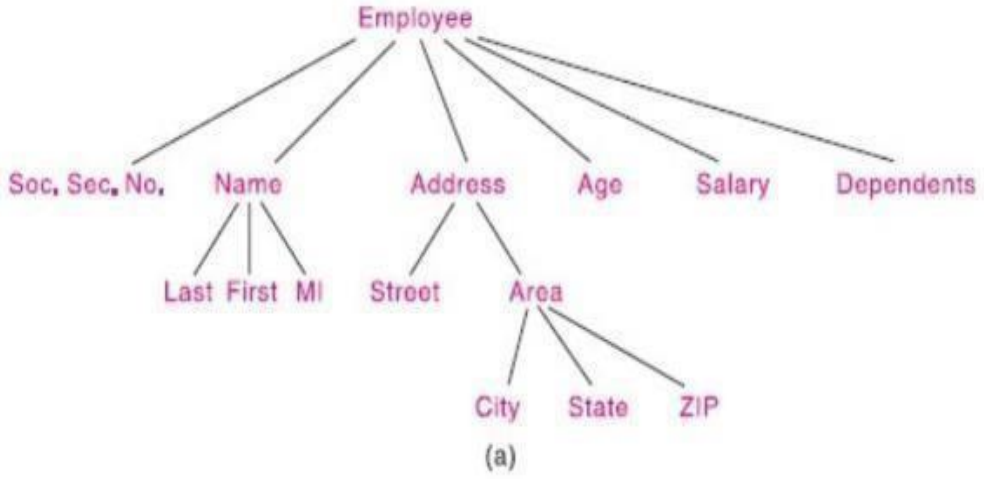
Example 1: Record Structure

Although a file may be maintained by means of one or more arrays a record, where one indicates both the group items and the elementary items, can best be described by means of a tree structure. For example, an employee personnel record may contain the following data items:

Social Security Number, Name, Address, Age, Salary, Dependents

However, Name may be a group item with the sub-items Last, First and MI (middle initial). Also Address may be a group item with the subitems Street address and Area address, where Area itself may be a group item having subitems City, State and ZIP codenumber.

This hierarchical structure is pictured below



(a)

Another way of picturing such a tree structure is in terms of levels, as shown below

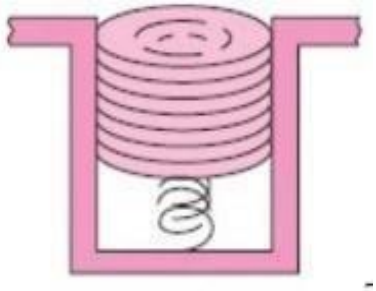
```
01 Employee
     02 Social Security Number
     02 Name
          03 Last
          03 First
          03 Middle Initial
     02 Address
          03 Street
          03 Area
               04 City
               04 State
               04 ZIP
     02 Age
     02 Salary
     02 Dependents
```

(b)

Some of the data structures are briefly described below.

1. **Stack:** A stack, also called a fast-in first-out (LIFO) system, is a linear list in which insertions and deletions can take place only at one end, called the top. This structure is similar in its operation to a stack of dishes on a spring system as shown in fig.

Note that new 4 dishes are inserted only at the top of the stack and dishes can be deleted only from the top of the Stack.



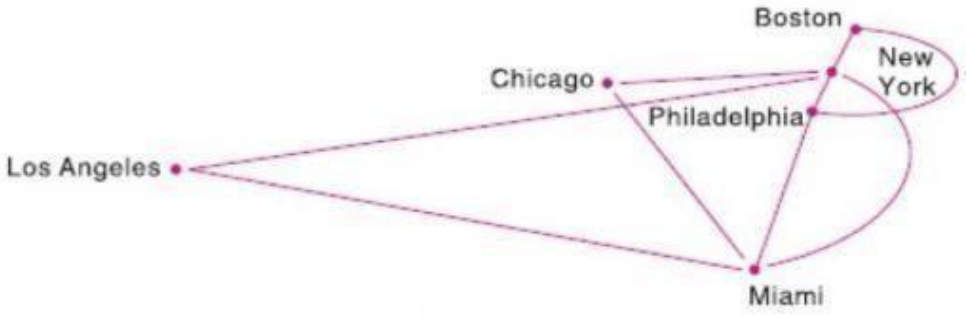
(a) Stack of dishes

2. **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, the "from" of the list, and insertions can take place only at the other end of the list, the “rear” of the list.

This structure operates in much the same way as a line of people waiting at a bus stop, as pictured in Fig. the first person in line is the first person to board the bus. Another analogy is with automobiles waiting to pass through an intersection the first car in line is the first car through.

Queue waiting for a bus

3. **Graph:** Data sometimes contain a relationship between pairs of elements which is not necessarily hierarchical in nature. For example, suppose an airline flies only between the cities connected by lines in Fig. Thedata structure which reflects this type of relationship is called a graph



Airline flights

DATA STRUCTURES OPERATIONS

The data appearing in data structures are processed by means of certain operations. The following four operations play a major role in this text:

1. **Traversing:** accessing each record/node exactly once so that certain items in the record may be processed. (This accessing and processing is sometimes called "**visiting**" the record.)
2. **Searching:** Finding the location of the desired node with a given key value, or finding the locations of all such nodes which satisfy one or more conditions.
3. **Inserting:** Adding a new node/record to the structure.
4. **Deleting:** Removing a node/record from the structure.

The following two operations, which are used in special situations:

1. **Sorting:** Arranging the records in some logical order (e.g., alphabetically according to some NAME key, or in numerical order according to some NUMBER key, such as social security number or account number)
2. **Merging:** Combining the records in two different sorted files into a single sorted file.

ARRAYS

- An Array is defined as, an ordered set of similar data items. All the data items of an array are stored in consecutive memory locations.
- The data items of an array are of **same type** and each data items can be accessed using the same **name** but different **index** value.
- An array is a set of pairs, <*index, value* >, such that each index has a value associated with it. It can be called as ***corresponding*** or a ***mapping***

Ex: <index, value>

< 0 , 25 > list[0]=25
< 1 , 15 > list[1]=15
< 2 , 20 > list[2]=20
< 3 , 17 > list[3]=17
< 4 , 35 > list[4]=35

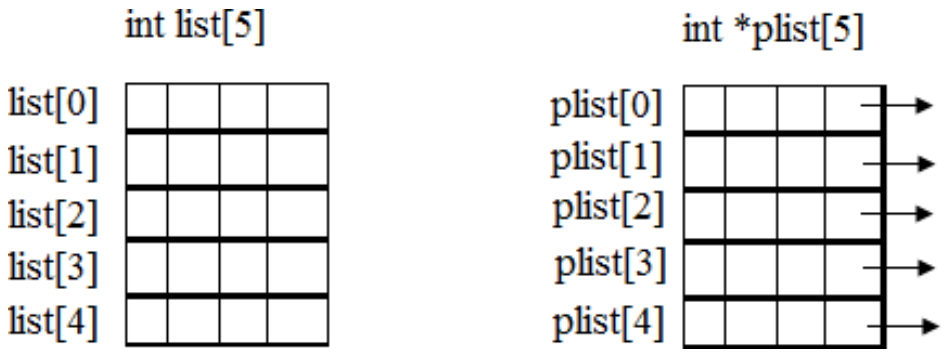
Here, ***list*** is the name of array. By using, list [0] to list [4] the data items in list can be accessed.

Array in C

Declaration: A one dimensional array in C is **declared** by adding brackets to the name of a variable.

Ex: **int list[5], *plist[5];**

- The array **list[5],** defines 5 integers and in C array start at index 0, so list[0], list[1], list[2], list[3], list[4] are the names of five array elements which contains an integer value.
- The array ***plist[5],** defines an array of 5 pointers to integers. Where, plist[0], plist[1], plist[2], plist[3], plist[4] are the five array elements which contains a pointer to an integer.



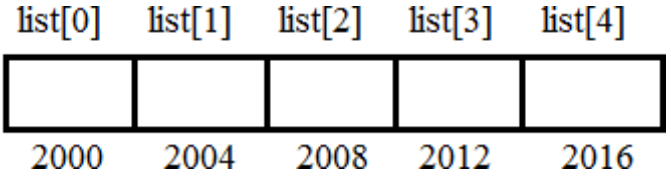
Implementation:

- When the complier encounters an array declaration, **list[5],** it allocates five consecutive memory locations. Each memory is enough large to hold a single integer.
- The address of first element of an array is called **Base Address.** Ex: For **list[5] t**he address of **list[0]** is called the base address.
- If the memory address of **list[i]** need to compute by the compiler, then the size of the **int** would get by **sizeof (int)**, then memory address of list[i] is as follows:

list[i] = α + i * sizeof (int)

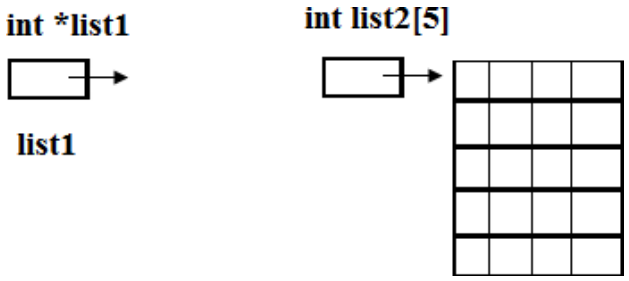
Where, α is base address.

list[3] = α+ 3 * sizeof(int)
= 2000 + 3 * 4
list[3] = 2012



Difference between **int *list1;** & **int list2[5];**

The variables **list1** and **list2** are both pointers to an **int,** but in **list2[5]** five memory locations are reserved for holding integers. **list2** is a pointer to **list2[0]** and **list2+i** is a pointerto **list2[i]**.



Note: In **C** the offset ***i*** do not multiply with the size of the type to get to the appropriate element of the array. Hence **(list2+i)** is equal **&list2[i]** and ***(list2+i)** is equal to **list2[i].**

How C treats an array when it is parameter to a function?

- All parameters of a C functions must be declared within the function. As various parameters are passed to functions, the name of an array can be passed as parameter.
- The range of a one-dimensional array is defined only in the main function since new storage for an array is not allocated within a function.
- If the size of a one dimensional array is needed, it must be passed into function as a argument or accessed as a global variable.

Example: Array Program

```
#define   MAX_SIZE   100
float sum(float [], int);
float   input[MAX_SIZE],   answer;
void main(void)
{
        int i;
        for( i=0; i<MAX_SIZE;  i++)
        input[i]=  i;
        answer  =  sum(input,  MAX_SIZE);
        printf("\n The sum is: %f \n",answer);
}

float sum(float list[], int n)
{
        int i;
        float  tempsum  =  0;
        for(i=0; i<n; i++)
                tempsum = tempsum + list[i];
        return tempsum;
```

When **sum** is invoked, **input=&input[0]** is copied into a temporary location and associated with the formal parameter ***list***

A function that prints out both the address of the *i*th element of the array and the value found at that address can written as shown in below program.

```
void print1 (int *ptr, int rows)
{
        int i;
        printf(“ Address contents \n”);
        for(i=0; i<rows; i++)
                printf(“% 8u %5d \n”, ptr+i, *(prt+i));
        printf(“\n”);
}
```

Output:

Address	**Content**
12244868	0
12344872	1
12344876	2
12344880	3
12344884	4

STRUCTURES

Ex:
```
struct {
        char name[10];
        int age;
        float salary;
} Person;
```

The above example creates a **structure** and variable name is **Person** and that has three fields:

name = a name that is a characterarray
age = an integer value representing the age of the person
salary = a float value representing the salary of the individual

Assign values to fields

To assign values to the fields, use . (dot) as the structure member operator. This operator is used to select a particular member of the structure

Ex:
```
strcpy(Person.name,"james");
Person.age =10;
Person.salary = 35000;
```

Type-Defined Structure

The structure definition associated with keyword **typedef** is called Type-Defined Structure.

Syntax 1: typedef struct
```
{
    data_type member 1;
    data_type member 2;
    ……
    ……
    data_type member n;
}Type_name;
```

Where,

- **typedef** is the keyword used at the beginning of the definition and by using typedef user defined data type can be obtained.
- **struct** is the keyword which tells structure is defined to the complier
- The members are declare with their data_type
- **Type_name** is not a variable, it is user defined data_type.

Syntax 2: struct struct_name

```
{
        data_type member 1;
        data_type member 2;
        ……….
        ……….
        data_type member n;
    };
typedef struct struct_name Type_name;
```

Ex:

```
typedef struct{
        char name[10];
        int age;
        float salary;
    }humanBeing;
```

In above example, **humanBeing** is the name of the type and it is a user defined data type.

Declarations of structure variables:

```
humanBeing  person1, person2;
```

This statement declares the variable **person1** and **person2** are of type **humanBeing.**

Structure Operation

The various operations can be performed on structures and structure members.

1. Structure Equality Check:

Here, the equality or inequality check of two structure variable of same type or dissimilar type is not allowed

```
typedef struct{
        char name[10];
        int age;
        float salary;
    }humanBeing;
humanBeing person1, person2;
```

if (person1 = = person2) is invalid.

The **valid function** is shown below

```
#define FALSE 0
#define TRUE 1
if (humansEqual(person1,person2))
        printf("The two human beings are the same\n");
else
        printf("The two human beings are not the same\n");
```

```
int humansEqual(humanBeing person1, humanBeing person2)
   { /* return TRUE if person1 and person2 are the same human being otherwise
                return FALSE */
      if    (strcmp(person1.name,    person2.name))
                return FALSE;
      if  (person1.age  !=  person2.age)
                return FALSE;
      if  (person1.salary  !=  person2.salary)
                return FALSE;
      return TRUE;
    }
```

Program: Function to check equality of structures

2. Assignment operation on Structure variables:

person1 = person2

The above statement means that the value of every field of the structure of person 2 is assigned as the value of the corresponding field of person 1, but this is invalid statement.

Valid Statements is given below:

```
strcpy(person1.name, person2.name);
person1.age = person2.age;
person1.salary = person2.salary;
```

Structure within a structure:

There is possibility to embed a structure within a structure. There are 2 ways to embed structure.

1. The structures are defined separately and a variable of structure type is declared inside the definition of another structure. The accessing of the variable of a structure type that are nested inside another structure in the same way as accessing other memberof that structure

Example: The following example shows two structures, where both the structure are defined separately.

```
typedef struct {
        int month;
        int  day;
        int year;
    }date;

typedef struct {
        char name[10];
        int age;
        float salary;
        date dob;
    } humanBeing;
humanBeing person1;
```

A person born on February 11, 1944, would have the values for the date struct set as:

```
person1.dob.month = 2;
person1.dob.day = 11;
person1.dob.year = 1944;
```

2. The complete definition of a structure is placed inside the definition of another structure.

Example:

```
typedef struct {
        char  name[10];
        int age;
        float salary;
        struct {
                int month;
                int  day;
                int year;
            } date;
    } humanBeing;
```

SELF-REFERENTIAL STRUCTURES

A self-referential structure is one in which one or more of its components is a pointer to itself. Self-referential structures usually require dynamic storage management routines (malloc and free) to explicitly obtain and release memory.

Consider as an example:

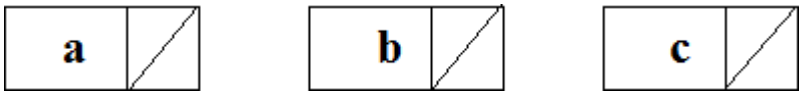
```
typedef struct {
        char data;
        struct list *link ;
} list;
```

Each instance of the structure **list** will have two components **data** and **link**.

- **Data:** is a single character,
- **Link:** link is a pointer to a list structure. The value of link is either the address in memory of an instance of list or the null pointer.

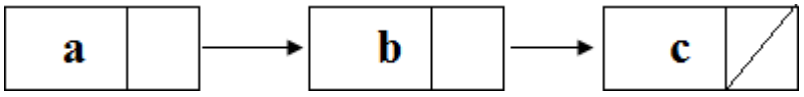
Consider these statements, which create three structures and assign values to their respective fields:

```
list item1, item2, item3;
item1.data = 'a';
item2.data = 'b';
item3.data = 'c';
item1.link = item2.1ink = item3.link = NULL;
```



Structures item1, item2 and item3 each contain the data item **a, b,** and **c** respectively, and the null pointer. These structures can be attached together by replacing the **null link** field in item 2 with one that points to item 3 and by replacing the null link field in item 1 with one that points to item 2.

```
item1.link = &item2;
item2.1ink = &item3;
```



Unions:

A union is similar to a structure, it is collection of data similar data type or dissimilar.

Syntax:
```
union{
        data_type  member 1;
        data_type  member 2;
        ……….
        ……….
        data_type member n;
}variable_name;
```

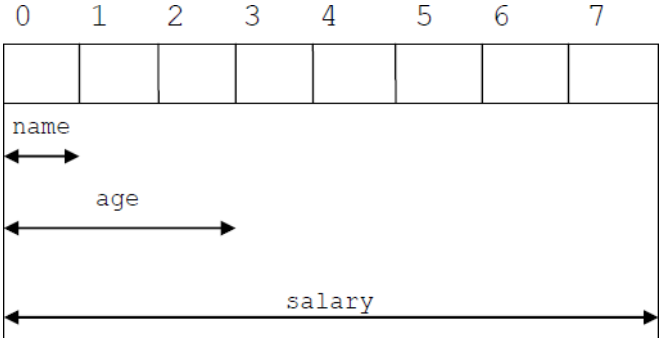
Example:

```
union{
        int children;
        int beard;
} u;
```

Union Declaration:

A union declaration is similar to a structure, but the fields of a union must share their memory space. This means that only one field of the union is "active" at any given time.

```
union{
        char  name;
        int       age;
        float salary;
}u;
```



The major difference between a union and a structure is that unlike structure members which are stored in separate memory locations, all the members of union must share the same memory space. This means that only one field of the union is "active" at any given time.

Example:

```
#include <stdio.h>
union job {
                    char name[32];
                    float salary;
                    int worker_no;
                  }u;

int main( ){
                    printf("Enter name:\n");
                    scanf("%s", &u.name);
                    printf("Enter salary: \n");
                    scanf("%f", &u.salary);
                    printf("Displaying\n Name :%s\n",u.name);
                    printf("Salary: %.1f",u.salary);
                    return 0;
}
```

Output:

Enter name: Albert
Enter salary: 45678.90

Displaying
Name: f%gupad (Garbage Value)
Salary: 45678.90

POINTERS

A pointer is a variable which contains the address in memory of another variable.

The two most important operator used with the pointer type are

& - The unary operator ***&*** which gives the address of a variable

* - The indirection or dereference operator * gives the content of the object pointed to by apointer.

Declaration

```
int i, *pi;
```

Here, ***i*** is the integer variable and ***pi*** is a pointer to an integer

```
pi = &i;
```

Here, &i returns the address of ***i*** and assigns it as the value of ***pi***

Null Pointer

The null pointer points to no object or function.
The null pointer is represented by the integer 0.
The null pointer can be used in relational expression, where it is interpreted as false.

Ex: if (pi = = NULL) or if (!pi)

Pointers can be Dangerous:

Pointer can be very dangerous if they are misused. The pointers are dangerous in following situations:

1. Pointer can be dangerous when an attempt is made to access an area of memory that is either out of range of program or that does not contain a pointer reference to a legitimate object.

Ex:
```
main ()
{
    int *p;
    int pa = 10;
    p = &pa;
    printf("%d", *p);      //output = 10;
    printf("%d", *(p+1)); //accessing memory which is out of range
}
```

2. It is dangerous when a NULL pointer is de-referenced, because on some computer it may return 0 and permitting execution to continue, or it may return the result stored in location zero, so it may produce a serious error.

3. Pointer is dangerous when use of explicit **type casts** in converting between pointer types

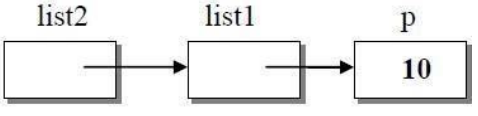
Ex:
```
pi = malloc (sizeof (int));
pf = (float*) pi;
```

4. In some system, pointers have the same size as type **int**, since **int** is the default type specifier, some programmers omit the return type when defining a **function**. The return type defaults to **int** which can later be interpreted as a pointer. This has proven to be a dangerous practice on some computer and the programmer is made to define explicit types for functions.

Pointers to Pointers

A variable which contains address of a pointer variable is called pointer-to-pointer.

Example:
```
int p;
int *list1, * *list2;
p=10;
list1=&p;
list2=&list1;
printf("%d, %d, %d", a, *list1, **list2);
```



Output: 10 10 10

DYNAMIC MEMORY ALLOCATION FUNCTIONS

1. malloc( ):

The function *malloc* allocates a user- specified amount of memory and a pointer to the start of the allocated memory is returned.

If there is insufficient memory to make the allocation, the returned value is NULL.

Syntax:

```
data_type *x;
x= (data_type *) malloc(size);
```

Where,

x is a pointer variable of data_type

size is the number of bytes

Ex:

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

2. calloc( ):

The function *calloc* allocates a user- specified amount of memory and initializes the allocated memory to **0** and a pointer to the start of the allocated memory is returned.

If there is insufficient memory to make the allocation, the returned value is NULL.

Syntax:

```
data_type *x;
x= (data_type *) calloc(n, size);
```

Where,

x is a pointer variable of type int

n is the number of block to be allocated

size is the number of bytes in each block

Ex:

```
int *x
x= calloc (10, sizeof(int));
```

The above example is used to define a one-dimensional array of integers. The capacity of this array is n=10 and x [0: n-1] (x [0, 9]) are initially 0

Macro CALLOC

```
#define CALLOC (p, n, s)\
if ( ! ((p) = calloc (n, s)))\
    {\
        fprintf(stderr, "Insuffiient memory");\
        exit(EXIT_FAILURE);\
    }\
```

3. realloc( ):

- Before using the realloc( ) function, the memory should have been allocated using malloc( ) or calloc( ) functions.
- The function relloc( ) resizes memory previously allocated by either ***mallor*** or ***calloc***, which means, the size of the memory changes by extending or deleting the allocated memory.
- If the existing allocated memory need to extend, the pointer value will not change.
- If the existing allocated memory cannot be extended, the function allocates a new block and copies the contents of existing memory block into new memory block and then deletes the old memory block.
- When realloc is able to do the resizing, it returns a pointer to the start of the new block and when it is unable to do the resizing, the old block is unchanged and the function returns the value NULL

Syntax:

```
data_type *x;
x= (data_type *) realloc(p, s );
```

The size of the memory block pointed at by p changes to S. When **s > p** the additional **s-p** memory block have been extended and when **s < p**, then **p-s** bytes of the old block are freed.

Macro REALLOC

```
#define REALLOC(p,S)\
if (!((p) = realloc(p,s))) \
    { \
        fprintf(stderr, "Insufficient memory");\
        exit(EXIT_FAILURE);\
    }\
```

4. free( )

Dynamically allocated memory with either malloc( ) or calloc ( ) does not return on its own. The programmer must use free( ) explicitly to release space.

Syntax:

```
free(ptr);
```

This statement cause the space in memory pointer by ptr to be deallocated

REPRESENTATION OF LINEAR ARRAYS IN MEMORY

Linear Array

A linear array is a list of a finite number *'n'* of homogeneous data element such that

a. The elements of the array are reference respectively by an index set consisting of n consecutive numbers.
b. The element of the array are respectively in successive memory locations.

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

When LB = 0,

Length = UB – LB + 1

When LB = 1,

Length = UB

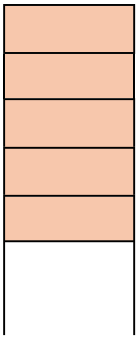
Where,

UB is the largest index called the Upper Bound
LB is the smallest index, called the Lower Bound

Representation of linear arrays in memory

Let LA be a linear array in the memory of the computer. The memory of the computer is simply a sequence of address location as shown below,

1000
1001
1002
1003
1004



LOC (LA [K]) = address of the element LA [K] of the array LA

The elements of LA are stored in successive memory cells.

The computer does not keep track of the address of every element of LA, but needs to keep track only the address of the first element of LA denoted by,

Base (LA)

and called the base address of LA.

Using the base address of LA, the computer calculates the address of any element of LA by the formula

LOC (LA[K]) = *Base*(LA) + w(K – lower bound)

Where, w is the number of words per memory cell for the array LA.

DYNAMICALLY ALLOCATED ARRAYS

One Dimensional Array

While writing computer programs, if finds ourselves in a situation where we cannot determine how large an array to use, then a good solution to this problem is to defer this decision to run time and allocate the array when we have a good estimate of the required array size.

Example:

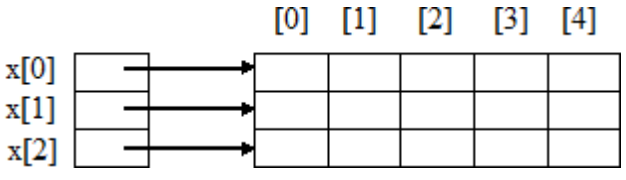
```
int i, n, *list;
printf("Enter the number of numbers to generate:");
scanf("%d", &n);
if(n<1)
{
        fprintf (stderr, "Improper value of n \n");
        exit(EXIT_FAILURE);
}
MALLOC (list, n*sizeof(int));
```

The programs fails only when n<1 or insufficient memory to hold the list of numbers that are to be sorted.

Two DimensionalArrays

C uses array-of-arrays representation to represent a multidimensional array. The two dimensional arrays is represented as a one-dimensional array in which each element is itself a one-dimensional array.

Example: int x[3][5];



Array-of-arrays representation

C find element x[i][j] by first accessing the pointer in x[i].
Where **x[i] = α+ i* sizeof(int)**, which give the address of the zeroth element of row ***i*** of the array.
Then adding **j*sizeof(int)** to this pointer (**x[i]**) **,** the address of the [j]th element of row ***i*** is determined**.**

x[i] = α+ i* sizeof(int)
x[j] = α+ j* sizeof(int)
x[i][j] = x[i]+ i* sizeof(int)

Creation of Two-Dimensional Array Dynamically

```
int **myArray;
myArray = make2dArray(5,10);
myArray[2][4]=6;

int ** make2dArray(int rows, int cols)
{ /* create a two dimensional rows X cols array */
        int **x, i;
        MALLOC(x, rows * sizeof (*x)); /*get memory for row pointers*/
        for (i= 0;i<rows; i++)                  /* get memory for each row */
        MALLOC(x[i], cols *sizeof(**x));
        return x;
}
```

The second line allocates memory for a 5 by 10 two-dimensional array of integers and the third line assigns the value 6 to the [2][4] element of this array.

ARRAY OPERATIONS

1. Traversing

- Let A be a collection of data elements stored in the memory of the computer. Suppose if the contents of the each elements of array A needs to be printed or to count the numbers of elements of A with a given property can be accomplished by Traversing.
- Traversing is a accessing and processing each element in the array exactly once.

Algorithm 1: (Traversing a Linear Array)

Hear LA is a linear array with the lower bound LB and upper bound UB. This algorithm traverses LA applying an operation PROCESS to each element of LA using while loop.

1. [Initialize Counter] set K:= LB
2. Repeat step 3 and 4 while K ≤ UB
3. [Visit element] Apply PROCESS to LA [K]
4. [Increase counter] Set K:= K + 1
   [End of step 2 loop]
5. Exit

Algorithm 2: (Traversing a Linear Array)

Hear LA is a linear array with the lower bound LB and upper bound UB. This algorithm traverses LA applying an operation PROCESS to each element of LA using repeat – for loop.

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

Example:
Consider the array AUTO which records the number of automobiles sold each year from 1932 through 1984.

To find the number NUM of years during which more than 300 automobiles were sold, involves traversing AUTO.

1. [Initialization step.] Set NUM := 0
2. Repeat for K = 1932 to 1984:
   If AUTO [K] > 300, then: Set NUM: = NUM + 1.
   [End of loop.]
3. Return.

2. Inserting

- Let A be a collection of data elements stored in the memory of the computer. Inserting refers to the operation of adding another element to the collection A.
- Inserting an element at the "end" of the linear array can be easily done provided the memory space allocated for the array is large enough to accommodate the additional element.
- Inserting an element in the middle of the array, then on average, half of the elements must be moved downwards to new locations to accommodate the new element and keep the order of the otherelements.

Algorithm:

INSERT (LA, N, K, ITEM)

Here LA is a linear array with N elements and K is a positive integer such that $K \leq N$. This algorithm inserts an element ITEM into the K^{th} position in LA.

1. [Initialize counter] set J:= N
2. Repeat step 3 and 4 while $J \geq K$
3. [Move J^{th} element downward] Set LA [J+1] := LA[J]
4. [Decrease counter] set J:= J – 1

   [End of step 2 loop]
5. [Insert element] set LA[K]:= ITEM
6. [Reset N] set N:= N+1
7. Exit

3. Deleting

- Deleting refers to the operation of removing one element to the collection A.
- Deleting an element at the "end" of the linear array can be easily done with difficulties.
- If element at the middle of the array needs to be deleted, then each subsequent elements be moved one location upward to fill up the array.

Algorithm

DELETE (LA, N, K, ITEM)

Here LA is a linear array with N elements and K is a positive integer such that $K \leq N$. this algorithm deletes the K^{th} element from LA

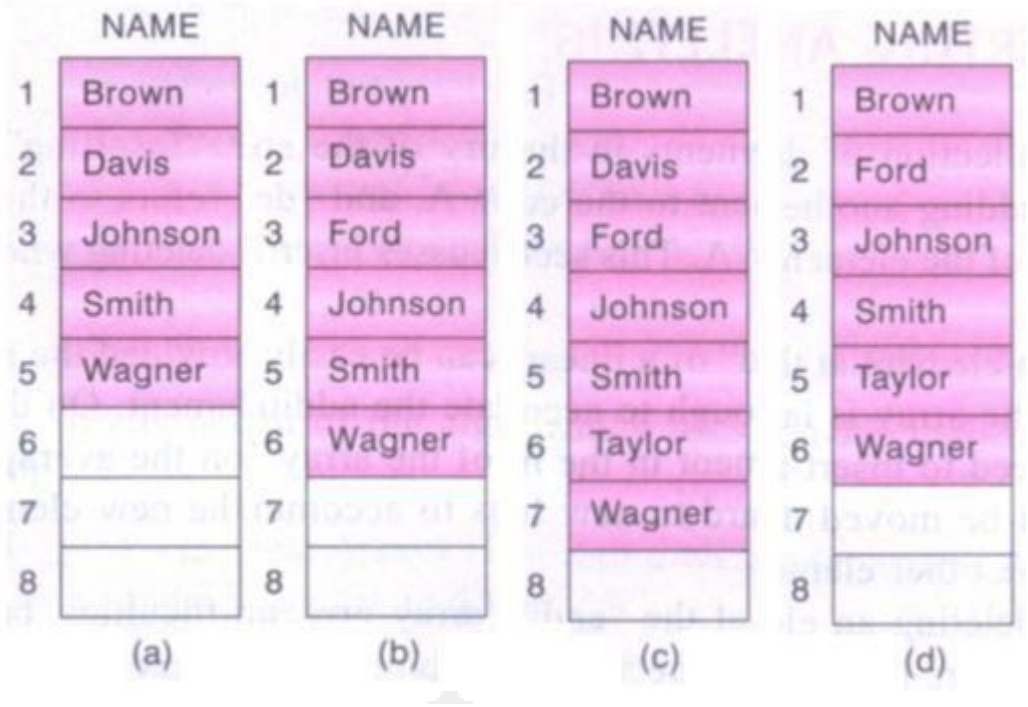
1. Set ITEM:= LA[K]
2. Repeat for J = K to N – 1

   [Move J + 1 element upward] set LA[J]:= LA[J+1]

   [End of loop]
3. [Reset the number N of elements in LA] set N:= N – 1
4. Exit

Example: Inserting and Deleting

Suppose NAME is an 8-element linear array, and suppose five names are in the array, as in Fig.(a). Observe that the names are listed alphabetically, and suppose we want to keep the array names alphabetical at all times. Suppose ***Ford*** is added to the array. Then ***Johnson, Smith*** and ***Wagner*** must each be moved downward one location, as in Fig.(b). Next suppose Taylor is added to the array; then Wagner must be moved, as in Fig.(c). Last, suppose Davis is removed from the array. Then the five names Ford, Johnson, Smith, Taylor and Wagner must each be moved upward one location, as in Fig.(d).



4. Sorting

Sorting refers to the operation of rearranging the elements of a list. Here list be a set of n elements. The elements are arranged in increasing or decreasing order.

Ex: suppose A is the 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,

$$A[I] < A[2] < A[3] < ... < A[N]$$

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

Bubble Sort

Suppose the list of numbers A[l], A[2], ... , A[N] is in memory. The bubble sort algorithm works as follows:

Algorithm: Bubble Sort – BUBBLE (DATA, N)
Here DATA is an array with N elements. This algorithm sorts the elements in DATA.

1. Repeat Steps 2 and 3 for K = 1 to N - 1.
2. Set PTR: = 1. [Initializes pass pointer PTR.]
3. Repeat while PTR ≤ N - K: [Executes 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 inner loop.]
   [End of Step 1 outer loop.]
4. Exit.

Example:

Suppose the following numbers are stored in an array A:

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

We apply the bubble sort to the array A. We discuss each pass separately.
Pass 1. We have the following comparisons:

(a) Compare A_1 and A_2. Since 32 < 51, the list is not altered.
(b) Compare A_2 and A_3. Since 51 > 27, interchange 51 and 27 as follows:
32, 27, 51, 85, 66, 23, 13, 57
(c) Compare A_3 and A_4. Since 51 < 85, the list is not altered.
(d) Compare A_4 and A_5. Since 85 > 66, interchange 85 and 86 as follows:
32, 27, 51, 66, 85, 23, 13, 57
(e) Compare A_5 and A_6. Since 85 > 23, interchange 85 and 23 as follows:
32, 27, 51, 66, 23, 85, 13, 57
(f) Compare A_6 and A_7. Since 85 > 13, interchange 85 and 13 to yield:
32, 27, 51, 66, 23, 13, 85, 57
(g) Compare A_7 and A_8. Since 85 > 57, interchange 85 and 51 to yield:
32, 27, 51, 66, 23, 13, 57, 85

At the end of this first pass, the largest number, 85, has moved to the last position. However, the rest of the numbers are not sorted, even though some of them have changed their positions.

For the remainder of the passes, we show only the interchanges.

Pass 2. (27, 33,) 51, 66, 23, 13, 57, 85
27, 33, 51, (23, 66,) 13, 57, 85
27, 33, 51, 23, (13, 66,) 57, 85
27, 33, 51, 23, 13, (57, 66,) 85

At the end of Pass 2, the second largest number, 66, has moved its way down to the next-to-last position.

Pass 3. 27, 33, (23, 51,) 13, 57, 66, 85
27, 33, 23, (13, 51,) 57, 66, 85

Pass 4. 27, (23, 33,) 13, 51, 57, 66, 85
27, 23, (13, 33,) 51, 57, 66, 85

Pass 5. (23, 27,) 13, 33, 51, 57, 66, 85
23, (13, 27,) 33, 51, 57, 66, 85

Pass 6. (13, 23,) 27, 33, 51, 57, 66, 85

Pass 6 actually has two comparisons, A_1 with A_2 and A_2 and A_3. The second comparison does not involve an interchange.

Pass 7. Finally, A_1 is compared with A_2. Since 13 < 23, no interchange takes place.

Since the list has 8 elements; it is sorted after the seventh pass.

Complexity of the Bubble Sort Algorithm

The time for a sorting algorithm is measured in terms of the number of comparisons f(n). 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 in the next-to-last position; and so on. Thus

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

5. 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.
- Thesearch is said to be successful if ITEMdoes appear in DATA and unsuccessful otherwise.

Linear Search

Suppose DATA is a linear array with n elements. Given no other information about DATA, The way to search for a given ITEM in DATA is to compare ITEM with each element of DATA one by one. That is, first test whether DATA [l] = ITEM, and then test whether DATA[2] = ITEM, and so on. This method, which traverses DATA sequentially to locate ITEM, is called ***linear search or sequential search.***

Algorithm: (Linear Search) LINEAR (DATA, N, ITEM, LOC)
Here DATA is a linear array with N elements, and ITEM is a given item of information. This algorithm finds the location LOC of ITEM in DATA, or sets LOC: = 0 if the search is unsuccessful.

1. [Insert ITEM at the end of DATA.] Set DATA [N + 1]: = ITEM.
2. [Initialize counter.] Set LOC: = l.
3. [Search forITEM.]
   Repeat while DATA [LOC] ≠ ITEM:
   Set LOC:= LOC + 1.
   [End of loop.]
4. [Successful?] If LOC = N + 1, then: Set LOC:=0
5. Exit.

Complexity of the Linear Search Algorithm

Worst Case: The worst case occurs when one must search through the entire array DATA, i.e., when ITEM does not appear in DATA. In this case, the algorithm requires comparisons.

$$f(n) = n + 1$$

Thus, in the worst case, the running time is proportional to n.

Average Case: The average number of comparisons required to find the location of ITEM is approximately equal to half the number of elements in the array.

$$f(n) = \frac{n+1}{2}$$

Binary Search

Suppose DATA is an array which is sorted in increasing numerical order or, equivalently, alphabetically. Then there is an extremely efficient searching algorithm, called *binary search*, which can be used to find the location LOC of a given ITEM of information in DATA.

Algorithm: (Binary Search) BINARY (DATA, LB, UB, ITEM, LOC)
Here DATA is a sorted array with lower bound LB and upper bound UB, and ITEM is a given item of information. The variables BEG, END and MID denote, the beginning, end and middle locations of a segment of elements of DATA.
This algorithm finds the location LOC of ITEM in DATA or sets LOC = NULL.

1. [Initialize segment variables.]
   Set BEG: = LB, END := UB and MID = INT((BEG + END)/2).
2. Repeat Steps 3 and 4 while BEG ≤ END and DATA [MID] ≠ ITEM.
3. If ITEM < DATA [MID], then:
   Set END := MID - 1.
   Else:
   Set BEG := MID + 1.
   [End of If structure.]
4. Set MID := INT((BEG + END)/2).
   [End of Step 2 loop.]
5. If DATA[MID] = ITEM, then:
   Set LOC := MID.
   Else:
   Set LOC := NULL.
   [End of If structure.]
6. Exit.

Remark: Whenever ITEM does

Complexity of the Binary Search Algorithm

The complexity is measured by the number *f(n)* of comparisons to locate ITEM in DATA where DATA contains *n* elements. Observe that each comparison reduces the sample size in half. Hence we require at most *f(n)* comparisons to locate ITEM where

$2^{f(n)} > n$ or equivalently $f(n) = [\log_2 n] + 1$

That is, the running time for the worst case is approximately equal to $\log_2 n$. One can also show that the running time for the average case is approximately equal to the running time for the worstcase.

MULTIDIMENSIONAL ARRAY

Two-Dimensional Arrays

A two-dimensional m x 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

$$1 \le J \le m \text{ and } 1 \le K \le n$$

The element of A with first subscript j and second subscript k will be denoted by $A_{J,K}$ or A[J, K]

Two-dimensional arrays are called ***matrices*** in mathematics and ***tables*** in business applications.

There is a standard way of drawing a two-dimensional m x 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*.

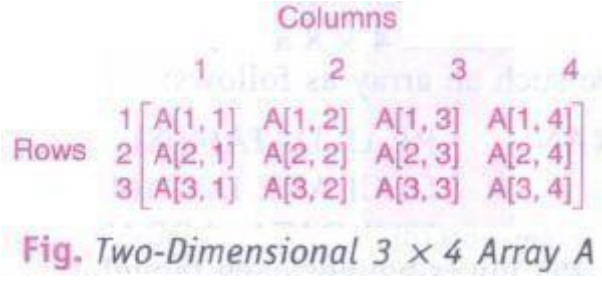
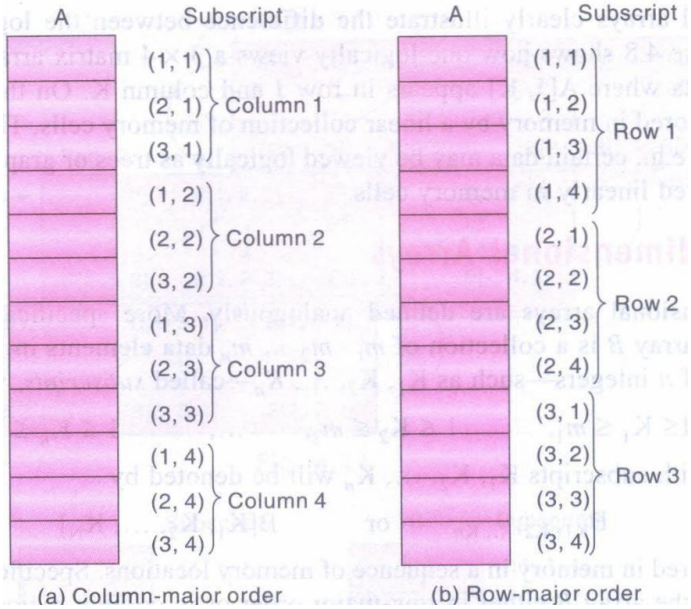


Fig. *Two-Dimensional 3 × 4 Array A*

Representation of Two-Dimensional Arrays in Memory

Let A be a two-dimensional *m* x *n* array. Although A is pictured as a rectangular array of elements with *m* rows and *n* columns, the array will be represented in memory by a block of *m . n* sequential memory locations.

The programming language will store the array A either (1) column by column, is called



column-major order, or (2) row by row, in *row-major order*

The computer uses the formula to find the address of LA[K] in time independent of K.

LOC (LA[K]) = *Base(LA)* + w(K - 1)

The computer keeps track of *Base(A)-the* address of the first element A[1, 1] of A-and computes the address LOC(A[J, K]) of A[J, K] using the formula

(Column-major order) LOC(A[J, K]) = *Base(A)* + w[M(K - 1)+ (J - 1)]

(Row-major order) LOC(A[J, K]) = *Base(A)* + w[N(J - 1) + (K - 1)]

General Multidimensional Arrays

An n-dimensional m_1 X m_2 X ... X m_n array B is a collection of m_1, m_2 ... m_n data elements in which each element is specified by a list of n integers-such as K_1 K_2 ... , K_n called subscripts, with the property that

$1 \le K_1 \le m_1$, $1 \le K_2 \le m_2$ ….. $1 \le K_n \le m_n$

The element of *B* with subscripts K_1 K_2 ... , K_n will be denoted by B[K_1 K_2 ... , K_n]

The programming language will store the array *B* either in row-major order or in column-major order.

Let C be such an n-dimensional array. The index set for each dimension of C consists of the consecutive integers from the lower bound to the upper bound of the dimension. The length L_i of dimension i of C is the number of elements in the index set, and L_i can be calculated, as

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

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

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

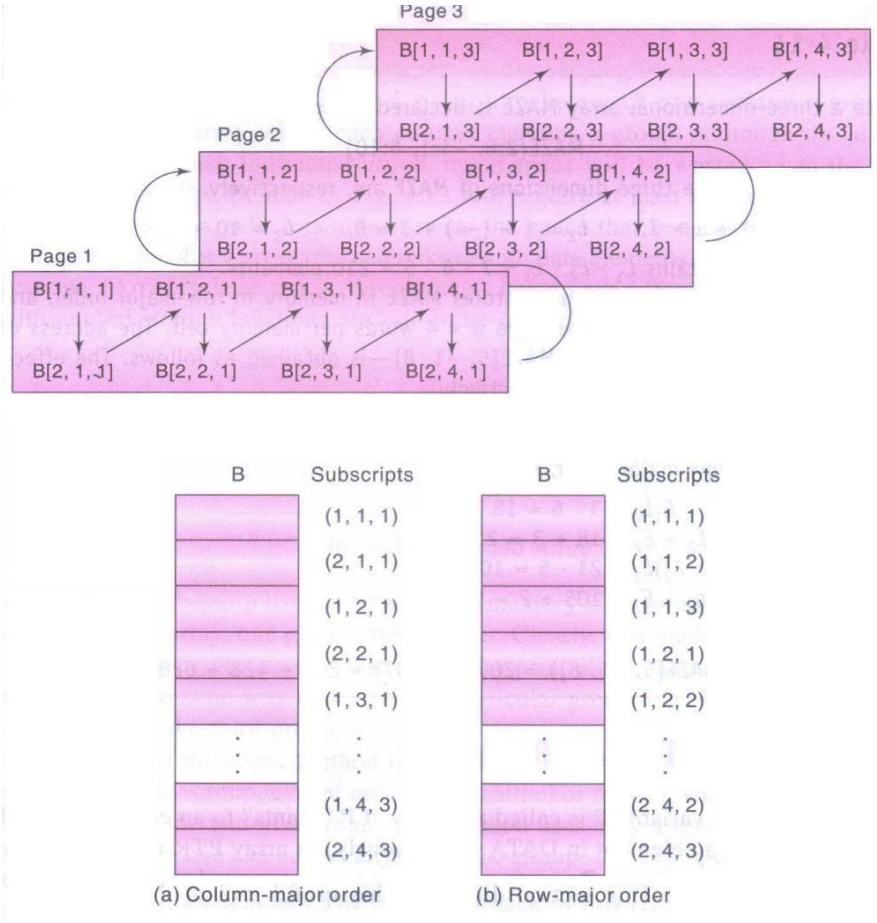
Then the address LOC(C[K_1 K_2 ... , K_n] of an arbitrary element of C can be obtained from the formula

$Base(C) + w[(((... (E_N L_{N-1}] + E_{N-1}])L_{N-2}) + ... + E_3))L_2 + E_2)L_1 + E_1]$

or from the formula

$Base(C) + w[(... ((E_1 L_2 + E2)L_3 + E_3)L_4 + ... + E_{N-1})L_N + E_N]$

according to whether C is stored in column-major or row-major order.



POLYNOMIALS

What is a polynomial?

"A polynomial is a sum of terms, where each term has a form $\mathbf{ax^e}$, where **x** is the variable, **a** is the coefficient and **e** is the exponent."

Two example polynomials are:

$A(x) = 3x^{20} + 2x^5 + 4$

$B(x) = x^4 + 10x^3 + 3x^2 + 1$

The largest (or leading) exponent of a polynomial is called its degree. Coefficients that are zero are not displayed. The term with exponent equal to zero does not show the variable since **x** raised to a power of zero is **1**.

Assume there are two polynomials,

$A(x) = \Sigma\, a_i x_i$ and $B(x) = \Sigma\, b_i x_i$

then:

$A(x) + B(x) = \Sigma\, (a_i + b_i)\, x_i$

$A(x).B(x) = \Sigma\, (a_i x_i . \Sigma\, (b_j x_j))$

Polynomial Representation

One way to represent polynomials in C is to use **typedef** to create the type polynomial as below:

```
#define MAX-DEGREE  101                 /*Max degree of polynomial+1*/
typedef struct{
                    int degree;
                    float coef[MAX-DEGREE];
              } polynomial;
```

Now if **a** is a variable and is of type polynomial and n < MAX_DEGREE, the polynomial $A(x) = \Sigma a_i x_i$ would be represented as:

a.degree = n

$a.coef[i] = a_{n-i}$, $0 \leq i \leq n$

In this representation, the coefficients is stored in order of decreasing exponents, such that a**.**coef [i] is the coefficient of x^{n-i} provided a term with exponent n-i exists;
Otherwise, a**.**coef [i] =0. This representation leads to very simple algorithms for most of the operations, it wastes a lot of space.

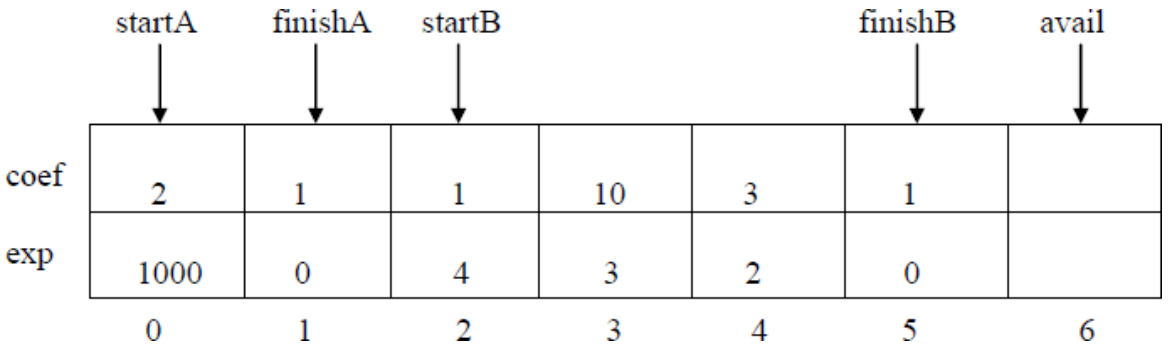
To preserve space an alternate representation that uses only one global array, **terms** to store all polynomials.
The C declarations needed are:

```
MAX_TERMS  100                  /*size of terms array*/
typedef struct{
                    float coef;
                    int expon;
              }       polynomial;
polynomial terms[MAX-TERMS];
int avail = 0;
```

Consider the two polynomials

$A(x) = 2x^{1000} + 1$

$B(x) = x^4 + 10x^3 + 3x^2 + 1$



- The above figure shows how these polynomials are stored in the array terms. The index of the first term of A and B is given by startA and startB, while finishA and finishB give the index of the last term of A and B.
- The index of the next free location in the array is given by avail.
- For above example, startA=0, finishA=1, startB=2, finishB=5, & avail=6.

Polynomial Addition

- C function is written that adds two polynomials, A and B to obtain D =A + B.
- To produce D (x), **padd( )** is used to add A (x) and B (x) term by term. Starting at position avail, **attach( )** which places the terms of D into the array, ***terms***.
- If there is not enough space in terms to accommodate D, an error message is printed to the standard error device & exits the program with an error condition

```
void padd(int startA, int finishA, int startB, int finishB, int *startD,int *finishD)
{ /* add A(x) and B(x) to obtain D(x) */
        float coefficient;
        *startD = avail;
        while (startA <= finishA && startB <= finishB)
        switch(COMPARE(terms[startA].expon, terms[startB].expon))
        {
                case -1:                /* a expon < b expon */
                        attach (terms [startB].coef, terms[startB].expon);
                        startB++;
                        break;

                case 0:                 /* equal exponents */
                        coefficient = terms[startA].coef + terms[startB].coef;

                        if (coefficient)
                                attach (coefficient, terms[startA].expon);
                        startA++;
                        startB++;
                        break;
```

```
        case 1:                 /* a expon > b expon */
                attach (terms [startA].coef, terms[startA].expon);
                startA++;
    }

    /* add in remaining terms of A(x) */
    for(; startA <= finishA; startA++)
        attach (terms[startA].coef, terms[startA].expon);

    /* add in remaining terms of B(x) */
    for( ; startB <= finishB; startB++)
        attach (terms[startB].coef, terms[startB].expon);
    *finishD = avail-i;
```

Function to add two polynomials

```
void attach(float coefficient, int exponent)
    {                       /* add a new term to the polynomial */
        if (avail >= MAX-TERMS)
        {
            fprintf(stderr,"Too many terms in the polynomial\n");
            exit(EXIT_FAILURE);
        }
        terms[avail].coef = coefficient;
        terms[avail++].expon = exponent;
    }
```

Function to add new term

Analysis of padd( ):

The number of non-zero terms in A and B is the most important factors in analyzing the time complexity.

Let **m** and **n** be the number of non-zero terms in A and B, If m >0 and n > 0, the **while** loop is entered. Each iteration of the loop requires O(1) time. At each iteration, the value of startA or startB or both is incremented. The iteration terminates when either startA or startB exceeds finishA or finishB.

The number of iterations is bounded by **m + n -1**

$A(x) = \sum_{i=0}^{n} x^{2i}$ and $B(x) = \sum_{i=0}^{m} x^{2i+1}$

The time for the remaining two **for** loops is bounded by O(n + m) because we cannot iterate the first loop more than m times and the second more than n times. So, the asymptotic computing time of this algorithm is O(n +m).

SPARSE MATRICES

A matrix contains **m rows** and **n columns** of elements as illustrated in below figures. In this figure, the elements are numbers. The first matrix has five rows and three columns and the second has six rows and six columns. We write m x n (read "m by n") to designate a matrix with m rows and n columns. The total number of elements in such a matrix is **mn**. If **m** equals **n**, the matrix is square.

	col0	col1	col2
row 0	-27	3	4
row 1	6	82	-2
row 2	109	-64	11
row 3	12	8	9
row 4	48	27	47

Figure A

	col0	col1	col2	col3	col4	col 5
row0	15	0	0	22	0	-15
row1	0	11	3	0	0	0
row2	0	0	0	-6	0	0
row3	0	0	0	0	0	0
row4	91	0	0	0	0	0
row5	0	0	28	0	0	0

Figure B

What is Sparse Matrix?

A matrix which contains many zero entries or very few non-zero entries is called as Sparse matrix.

In the **figure B** contains only 8 of 36 elements are nonzero and that is sparse.

Important Note:

A sparse matrix can be represented in 1-Dimension, 2- Dimension and 3- Dimensional array. When a sparse matrix is represented as a two-dimensional array as shown in **Figure B**, more space is wasted.

Example: consider the space requirements necessary to store a 1000 x 1000 matrix that has only 2000 non-zero elements. The corresponding two-dimensional array requires space for 1,000,000 elements. The better choice is by using a representation in which only the nonzero elements are stored.

Sparse Matrix Representation

- An element within a matrix can characterize by using the **triple <row,col,value>** This means that, an array of triples is used to represent a sparse matrix.
- Organize the triples so that the row indices are in ascending order.
- The operations should terminate, so we must know the number of rows and columns, and the number of nonzero elements in the matrix.

Implementation of the **Create** operation as below:

SparseMatrix Create(maxRow, maxCol) ::=

```
#define MAX_TERMS  101          /* maximum number of terms +1*/
typedef struct {
                    int     col;
                    int    row;
                    int value;
                }      term;
term a[MAX_TERMS];
```

- The below figure shows the representation of matrix in the array **"a" a[0].row** contains the number of rows, **a[0].col** contains the number of columns and **a[0].value** contains the total number of nonzero entries.
- Positions 1 through 8 store the triples representing the nonzero entries. The row index is in the field row, the column index is in the field col, and the value is in the field value. The triples are ordered by row and within rows bycolumns.

a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	-15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	-6
[7]	4	0	91
[8]	5	2	28

Fig (a): Sparse matrix stored as triple

b[0]	6	6	8
[1]	0	0	15
[2]	0	4	91
[3]	1	1	11
[4]	2	1	3
[5]	2	5	28
[6]	3	0	22
[7]	3	2	-6
[8]	5	0	-15

Fig (b): Transpose matrix stored as triple

Transposing a Matrix

To transpose a matrix, interchange the rows and columns. This means that each element **a[i][j]** in the original matrix becomes element **a[j][i]** in the transpose matrix.

A good algorithm for transposing a matrix:

```
for each row i
        take element <i, j, value> and store it as
        element <j, i, value> of the transpose;
```

If we process the original matrix by the **row indices** it is difficult to know exactly where to place element <j, i, value> in the transpose matrix until we processed all the elements that precede it.

This can be avoided by using the **column indices** to determine the placement of elements in the transpose matrix. This suggests the following algorithm:

```
for all elements in column j
        place element <i, j, value> in
        element <j, i, value>
```

The columns within each row of the transpose matrix will be arranged in ascending order. void

```
transpose (term a[], termb[])
{               /* b is set to the transpose of a  */
    int n, i, j, currentb;
    n = a[0].value;             /* total number of elements */
    b[0].row  = a[0].col;       /* rows in b = columns in a */
    b[0].col = a[0].row;        /* columns in b = rows in a */
    b[0].value = n;
    if (n > 0)
        { currentb = 1;
            for (i = 0; i < a[O].col; i++)
                for (j= 1; j<=n; j++)
                    if (a[j].col == i)
                    {
                        b[currentb].row   =   a[j].col;
                        b[currentb].col   =   a[j].row;
                        b[currentb].value = a[j].value;
                        currentb++;
                    }
        }
}
```

Transpose of a sparse matrix

STRINGS:

Each programming languages contains a character set that is used to communicate with the computer. The character set include the following:

Alphabet: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Digits: 0 1 2 3 4 5 6 7 8 9
Special characters: + - / * ( ) , . $ = ‘ _ (Blank space)

String: A finite sequence S of zero or more Characters is called string.

Length: The number of characters in a string is called length of string.

Empty or Null String: The string with zero characters.

Concatenation: Let S_1 and S2 be the strings. The string consisting of the characters of S_1 followed by the character S_2 is called Concatenation of S_1 and S_2.

Ex: ‘THE’ // ‘END’ = ‘THEEND’
‘THE’ // ‘ ’ // ‘END’ = ‘THE END’

Substring: A string Y is called substring of a string S if there exist string X and Z such that S = X // Y // Z
If X is an empty string, then Y is called an Initial substring of S, and Z is an empty string then Y is called a terminal substring of S.

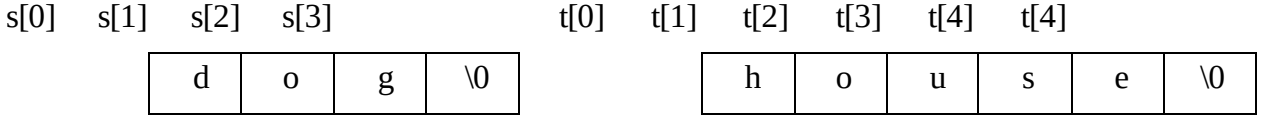
Ex: ‘BE OR NOT’ is a substring of ‘TO BE OR NOT TO BE’
‘THE’ is an initial substring of ‘THE END’

STRINGS IN C

In C, the strings are represented as character arrays terminated with the null character \0.

Declaration 1:

```
#define MAX_SIZE  100              /* maximum size of string */
char s[MAX_SIZE]  = {“dog”};
char t[MAX_SIZE] = {“house”};
```



The above figure shows how these strings would be represented internally in memory.

Declaration 2:

char s[ ] = {“dog”};
char t[ ] = {“house”};

Using these declarations, the C compiler will allocate just enough space to hold each word including the null character.

STORING STRINGS

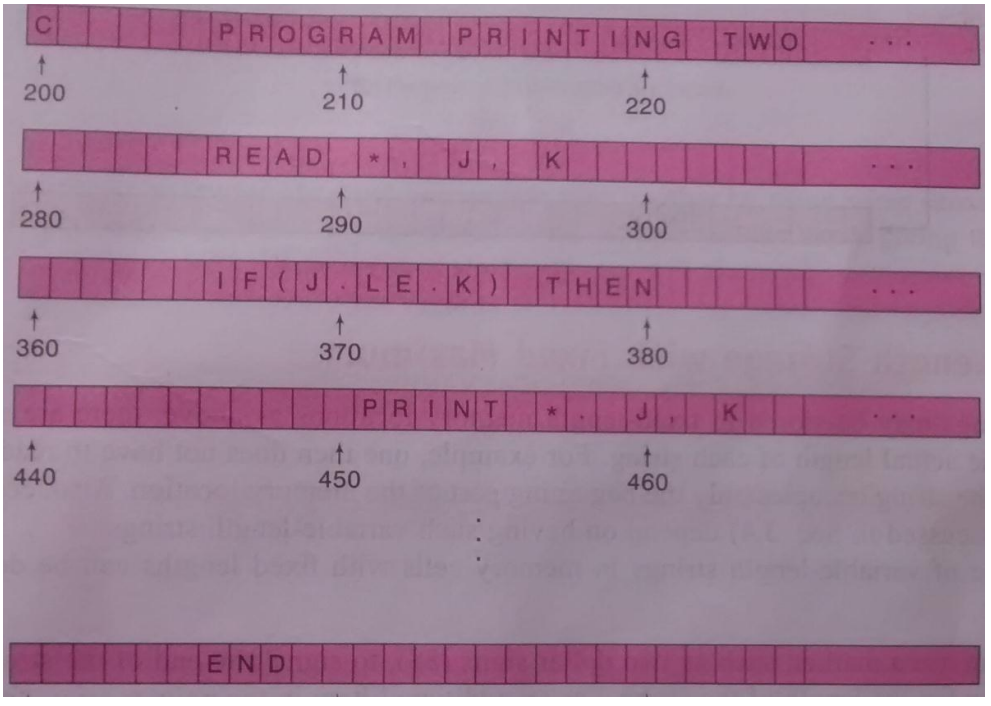
Strings are stored in three types of structures

1. Fixed length structures
2. Variable length structures with fixed maximum
3. Linked structures

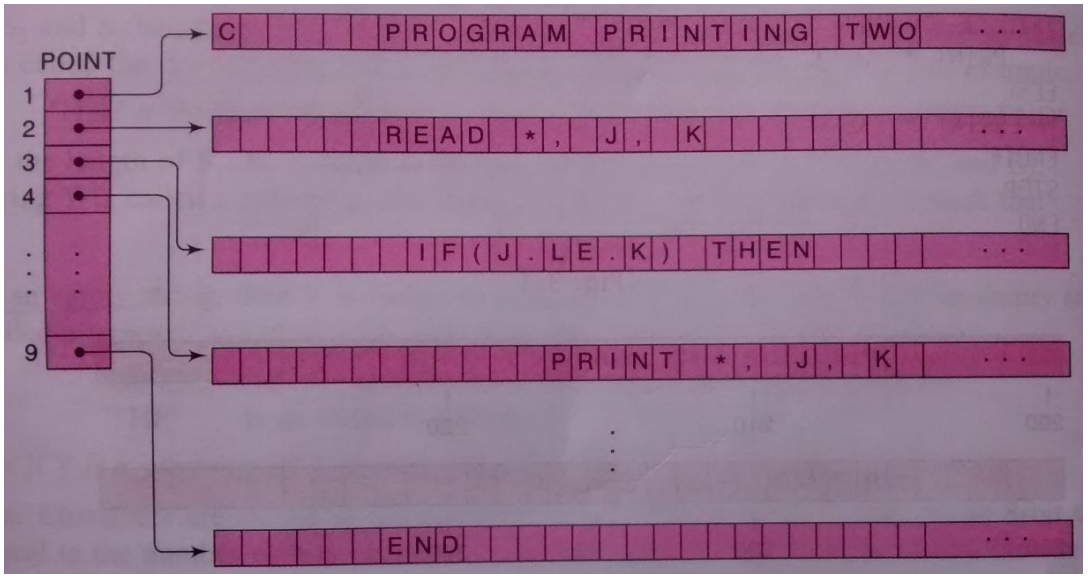
Record Oriented Fixed length storage:

In fixed length structures each line of print is viewed as a record, where all have the same length i.e., where each record accommodates the same number of characters.

Example: Suppose the input consists of the program. Using a record oriented, fixed length storage medium, the input data will appear in memory as pictured below.



Suppose, if new record needs to be inserted, then it requires that all succeeding records be moved to new memory location. This disadvantages can be easily remedied as shown in below figure.



That is, one can use a linear array POINT which gives the address of successive record, so that the records need not be stored in consecutive locations in memory. Inserting a new record will require only an updating of the array POINT.

The main advantages of this method are

1. The ease of accessing data from anygiven record
2. The ease of updating data in anygiven record (as long as the length of the new data does not exceed the record length)

The main disadvantages are

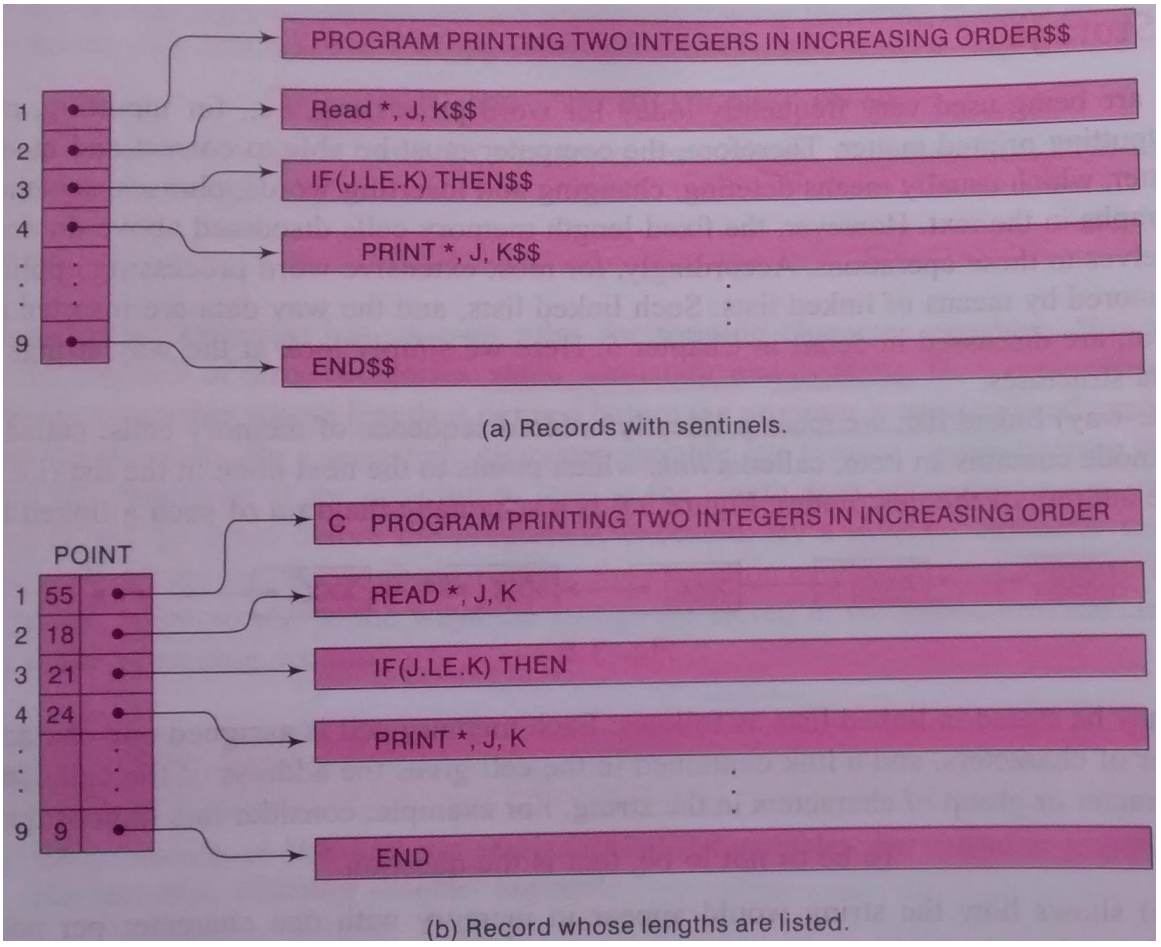
1. Time is wasted reading an entire record if most of the storage consists of inessential blank spaces.
2. Certain records may require more space thanavailable
3. When the correction consists of more or fewer characters than the original text, changing a misspelled word requires record to be changed.

Variable length structures with fixed maximum

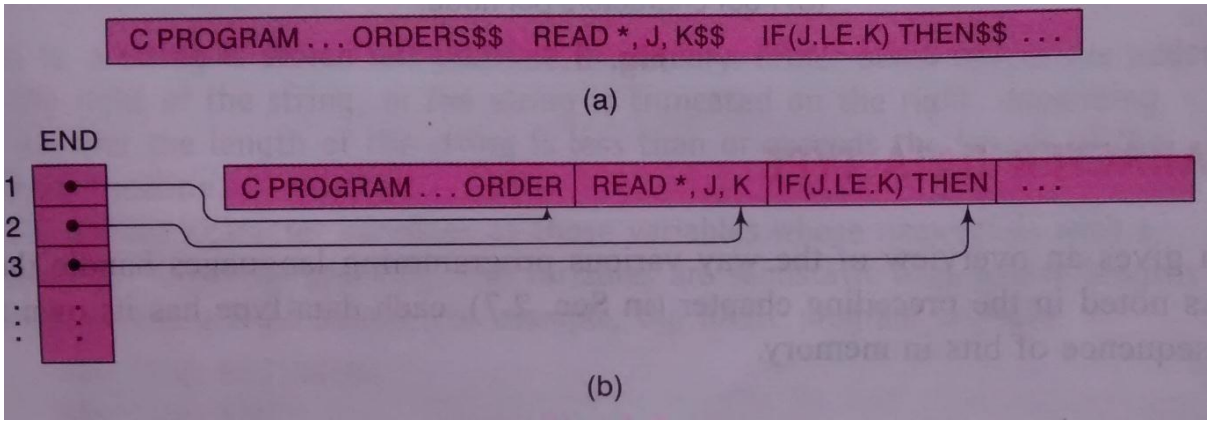
The storage of variable-length strings in memory cells with fixed lengths can be done in two general ways

1. One can use a marker, such as two dollar signs ($$), to signal the end of the string
2. One can list the length of the string—as an additional item in the pointer array

Example:



The other method to store strings one after another by using some separation marker, such as the two dollar sign ($$) or by using a pointer giving the location of the string.

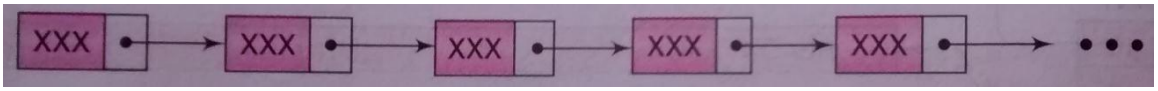


These ways of storing strings will save space and are sometimes used in secondary memory when records are relatively permanent and require little changes.

These types of methods of storage are usually inefficient when the strings and their lengths are frequently being changed.

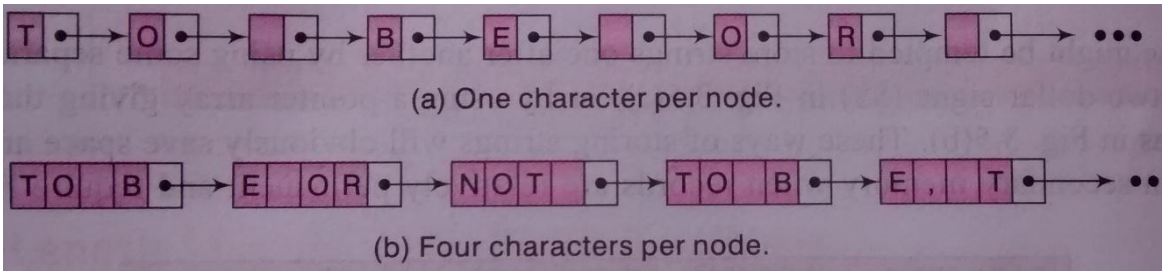
Linked Storage

- Most extensive word processing applications, strings are stored by means of linked lists.
- In a one way linked list, a linearly ordered sequence of memory cells called nodes, where each node contains an item called a ***link***, which points to the next node in the list, i.e., which consists the address of the nextnode.



Strings may be Stored in linked list as follows:
Each memory cell is assigned one character or a fixed number of characters and a link contained in the cell gives the address of the cell containing the next character or group of character in the string.

Ex: TO BE OR NOT TO BE



CHARACTER DATA TYPE

The various programming languages handles character data type in different ways.

Constants

Many programming languages denotes string constants by placing the string in either single or double quotation marks.

Ex: 'THE END'
"THE BEGINNING"

The string constants of length 7 and 13 characters respectively.

Variables

Each programming languages has its own rules for forming character variables. These variables fall into one of three categories

1. **Static:** In static character variable, whose length is defined before the program is executed and cannot change throughout the program

2. **Semi-static:** The length of the variable may vary during the execution of the program as long as the length does not exceed a maximum value determined by the program before the program is executed.
3. **Dynamic:** The length of the variable can change during the execution of the program.

STRING OPERATION

Substring

Accessing a substring from a given string requires three pieces of information:

(1) The name of the string or the string itself
(2) The position of the first character of the substring in the givenstring
(3) The length of the substring or the position of the last character of the substring.

Syntax: SUBSTRING (string, initial, length)

The syntax denote the substring of a string S beginning in a position K and having a length L.

Ex: SUBSTRING ('TO BE OR NOT TO BE', 4, 7) = 'BE OR N'
SUBSTRING ('THE END', 4, 4) = ' END'

Indexing

Indexing also called pattern matching, refers to finding the position where a string pattern P first appears in a given string text T. This operation is called INDEX

Syntax: INDEX (text, pattern)

If the pattern P does not appears in the text T, then INDEX is assigned the value 0.
The arguments "text" and "pattern" can be either string constant or string variable.

Concatenation

Let S_1 and S_2 be string. The concatenation of S_1 and S_2 which is denoted by S_1 // S_2, is the string consisting of the characters of S_1 followed by the character of S_2.

Ex:

(a) Suppose S_1 = 'MARK' and S_2= 'TWAIN' then
S_1 // S_2 = 'MARKTWAIN'

Concatenation is performed in C language using ***strcat*** function as shown below

strcat (S1, S2);

Concatenates string S1 and S2 and stores the result in S1

strcat ( ) function is part of the ***string.h*** header file; hence it must be included at the time of pre- processing

Length

The number of characters in a string is called its length.

Syntax: LENGTH (string)

Ex: LENGTH ('computer') = 8

String length is determined in C language using the ***strlen( )*** function, as shown below:

X = strlen ("sunrise");

strlen function returns an integer value 7 and assigns it to the variable X

Similar to ***strcat, strlen*** is also a part of string.h, hence the header file must be included at the time of pre-processing.

Function	Description
*char *strcat(char *dest, char *src)*	concatenate *dest* and *src* strings; return result in *dest*
*char *strncat(char *dest, char *src, int n)*	concatenate *dest* and *n* characters from *src*; return result in *dest*
*char *strcmp(char *str1, char *str2)*	compare two strings; return < 0 if *str1* < *str2*; 0 if *str1* = *str2*; > 0 if *str1* > *str2*
*char *strncmp(char *str1, char *str2, int n)*	compare first *n* characters return < 0 if *str1* < *str2*; 0 if *str1* = *str2*; > 1 if *str1* > *str2*
*char *strcpy(char *dest, char *src)*	copy *src* into *dest*; return *dest*
*char *strncpy(char *dest, char *src, int n)*	copy *n* characters from *src* string into *dest*; return *dest*;
*size_t strlen(char *s)*	return the length of a *s*
*char *strchr(char *s, int c)*	return pointer to the first occurrence of *c* in *s*; return *NULL* if not present
*char *strrchr(char *s, int c)*	return pointer to last occurrence of *c* in *s*; return *NULL* if not present
*char *strtok(char *s, char *delimiters)*	return a token from *s*; token is surrounded by *delimiters*
*char *strstr(char *s, char *pat)*	return pointer to start of *pat* in *s*
*size_t strspn(char *s, char *spanset)*	scan *s* for characters in *spanset*; return length of span
*size_t strcspn(char *s, char *spanset)*	scan *s* for characters not in *spanset*; return length of span
*char *strpbrk(char *s, char *spanset)*	scan *s* for characters in *spanset*; return pointer to first occurrence of a character from *spanset*

PATTERN MATCHING ALGORITHMS

Pattern matching is the problem of deciding whether or not a given string pattern P appears in a string text T. The length of P does not exceed the length of T.

First Pattern Matching Algorithm

- The first pattern matching algorithm is one in which comparison is done by a given pattern P with each of the substrings of T, moving from left to right, until a match is found.

W_K = SUBSTRING (T, K, LENGTH (P))

- Where, W_K denote the substring of T having the same length as P and beginning with the K^{th} character of T.
- First compare P, character by character, with the first substring, W_1. If all the characters are the same, then P = W_1 and so P appears in T and INDEX (T, P) = 1.
- Suppose it is found that some character of P is not the same as the corresponding character of W_1. Then $P \neq W_1$
- Immediately move on to the next substring, W_2 That is, compare P with W_2. If $P \neq W_2$ then compare P with W_3 and so on.
- The process stops, When P is matched with some substring W_K and so P appears in T and INDEX(T,P) = K or When all the W_K'S with no match and hence P does not appear in T.
- The maximum value MAX of the subscript K is equal to LENGTH(T) -LENGTH(P) +1.

Algorithm: (Pattern Matching)
P and T are strings with lengths R and S, and are stored as arrays with one character per element. This algorithm finds the INDEX of P in T.

1. [Initialize.] Set K: = 1 and MAX: = S - R + 1
2. Repeat Steps 3 to 5 while $K \leq$ MAX
3. Repeat for L = 1 to R: [Tests each character of P]
   If $P[L] \neq T[K + L - 1]$, then: Go to Step 5
   [End of inner loop.]
4. [Success.] Set INDEX = K, and Exit
5. Set K := K + 1
   [End of Step 2 outer loop]
6. [Failure.] Set INDEX = O
7. Exit

Observation of algorithms

- P is an r-character string and T is an s-character string
- Algorithm contains two loops, one inside the other. The outer loop runs through each successive R-character substring W_K = T[K] T[K + 1] ... T[K+R-l] of T.
- The inner loop compares P with W_K, character by character. If any character does not match, then control transfers to Step 5, which increases K and then leads to the next substring of T.
- If all the R characters of P do match those of some W_K then P appears in T and K is the INDEX of P in T.
- If the outer loop completes all of its cycles, then P does not appear in T and so INDEX = 0.

Complexity

The complexity of this pattern matching algorithm is equal to $O(n^2)$

Second Pattern Matching Algorithm

The second pattern matching algorithm uses a table which is derived from a particular pattern P but is independent of the text T.

For definiteness, suppose

P = aaba

This algorithm contains the table that is used for the pattern P = *aaba.*

The table is obtained as follows.

- Let Qi denote the initial substring of *P* of length *i*, hence Q0 = A, Q1 = *a,* Q2 = *a2,* Q3 = *aab,* Q4 = *aaba* = *P* (Here Q0 = A is the empty string.)
- The rows of the table are labeled by these initial substrings of *P,* excluding *P* itself.
- The columns of the table are labeled *a, b* and *x,* where *x* represents any character that doesn't appear in the pattern *P.*
- Let f be the function determined by the table; i.e., let *f(Qi, t)* denote the entry in the table in row Qi and column *t* (where *t* is any character). This entry $f(Q_i, t)$ is defined to be the largest Q that appears as a terminal substring in the string $(Q_i t)$ the concatenation of Qi and *t.*

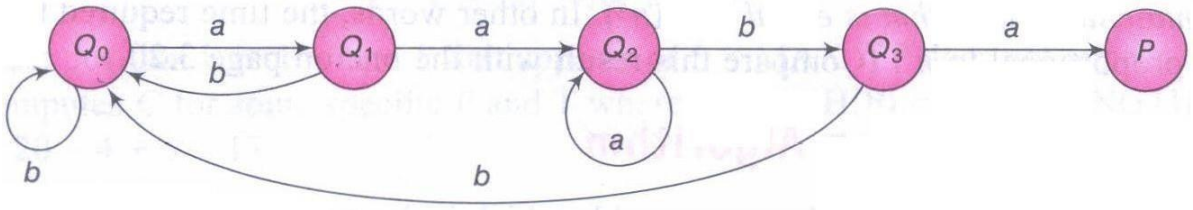
For example,

a2 is the largest Q that is a terminal substring of $Q2a = a^3$, so *f(Q2, a)* = Q2 A is the largest Q that is a terminal substring of Q1*b* = *ab,* so *f(Q1, b)* = Q0 *a* is the largest Q that is a terminal substring of *Q0a* = *a,* so *f(Q0, a)* = Q1

A is the largest Q that is a terminal substring of $Q3a = a^3bx$, so *f(Q3, x)* = Q0

	a	b	x
Q_0	Q_1	Q_0	Q_0
Q_1	Q_2	Q_0	Q_0
Q_2	Q_2	Q_3	Q_0
Q_3	P	Q_0	Q_0

(a) Pattern matching table



b Pattern matching graph

Although Q1 = *a* is a terminal substring of *Q2a* = a^3, we have *f(Q2, a)* = Q2 because Q2 is also a terminal substring of *Q2a* = a^3 and Q2 is larger than *Q1*. We note *that f(Qi, x)* = Q0 for any *Q*, since *x* does not appear in the pattern *P* Accordingly, the column corresponding to *x* is usually omitted from the table.

Pattern matching Graph

The graph is obtained with the table as follows.

First, a node in the graph corresponding to each initial substring Qi of *P.* The Q's are called the *states* of the system, and Q0 is called the *initial* state.

Second, there is an arrow (a directed edge) in the graph corresponding to each entry in the table. Specifically, if

$f(Q_i, t) = Q_j$

then there is an arrow labeled by the character *t* from *Qi* to *Qj*

For example, f(Q2, *b)* = *Q3* so there is an arrow labeled *b* from Q2 to *Q3*

For notational convenience, all arrows labeled *x* are omitted, which must lead to the initial state *Qo.*

The second pattern matching algorithm for the pattern P = *aaba.*

- Let T = T1 T2 T3 ... TN denote the n-character-string text which is searched for the pattern P. Beginning with the initial state Q0 and using the text T, wewill obtain a sequence of states S1, S2, S3, ... asfollows.
- Let S1 = Q0 and read the first character T1. The pair (S1, T1) yields a second state S2; that is, F(S1, T1) = S2, Read the next character T2, The pair (S2, T2) yields a state S3, and so

on.

There are two possibilities:

1. Some state SK = P, the desired pattern. In this case, P does appear in T and its index is K - LENGTH(P).
2. No state S1, S2, ... , SN +1 is equal to P. In this case, P does not appear in T.

Algorithm: (PATTERN MATCHING) The pattern matching table $F(Q_1, T)$ of a pattern P is in memory, and the input is an N-character string $T = T_1 T_2 T_3 \ldots\ldots T_N$. The algorithm finds the INDEX of P in T.

1. [Initialize] set K: =1 ans $S_1 = Q_0$
2. Repeat steps 3 to 5 while $S_K \neq P$ and $K \leq N$
3. Read T_K
4. Set S_{K+1} : = $F(S_K, T_K)$ [finds next state]
5. Set K: = K + 1 [Updates counter]

   [End of step 2 loop]
6. [Successful ?]

   If S_K = P, then

   INDEX = K – LENGTH (P)

   Else

   INDEX = 0

   [End of IF structure]
7. Exit.

STACKS AND QUEUES

STACKS

DEFINITION

"A **stack** is an **ordered list** in which insertions (pushes) and deletions (pops) are made at one end called the **top**."

Given a stack **S= (a_0, ... ,a_{n-1})**, where a_0 is the bottom element, a_{n-1} is the top element, and a_i is on top of element a_{i-1}, **0 < i < n**.

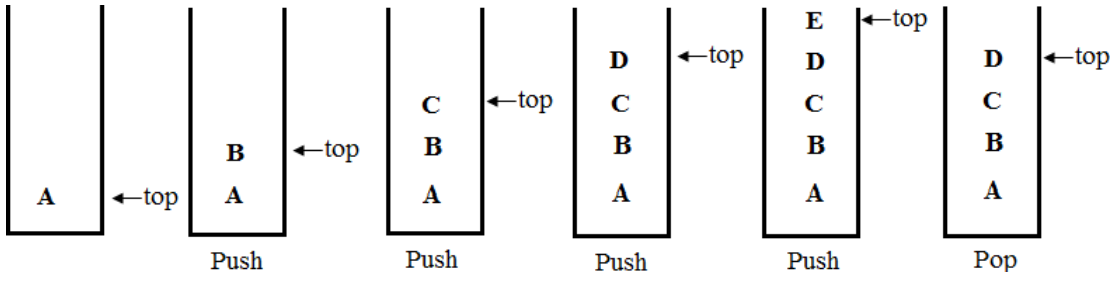


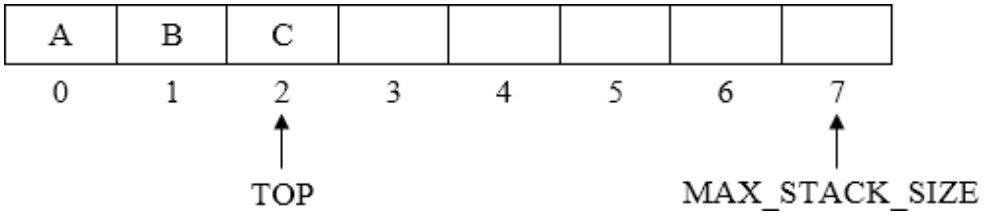
Figure: Inserting and deleting elements in a stack

As shown in above figure, the elements are added in the stack in the order A, B, C, D, E, then **E** is the first element that is deleted from the stack and the last element is deleted from stack is **A**. Figure illustrates this sequence of operations.
Since the last element inserted into a stack is the first element removed, a stack is also known as a **Last-In-First-Out (LIFO)** list.

ARRAY REPRESENTATION OF STACKS

- Stacks may be represented in the computer in various ways such as one-way linked list (Singly linked list) or linear array.
- Stacks are maintained by the two variables such as TOP and MAX_STACK_SIZE.
- TOP which contains the location of the top element in the stack. If TOP= -1, then it indicates stack is empty.
- MAX_STACK_SIZE which gives maximum number of elements that can be stored in stack.

Stack can represented using linear array as shown below



STACK OPERATIONS

Implementation of the stack operations as follows.

1. Stack Create

Stack CreateS(maxStackSize):::=

```
#define MAX_STACK_ SIZE 100 /* maximum stack size*/
typedef struct
        {
int key;
                        /* other fields */
        } element;

element stack[MAX_STACK_SIZE];
int top = -1;
```

The **element** which is used to insert or delete is specified as a structure that consists of only a **key** field.

2. Boolean IsEmpty(Stack)::= top < 0;

3. Boolean IsFull(Stack)::= top >= MAX_STACK_SIZE-1;

The I**sEmpty** and **IsFull** operations are simple, and is implemented directly in the program push and pop functions. Each of these functions assumes that the variables **stack** and **top** are global.

4. Push( )

Function **push** checks whether stack is full. If it is, it calls stackFull( ), which prints an error message and terminates execution. When the stack is not full, increment top and assign item to stack [top].

```
void push(element item)
        { /* add an item to the global stack */
                if (top >= MAX_STACK_SIZE-1)
                        stackFull();
                stack[++top] = item;
        }
```

5. Pop( )

Deleting an element from the stack is called pop operation. The element is deleted only from the top of the stack and only one element is deleted at a time.

```
element pop ( )
        { /*delete and return the top element from the stack */
                if (top == -1)
                        return stackEmpty(); /*returns an error key */
                return stack[top--];
         }
```

1. stackFull( )

The **stackFull** which prints an error message and terminates execution.

```
void stackFull()
{
                fprintf(stderr, "Stack is full, cannot add element");
                exit(EXIT_FAILURE);
        }
```

STACKS USING DYNAMIC ARRAYS

The array is used to implement stack, but the bound (MAX_STACK_ SIZE) should be known during compile time. The size of bound is impossible to alter during compilation hence this can be overcome by using dynamically allocated array for the elements and then increasing the size of array as needed.

Stack Operations using dynamic array

1. Stack CreateS( )::=

```
typedef struct
        {
                int key;        /* other fields */
        } element;
element *stack;
MALLOC(stack,   sizeof(*stack));
int capacity=  1;
int top= -1;
```

2. Boolean IsEmpty(Stack)::= top < 0;

3. Boolean IsFull(Stack)::= top >= capacity-1;

4. push()

Here the MAX_STACK_SIZE is replaced with **capacity**

```
void push(element item)
        {    /* add an item to the global stack */
                if (top >= capacity-1)
                        stackFull();
                stack[++top] = item;
        }
```

5. pop( )

In this function, no changes are made.

```
element pop ( )
        {      /* delete and return the top element from the stack */
                if (top == -1)
                        return stackEmpty(); /* returns an error key */
                return stack[top--];
        }
```

6. stackFull( )

The new code shown below, attempts to increase the **capacity** of the array **stack** so that new element can be added into the stack. Before increasing the capacity of an array, decide what the new capacity should be.
In array doubling, array capacity is doubled whenever it becomes necessary to increase the capacity of an array.

```
void stackFull()
{
        REALLOC (stack, 2*capacity*sizeof(*stack));
        capacity *= 2;
}
```

Stack full with array doubling

Analysis

In the **worst case**, the realloc function needs to allocate **2*capacity*sizeof (*stack)** bytes of memory and copy **capacity *sizeof (*stack))** bytes of memory from the old array into the new one. Under the assumptions that memory may be allocated in O(1) time and that a stack element can be copied in O(1) time, the time required by array doubling is O(capacity).
Initially, capacity is 1.

Suppose that, if all elements are pushed in stack and the capacity is 2k for some k, k>O, then the total time spent over all array doublings is **O ($\sum^{k}_{i=1} 2i$) = O(2^{k+l}) = O(2^{k})**.

Since the total number of pushes is more than **2k-1**, the total time spend in array doubling is **O(n)**, where **n** is the total number of pushes. Hence, even with the time spent on array doubling added in, the total run time of push over all **n** pushes is **O(n).**

STACK APPLICATIONS: POLISH NOTATION

Expressions: It is sequence of operators and operands that reduces to a single value after evaluation is called an expression.

X = a / b – c + d * e – a * c

In above expression contains operators (+, –, /, *) operands (a, b, c, d, e).

Expression can be represented in in different format such as

- Prefix Expression or Polish notation
- Infix Expression
- Postfix Expression or Reverse Polish notation

Infix Expression: In this expression, the binary operator is placed in-between the operand. The expression can be parenthesized or un- parenthesized.

Example: A + B

Here, **A** & **B** are operands and **+** is operand

Prefix or Polish Expression: In this expression, the operator appears before its operand.

Example: + A B

Here, **A** & **B** are operands and **+** is operand

Postfix or Reverse Polish Expression: In this expression, the operator appears after its operand.

Example: A B +

Here, **A** & **B** are operands and **+** is operand

Precedence of the operators

The first problem with understanding the meaning of expressions and statements is finding out the order in which the operations are performed.

Example: assume that a =4, b =c =2, d =e =3 in below expression

X = a / b – c + d * e – a * c

((4/2)-2) + (3*3)-(4*2)
=0+9-8
=1

OR

(4/ (2-2 +3)) *(3-4)*2
= (4/3) * (-1) * 2
= -2.66666

The first answer is picked most because division is carried out before subtraction, and multiplication before addition. If we wanted the second answer, write expression differently using parentheses to change the order of evaluation

X= ((a / (b – c + d)) * (e – a) * c

In C, there is a precedence hierarchy that determines the order in which operators are evaluated. Below figure contains the precedence hierarchy for C.

Token	Operator	Precedence	Associativity
( ) [ ] ⟶	function call array element struct or union member	17	left-to-right
-- ++	Increment, Decrement	16	left-to-right
--++ ! ~ -+ & * sizeof	decrement, increment logical not one's complement unary minus or plus address or indirection size (in bytes)	15	right-to-left
(type)	type cast	14	right-to-left
* / %	Multiplicative	13	left-to-right
+ -	binary add or subtract	12	left-to-right
<< >>	shift	11	left-to-right
> >= < <=	relational	10	left-to-right
== !=	equality	9	left-to-right
&	Bitwise and	8	left-to-right
^	bitwise exclusive or	7	left-to-right
|	Bitwise or	6	left-to-right
&&	logical and	5	left-to-right
||	logical or	4	left-to-right
?:	conditional	3	right-to-left
= += -= /= *= %= <<= >>= &= ^= |=	assignment	2	right -to-left
,	comma	1	left-to-right

- The operators are arranged from highest precedence to lowest. Operators with highest precedence are evaluated first.
- The associativity column indicates how to evaluate operators with the same precedence. For example, the multiplicative operators have left-to-right associativity. This means that the expression **a * b / c % d / e** is equivalent to **((((a * b) / c) % d) / e)**
- Parentheses are used to override precedence, and expressions are always evaluated from the innermost parenthesized expression first

INFIX TO POSTFIX CONVERSION

An algorithm to convert infix to a postfix expression as follows:

1. Fully parenthesize the expression.
2. Move all binary operators so that they replace their corresponding right parentheses.
3. Delete all parentheses.

Example: Infix expression: a/b -c +d*e -a*c
Fully parenthesized : ((((a/b)-c) + (d*e))-a*c))
: a b / e – d e * + a c *

Example [Parenthesized expression]: Parentheses make the translation process more difficult because the equivalent postfix expression will be parenthesis-free.
The expression **a*(b +c)*d** which results **abc +*d*** in postfix. Figure shows the translation process.

Token	Stack [0]	[1]	[2]	Top	Output
a				-1	a
*	*			0	a
(	*	(		1	a
b	*	(		1	ab
+	*	(	+	2	ab
c	*	(	+	2	abc
)	*			0	abc+
*	*			0	abc +*
d	*			0	abc +*d
eos	*			0	abc +*d*

- The analysis of the examples suggests a precedence-based scheme for stacking and unstacking operators.
- The left parenthesis complicates matters because it behaves like a low-precedence operator when it is on the stack and a high-precedence one when it is not. It is placed in the stack whenever it is found in the expression, but it is unstacked only when its matching right parenthesis is found.
- There are two types of precedence, **in-stack precedence (isp)** and **incoming precedence (icp)**.

The declarations that establish the precedence's are:

```
/* isp and icp arrays-index is value of precedence lparen rparen, plus, minus, times, divide,
mod, eos */
int isp[] = {0,19,12,12,13,13,13,0};
int icp[] = {20,19,12,12,13,13,13,0};
```

```
void postfix(void)
{
        char        symbol;
        precedence token;
        int n = 0,top = 0; /* place eos on stack */
        stack[0] = eos;
        for (token = getToken(&symbol, &n); token != eos; token =
            getToken(&symbol,& n ))
              {
                    if (token == operand)
                          printf("%c", symbol);
                    else if (token == rparen)
                          {
                                while (stack[top] != lparen)
                                      printToken(pop( ));
pop( );
}
else{
                          while(isp[stack[top]] >= icp[token])
                                printToken(pop());
                          push(token);
                       }
              }
        while((token = pop ())!= eos)
              printToken(token);
        printf("\n");
}
```

Program: Function to convert from infix to postfix

Analysis of postfix: Let **n** be the number of tokens in the expression. Θ (n) time is spent extracting tokens and outputting them. Time is spent in the **two while loops,** is Θ (n) as the number of tokens that get stacked and unstacked is linear in **n**. So, the complexity of function postfix is **Θ (n).**

EVALUATION OF POSTFIX EXPRESSION

- The evaluation process of postfix expression is simpler than the evaluation of infix expressions because there are no parentheses to consider.
- To evaluate an expression, make a single **left-to-right** scan of it. Place the operands on a stack until an operator is found. Then remove from the stack, the correct number of operands for the operator, perform the operation, and place the result back on the stack and continue this fashion until the end of the expression. We then remove the answer from the top of the stack.

Program: Function to evaluate a postfix expression

```
int eval(void)
    {
        precedence token;
        char symbol;
        int opl,op2, n=0;
        int top= -1;
        token = getToken(&symbol, &n);
        while(token! = eos)
            {
                if (token == operand)
                    push(symbol-'0'); /* stack insert */
                else {
                    op2 = pop(); /* stack delete */
                    opl = pop();
switch(token) {
                                case plus:      push(opl+op2);
                                break;
                                case minus:     push(opl-op2);
                                break;
                                case times:     push(opl*op2);
                                break;
                                case divide:    push(opl/op2);
                                break;
                                case mod:       push(opl%op2);
}

}
token = getToken(&symbol, &n);
}
        return pop(); /* return result */
    }
```

```
precedence getToken(char *symbol, int *n)
        {
                *symbol     =     expr[(*n)++];
                switch (*symbol)
                        {
                                case '(' : return lparen;
                                case ')' : return rparen;
                                case '+' : return plus;
                                case '-' : return minus;
                                case '/' : return divide;
                                case '*' : return times;
                                case '%' : return mod;
                                case ' ' : return eos;
                                default: return operand;
                        }
        }
```

Program: Function to get a token from the input string

- The function **eval ( )** contains the code to evaluate a postfix expression. Since an operand (symbol) is initially a character, convert it into a single digit integer.
- To convert use the statement, **symbol-'0'**. The statement takes the ASCII value of **symbol** and subtracts the ASCII value of '0', which is 48, from it. For example, suppose **symbol = '1.** The character '1' has an ASCII value of 49. Therefore, the statement **symbol-'0'** produces as result the number 1.
- The function **getToken( )**, obtain tokens from the expression string. If the token is an operand, convert it to a number and add it to the stack. Otherwise remove two operands from the stack, perform the specified operation, and place the result back on the stack. When the end of expression is reached, remove the result from the stack.

RECURSION

A recursive procedure

Suppose P is a procedure containing either a Call statement to itself or a Call statement to a second procedure that may eventually result in a Call statement back to the original procedure P. Then P is called a recursive procedure. So that the program will not continue to run indefinitely, a recursive procedure must have the following two properties:

1. There must be certain criteria, called base criteria, for which the procedure does not call itself.
2. Each time the procedure does call itself (directly or indirectly), it must be closer to the base criteria.

Recursive procedure with these two properties is said to be well-defined.

A recursive function

A function is said to be recursively defined if the function definition refers to itself. A recursive function must have the following two properties:

1. There must be certain arguments, called ***base values***, for which the function does not refer to itself.
2. Each time the function does refer to itself, the argument of the function must be closer to a ***base value***

A recursive function with these two properties is also said to be well-defined.

Factorial Function

"The product of the positive integers from 1 to n, is called "n factorial" and is denoted by n!"

$$n! = 1*2 * 3 \ldots (n - 2)*(n - 1)*n$$

It is also convenient to define 0! = 1, so that the function is defined for all nonnegative integers.

Definition: (Factorial Function)

a) If n = 0, then n! = 1.

b) If n > 0, then n! = n*(n - 1)!

Observe that this definition of n! is recursive, since it refers to itself when it uses (n - 1)!

(a) The value of n! is explicitly given when n = 0 (thus 0 is the base value)

(b) The value of n! for arbitrary *n* is defined in terms of a smaller value of *n* which is closer to the base value 0.

Tower of Hanoi

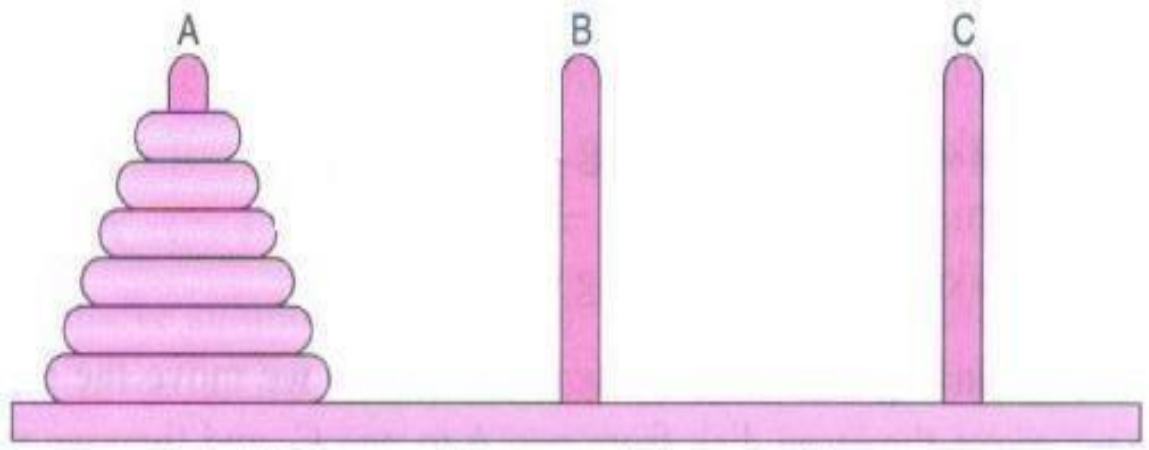
Problem description

Suppose three pegs, labeled A, Band C, are given, and suppose on peg A a finite number ***n*** of disks with decreasing size are placed.

The objective of the game is to move the disks from peg A to peg C using peg B as an auxiliary.

The rules of the game are as follows:

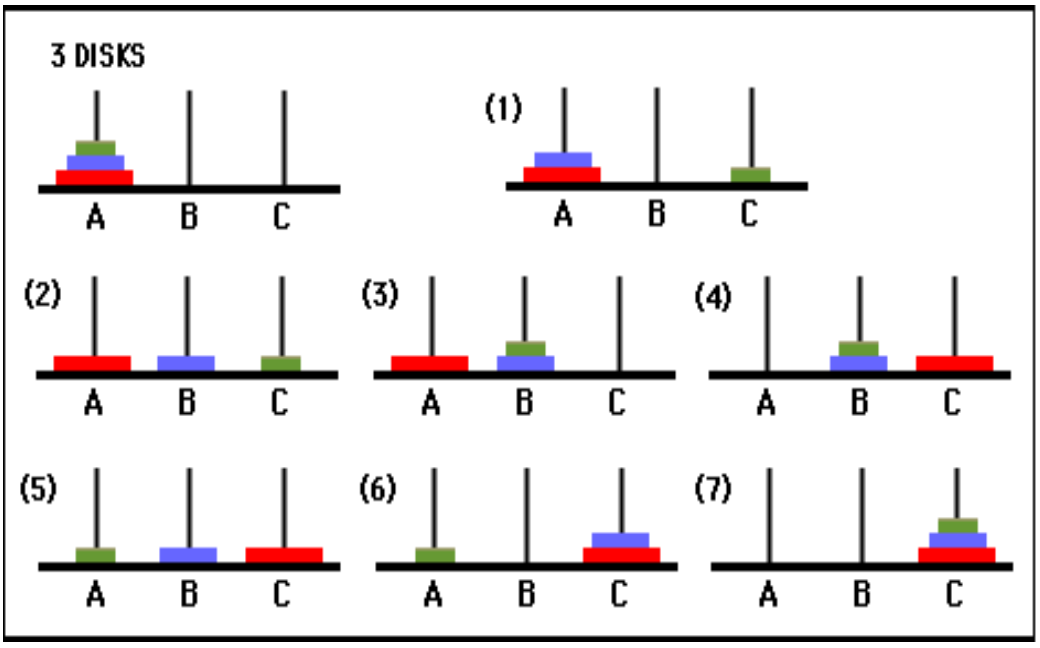
1. Only one disk may be moved at a time. Only the top disk on any peg may be moved to any other peg.
2. At no time can a larger disk be placed on a smaller disk.



Initial Setup of Towers of Hanoi with n = 6

Example: Towers of Hanoi problem for n = 3.

Solution: Observe that it consists of the following seven moves



1. Move top disk from peg A to peg C.
2. Move top disk from peg A to peg B.
3. Move top disk from peg C to peg B.
4. Move top disk from peg A to peg C.
5. Move top disk from peg B to peg A.
6. Move top disk from peg B to peg C.
7. Move top disk from peg A to peg C.

In other words,
n=3: A→C, A→B, C→B, A→C, B→A, B→C, A→C

For completeness, the solution to the Towers of Hanoi problem for n = 1 and n = 2
n=l: A→C
n=2: A→B, A→C, B→C

The Towers of Hanoi problem for n > 1 disks may be reduced to the following sub-problems:

(1) Move the top n - 1 disks from peg A to peg B
(2) Move the top disk from peg A to peg C: A→C.
(3) Move the top n - 1 disks from peg B to peg C.

The general notation

- TOWER (N, BEG, AUX, END) to denote a procedure which moves the top n disks from the initial peg BEG to the final peg END using the peg AUX as an auxiliary.

- When n = 1, the solution:
  TOWER (1, BEG, AUX, END) consists of the single instruction BEG→END

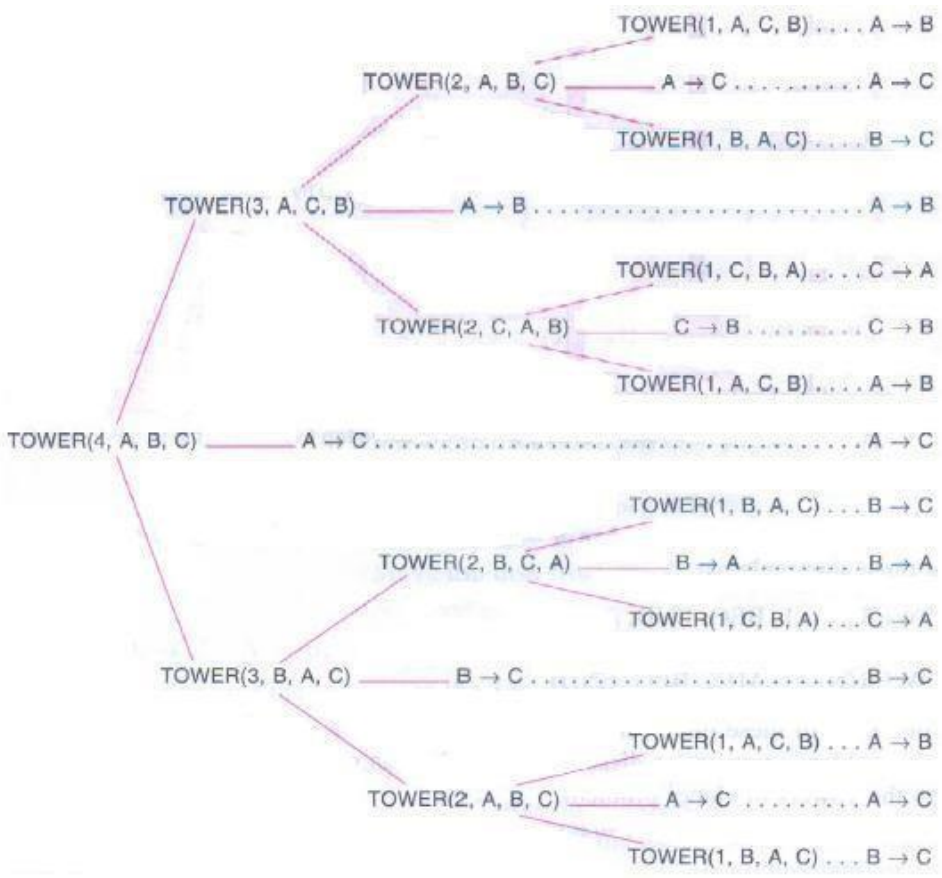
- When n > 1, the solution may be reduced to the solution of the following three sub-problems:
  (a) TOWER (N - I, BEG, END, AUX)
  (b) TOWER (l, BEG, AUX, END) or BEG → END
  (c) TOWER (N - I, AUX, BEG, END)

Procedure: TOWER (N, BEG, AUX, END)

This procedure gives a recursive solution to the Towers of Hanoi problem for N disks.

1. If N=l, then:
   (a) Write: BEG →END.
   (b) Return.
   [End of If structure.]
2. [Move N - 1 disks from peg BEG to peg AUX.]
   Call TOWER (N - 1, BEG, END, AUX).
3. Write: BEG →END.
4. [Move N - 1 disks from peg AUX to peg END.]
   Call TOWER (N - 1, AUX, BEG, END).
5. Return.

Example: Towers of Hanoi problem for n = 4



Ackermann function

The Ackermann function is a function with two arguments each of which can be assigned any nonnegative integer: 0, 1, 2, ...

Definition: (Ackermann Function)

(a) If $m = 0$, then $A(m, n) = n + 1$.

(b) If $m \neq 0$ but $n = 0$, then $A(m, n) = A(m - 1, 1)$

(c) If $m \neq 0$ and $n \neq 0$, then $A(m, n) = A(m - 1, A(m, n - 1))$