Faculty Name: Mrs. Madhu Nagaraj	Academic Year: 2024 - 2025

Department: Computer Science & Engineering

Course Code	Course Title	Core/Elective	Prerequisite	Contact Hours			Total Hrs/ Sessions
				L	T	P	
BCS304	Data Structure and Applications	Core	Basics of C programming concepts	3	-	-	40

Course Objective	1. To explain fundamentals of data structures and their applications. 2. To illustrate representation of Different data structures such as Stack, Queues, Linked Lists, Trees and Graphs. 3. To Design and Develop Solutions to problems using Linear Data Structures 4. To discuss applications of Nonlinear Data Structures in problem solving. 5. To introduce advanced Data structure concepts such as Hashing and Optimal Binary Search Trees

Topics Covered as per Syllabus

Module-1

INTRODUCTION TO DATA STRUCTURES: Data Structures, Classifications (Primitive & Non-Primitive), Data structure Operations Review of pointers and dynamic Memory Allocation, ARRAYS and STRUCTURES: Arrays, Dynamic Allocated Arrays, Structures and Unions, Polynomials, Sparse Matrices, representation of Multidimensional Arrays, Strings STACKS: Stacks, Stacks Using Dynamic Arrays, Evaluation and conversion of Expressions

Module-2

QUEUES: Queues, Circular Queues, Using Dynamic Arrays, Multiple Stacks and queues. LINKED LISTS: Singly Linked, Lists and Chains, Representing Chains in C, Linked Stacks and Queues, Polynomials

Module-3

LINKED LISTS: Additional List Operations, Sparse Matrices, Doubly Linked List. TREES: Introduction, Binary Trees, Binary Tree Traversals, Threaded Binary Trees.

Module-4

TREES(Cont..): Binary Search trees, Selection Trees, Forests, Representation of Disjoint sets, Counting Binary Trees, GRAPHS: The Graph Abstract Data Types, Elementary Graph Operations

Module-5

HASHING: Introduction, Static Hashing, Dynamic Hashing PRIORITY QUEUES: Single and double ended Priority Queues, Leftist Trees INTRODUCTION TO EFFICIENT BINARY SEARCH TREES: Optimal Binary Search Trees

Suggested Learning Resources:
Textbook: 1. Ellis Horowitz, Sartaj Sahni and Susan Anderson-Freed, Fundamentals of Data Structures in C, 2nd Ed, Universities Press, 2014
Reference Books: 1. Seymour Lipschutz, Data Structures Schaum's Outlines, Revised 1 st Ed, McGraw Hill, 2014. 2. Gilberg & Forouzan, Data Structures: A Pseudo-code approach with C, 2 nd Ed, Cengage Learning,2014. 3. Reema Thareja, Data Structures using C, 3 rd Ed, Oxford press, 2012. 4. Jean-Paul Tremblay & Paul G. Sorenson, An Introduction to Data Structures with Applications, 2nd Ed, McGraw Hill, 2013 5. A M Tenenbaum, Data Structures using C, PHI, 1989 6. Robert Kruse, Data Structures and Program Design in C, 2 nd Ed, PHI, 1996.
Web links and Video Lectures (e-Resources):
● http://elearning.vtu.ac.in/econtent/courses/video/CSE/06CS35.html ● https://nptel.ac.in/courses/106/105/106105171/ ● http://www.nptelvideos.in/2012/11/data-structures-and-algorithms.html ● https://www.youtube.com/watch?v=3Xo6P_V-qns&t=201s ● https://ds2-iiith.vlabs.ac.in/exp/selection-sort/index.html ● https://nptel.ac.in/courses/106/102/106102064/ ● https://ds1-iiith.vlabs.ac.in/exp/stacks-queues/index.html ● https://ds1-iiith.vlabs.ac.in/exp/linked-list/basics/overview.html ● https://ds1-iiith.vlabs.ac.in/List%20of%20experiments.html ● https://ds1-iiith.vlabs.ac.in/exp/tree-traversal/index.html ● https://ds1-iiith.vlabs.ac.in/exp/tree-traversal/depth-first-traversal/dft-practice.html ● https://infyspringboard.onwingspan.com/web/en/app/toc/lex_auth_01350159542807756812559/overview

Course Outcome	At the end of the course the student will be able to: **CO 1.** Explain different data structures and their applications. **CO 2.** Apply Arrays, Stacks and Queue data structures to solve the given problems. **CO 3.** Use the concept of linked list in problem solving. **CO 4.** Develop solutions using trees and graphs to model the real-world problem. **CO 5.** Explain the advanced Data Structures concepts such as Hashing Techniques and Optimal Binary Search Trees.

<u>**Assessment Details (both CIE and SEE)**</u>

The weightage of Continuous Internal Evaluation (CIE) is 50% and for Semester End Exam (SEE) is 50%. The minimum passing mark for the CIE is 40% of the maximum marks (20 marks out of 50) and for the SEE minimum passing mark is 35% of the maximum marks (18 out of 50 marks). A student shall be deemed to have satisfied the academic requirements and earned the credits allotted to each subject/ course if the student secures a minimum of 40% (40 marks out of 100) in the sum total of the CIE (Continuous Internal Evaluation) and SEE (Semester End Examination) taken together.

Continuous Internal Evaluation:

● For the Assignment component of the CIE, there are 25 marks and for the Internal Assessment Test component, there are 25 marks.

● The first test will be administered after 40-50% of the syllabus has been covered, and the second

test will be administered after 85-90% of the syllabus has been covered

● Any two assignment methods mentioned in the 22OB2.4, if an assignment is project-based then only one assignment for the course shall be planned. The teacher should not conduct two assignments at the end of the semester if two assignments are planned.

● For the course, CIE marks will be based on a scaled-down sum of two tests and other methods of assessment.

Internal Assessment Test question paper is designed to attain the different levels of Bloom's taxonomy as per the outcome defined for the course.

Semester-End Examination:

Theory SEE will be conducted by University as per the scheduled timetable, with common question papers for the course (duration 03 hours).

1. The question paper will have ten questions. Each question is set for 20 marks.

2. There will be 2 questions from each module. Each of the two questions under a module (with a maximum of 3 sub-questions), should have a mix of topics under that module.

3. The students have to answer 5 full questions, selecting one full question from each module.

4. Marks scored shall be proportionally reduced to 50 marks.

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 <u>two considerations</u>
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 maybe 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 way is 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 $a_1, a_2, a_3..............a_n$

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

Dept. Store	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 <u>rooted tree graph</u> or a <u>tree</u>.
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)
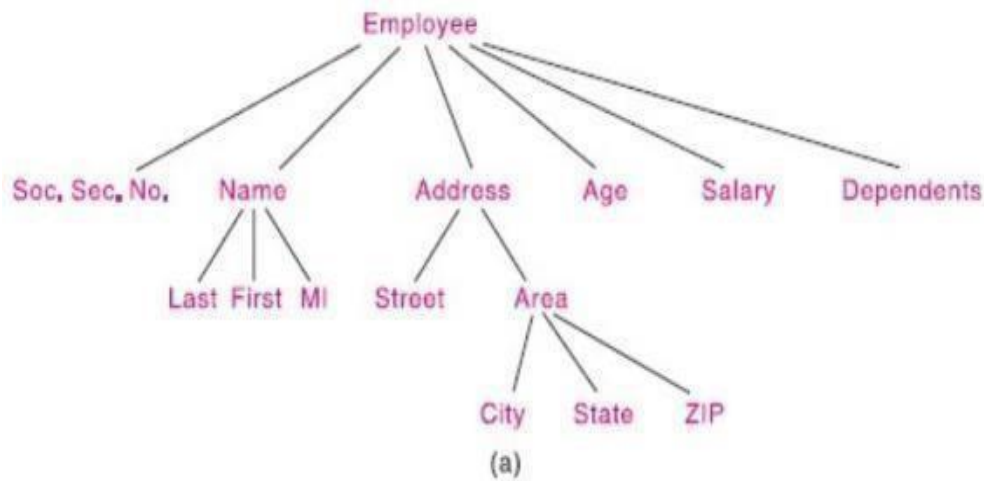
Another way of picturing such a tree structure is in terms of levels, as shown below

(b)
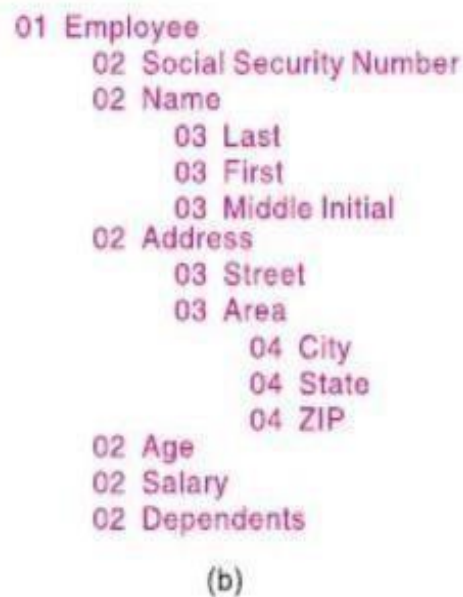
Some of the data structures are briefly described below.

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
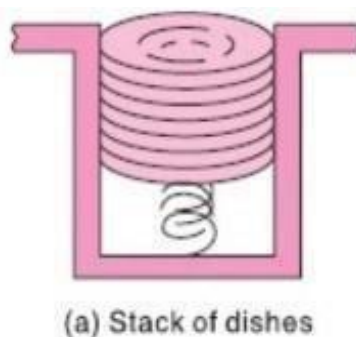
2. **Queue:** A queue, also called a first-in first-out (FIFO) system, is a linear list in which deletions can take place only at one end of the list, the "from" of the list, and insertions can take place only at the other end of the list, the "rear" of the list.

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
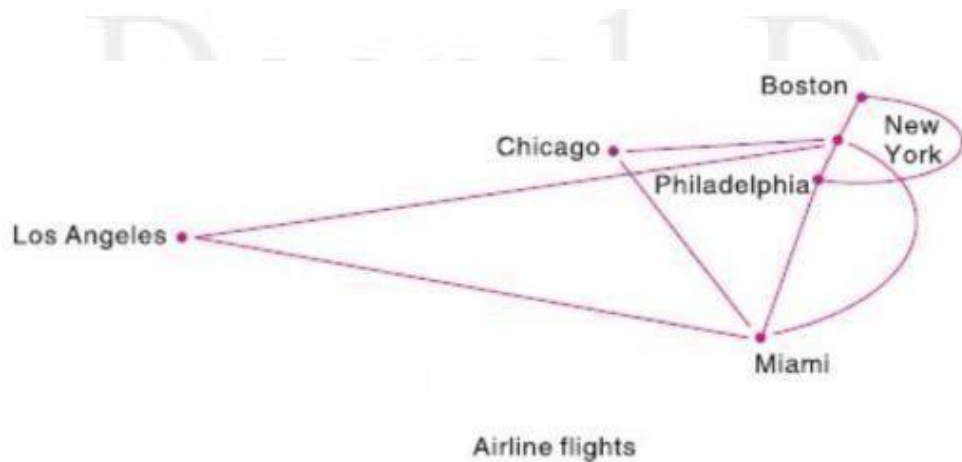
DATA STRUCTURES OPERATIONS

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.
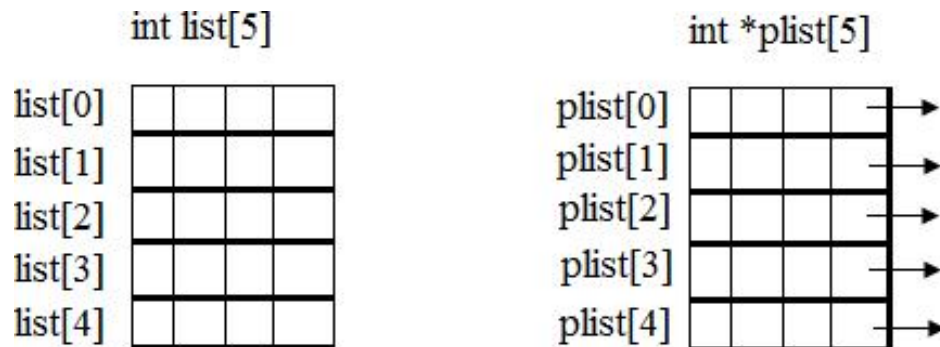

Implementation:

- 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]** the 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:
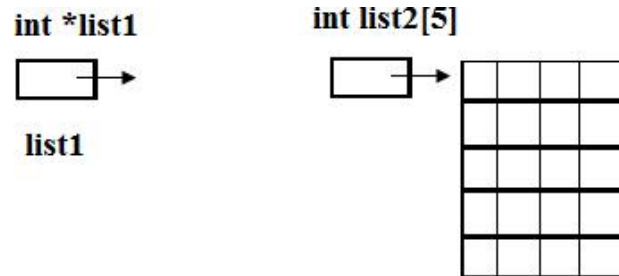
$$\text{list[i]} = \alpha + i * \text{sizeof (int)}$$

Where, α is base address.
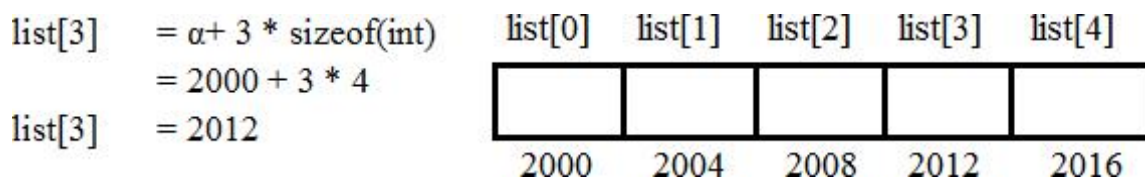
Difference between **int *list1; & int list2[5];**

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 pointer to **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:

<div align="center">

person1 = person2

</div>

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 member of 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.

<u>Consider as an example:</u>

```
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;
```
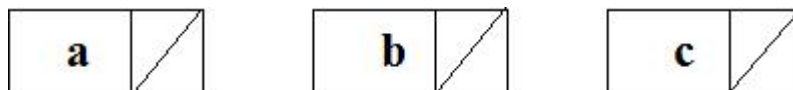

Structures item1, item2 and item3 each contain the data item **a, b,** and **c** respectively, and the null pointer. These structures can be attached together by replacing the **null link** field in item 2 with one that points to item 3 and by replacing the null link field in item 1 with one that points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

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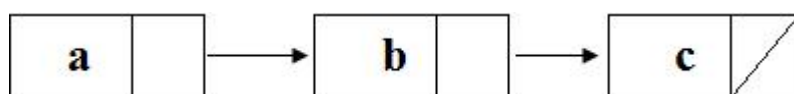
Unions:

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

Syntax: union{

                                data_type member 1;
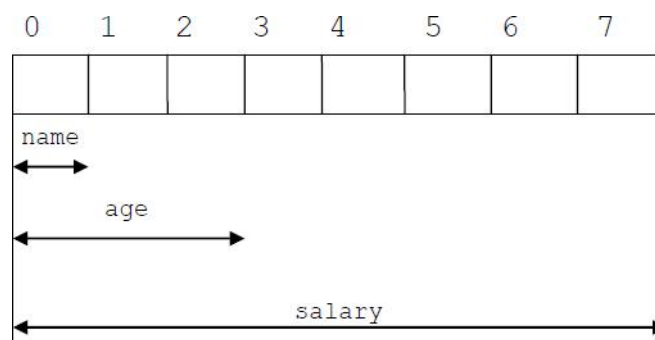   data_type member 2;
                                …………………
                                …………………
                                data_type member n;
                        }variable_name;

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);
```
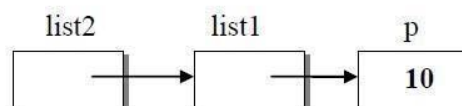

Output: 10   10   10

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 <u>length or size</u> 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,

$$\textbf{Length = UB – LB + 1}$$

When LB = 1,

$$\textbf{Length = UB}$$

Where,

        UB is the largest index called the <u>Upper Bound</u>
        LB is the smallest index, called the <u>Lower Bound</u>

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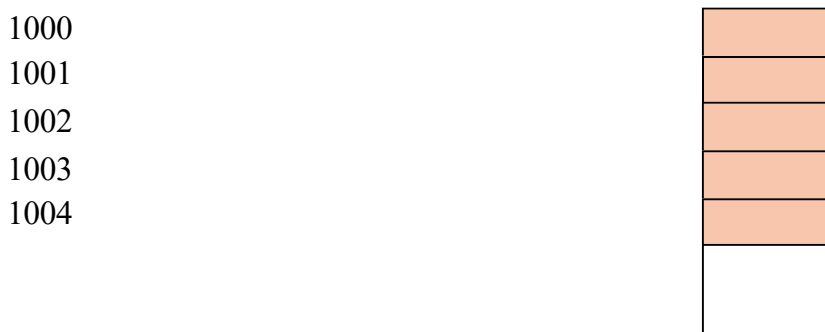
LOC (LA [K]) = address of the element LA [K] of the array LA

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];
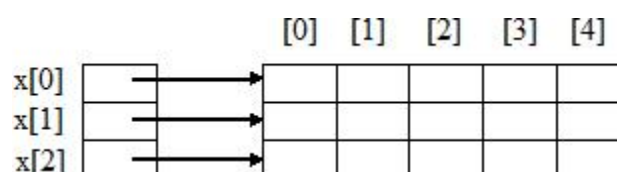
Array-of-arrays representation

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**.**

$$\textbf{x[i] = α+ i* sizeof(int)}$$
$$\textbf{x[j] = α+ j* sizeof(int)}$$
$$\textbf{x[i][j] = x[i]+ i* sizeof(int)}$$

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 $K \leq UB$
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 Kth 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 Kth 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).

	NAME		NAME		NAME		NAME
1	Brown	1	Brown	1	Brown	1	Brown
2	Davis	2	Davis	2	Davis	2	Ford
3	Johnson	3	Ford	3	Ford	3	Johnson
4	Smith	4	Johnson	4	Johnson	4	Smith
5	Wagner	5	Smith	5	Smith	5	Taylor
6		6	Wagner	6	Taylor	6	Wagner
7		7		7	Wagner	7	
8		8		8		8	
	(a)		(b)		(c)		(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

$$f(n) = (n - 1) + (n - 2) + ... + 2 + 1 = n(n-1) = n^2 \quad O(n) = O(n^2)$$

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.
- The search is said to be successful if ITEM does 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 for ITEM.]
   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.

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

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 DATAor 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 $m \cdot n$ data elements such that each element is specified by a pair of integers (such as J, K), called subscripts, with the property that
$$1 \leq J \leq m \text{ and } 1 \leq K \leq n$$

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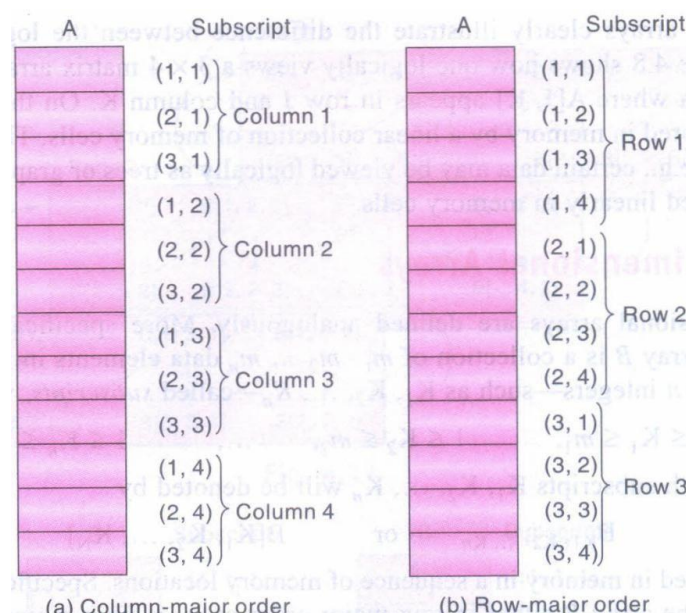
column-major   order,   or   (2)   row   by   row,   in   row-major   order

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.

B	Subscripts
	(1, 1, 1)
	(2, 1, 1)
	(1, 2, 1)
	(2, 2, 1)
	(1, 3, 1)
⋮	⋮
	(1, 4, 3)
	(2, 4, 3)

(a) Column-major order

B	Subscripts
	(1, 1, 1)
	(1, 1, 2)
	(1, 1, 3)
	(1, 2, 1)
	(1, 2, 2)
⋮	⋮
	(2, 4, 2)
	(2, 4, 3)

(b) Row-major order
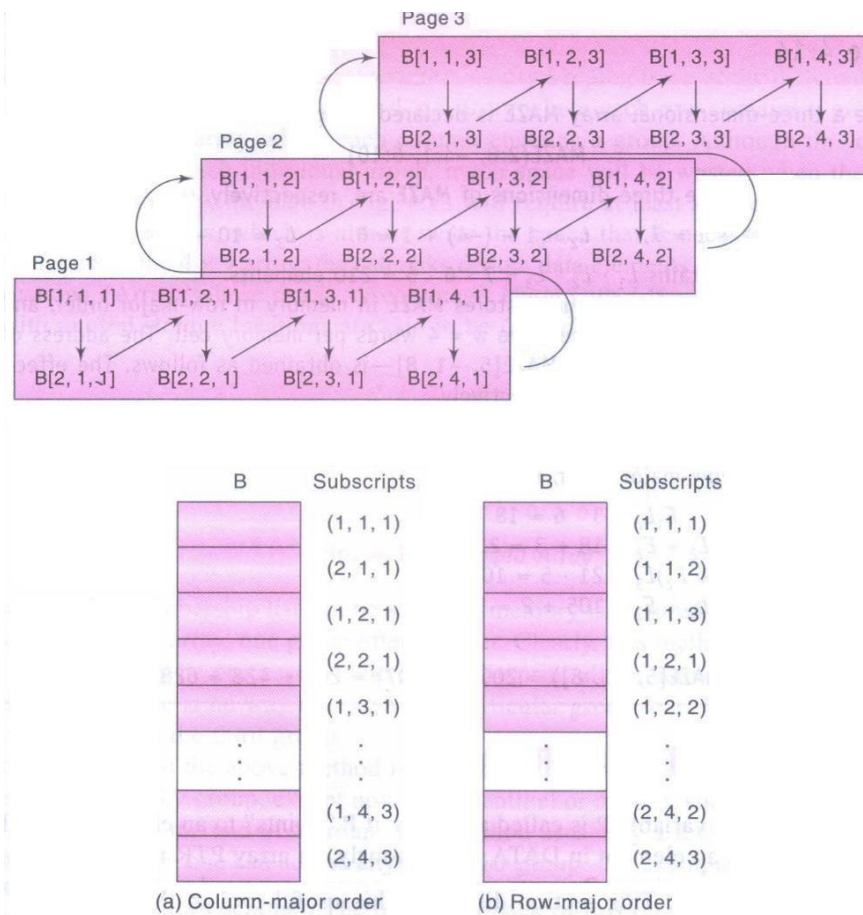
POLYNOMIALS

What is a polynomial?
"A polynomial is a sum of terms, where each term has a form 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,

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

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:

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

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
$A(x) = 2x^{1000} + 1$
$B(x) = x^4 + 10x^3 + 3x^2 + 1$

	startA	finishA	startB			finishB	avail
coef	2	1	1	10	3	1	
exp	1000	0	4	3	2	0	
	0	1	2	3	4	5	6

- 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**

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

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 by columns.

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

Fig (a): Sparse matrix stored as triple   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

STRING

BASIC TERMINOLOGY:

Each programming languages contains a <u>character set</u> 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 <u>Initial substring</u> of S, and Z is an empty string then Y is called a <u>terminal substring</u> 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"};
```

s[0]	s[1]	s[2]	s[3]
d	o	g	\0

t[0]	t[1]	t[2]	t[3]	t[4]	t[4]
h	o	u	s	e	\0

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.
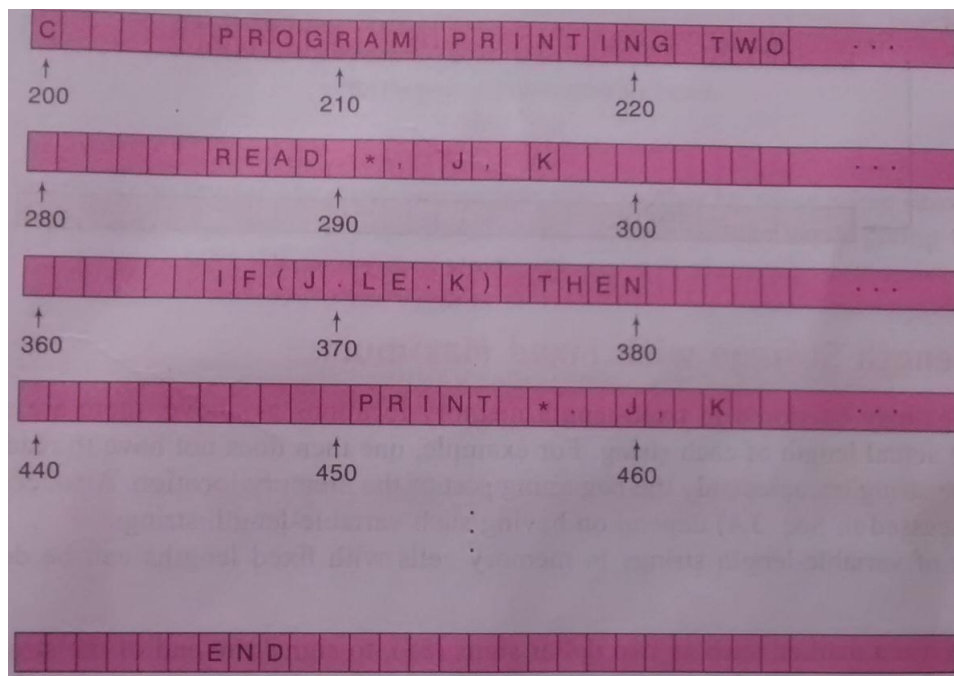
Suppose, if new record needs to be inserted, then it requires that all succeeding records be moved to new memory location. This disadvantages can be easily remedied as shown in below figure.
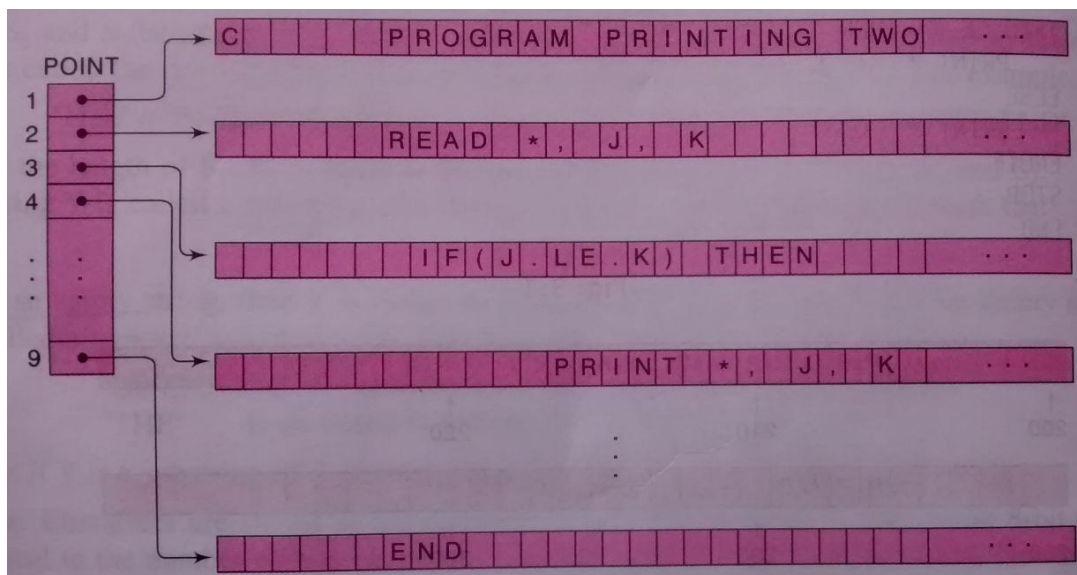
That is, one can use a linear array POINT which gives the address of successive record, so that the records need not be stored in consecutive locations in memory. Inserting a new record will require only an updating of the array POINT.

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:

(a) Records with sentinels.

(b) Record whose lengths are listed.

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.
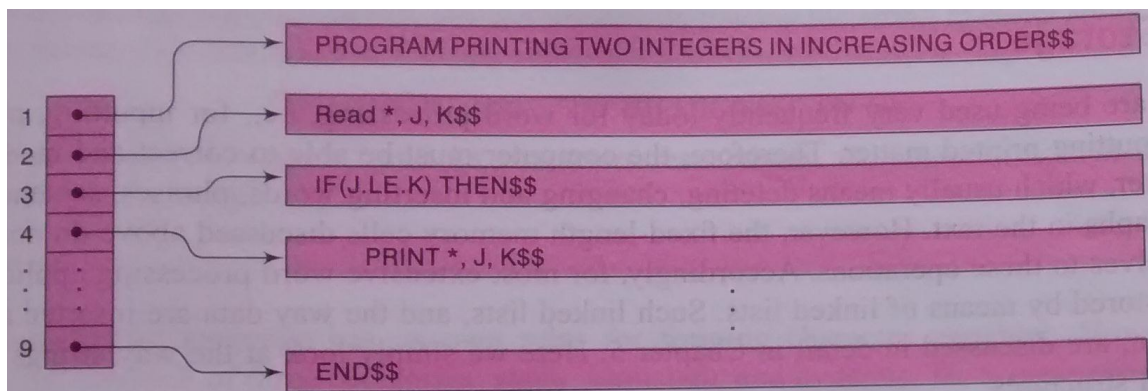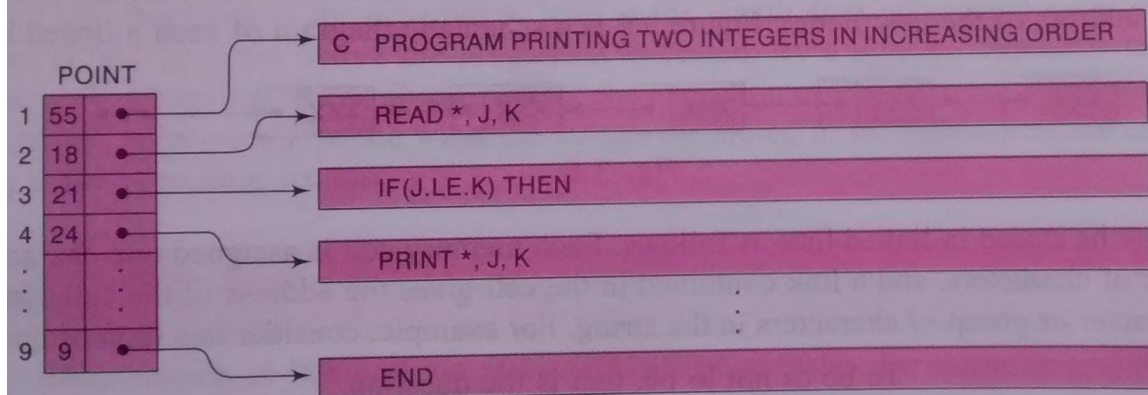
(a)

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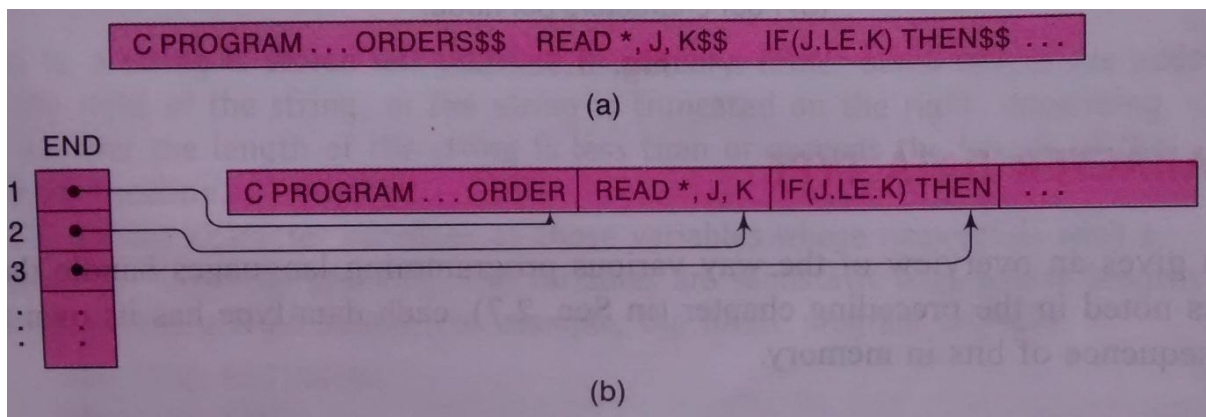
These ways of storing strings will save space and are sometimes used in secondary memory when records are relatively permanent and require little changes.

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, <u>a linearly ordered sequence of memory cells called nodes</u>, 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.
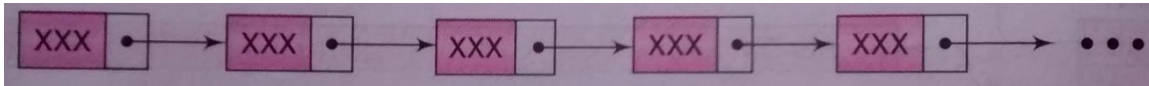
Strings may be Stored in linked list as follows:

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

(a) One character per node.

(b) Four characters per node.
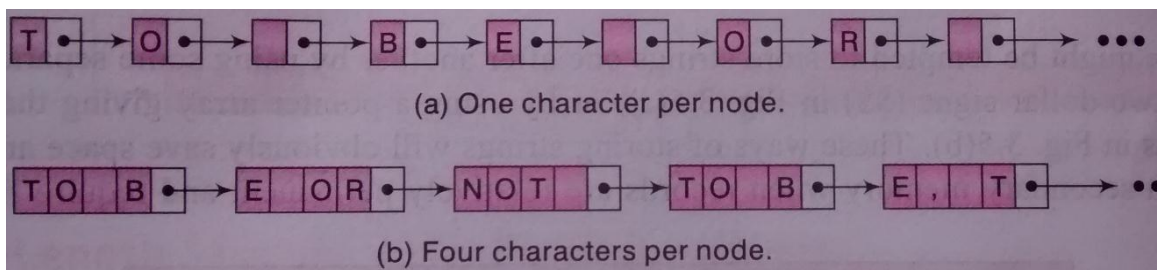
CHARACTER DATA TYPE

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 Kth 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 ≠ W_1
- Immediately move on to the next substring, W_2 That is, compare P with W_2. If P ≠ 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 ≤ MAX
3.   Repeat for L = 1 to R: [Tests each character of P]
           If P[L] ≠ T[K + L – l], 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

<u>Observation of algorithms</u>

- 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(Qi, 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 = a³,* 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 = a³bx,* 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
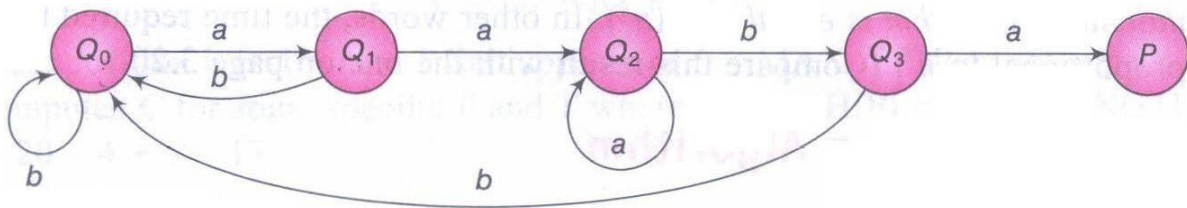
b Pattern matching graph

Although Q1 = *a* is a terminal substring of *Q2a = a³,* we have *f(Q2, a)* = Q2 because Q2 is also a terminal substring of *Q2a = a³* 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(Qᵢ, t) = Qⱼ

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, we will obtain a sequence of states S1, S2, S3, ... as follows.
- 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$_{i-1}$, $0 < i < n$.
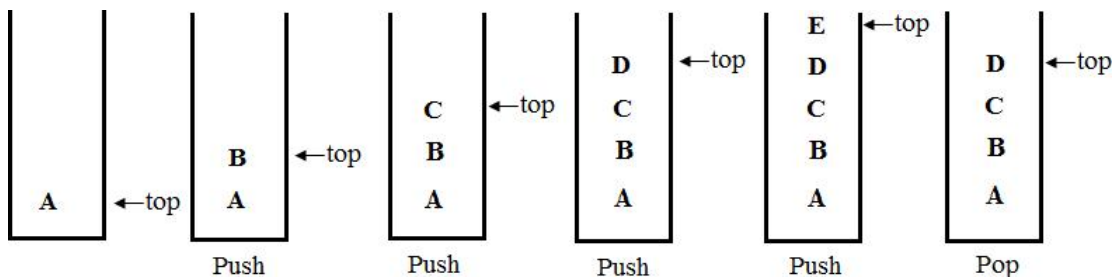
Figure: Inserting and deleting elements in a stack

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
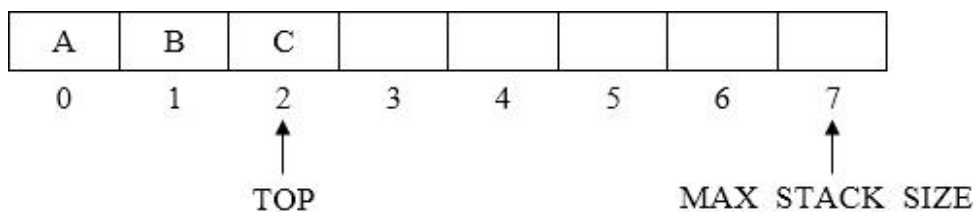
STACK OPERATIONS

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--];
}

6. 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 $O (\sum^k_{i=1} 2i) = O(2^{k+1}) = O(2^k)$.

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)              (4/ (2-2 +3)) *(3-4)*2
=0+9-8          OR                   = (4/3) * (-1) * 2
=1              = -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			Top	Output
	[0]	[1]	[2]		
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.

```
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 */
        }
```

Program: Function to evaluate a postfix expression

```
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!"

$$n! = 1*2 * 3 ... (n - 2)*(n - 1)*n$$

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.

The following are two procedures that each calculate n factorial .

1. **Using for loop:** This procedure evaluates N! using an iterative loop process

Procedure: FACTORIAL (FACT, N)

    This procedure calculates N! and returns the value in the variable FACT.
1. If N = 0, then: Set FACT: = 1, and Return.
2. Set FACT: = 1. [Initializes FACT for loop.]
3. Repeat for K = 1 to N.

Set FACT: = K*FACT.

       [End of loop.]
4. Return.

2. **Using recursive function:** This is a recursive procedure, since it contains a call to itself

Procedure: FACTORIAL (FACT, N)

This procedure calculates N! and returns the value in the variable FACT.
1. If N = 0, then: Set FACT: = 1, and Return.
2. Call FACTORIAL (FACT, N - 1).
3. Set FACT: = N*FACT.
4. Return.

GCD

The **greatest common divisor** (GCD) of two integers *m* and *n* is the greatest integer that divides both *m* and *n* with no remainder.
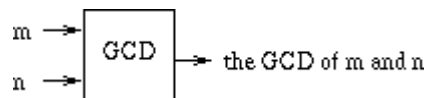
$$\text{For } m \geq n \geq 0, \ gcd \ (m, n) = \begin{cases} n & \text{If n divides m with no minimum} \\ gcd(n, remainder \ of \ \frac{m}{n}) & \text{Otherwise} \end{cases}$$

Procedure: GCD (M, N)
1. If (M % N) = 0, then set GCD=N and RETURN
2. Call GCD (N, M % N)
3. Return

Fibonacci Sequence

The Fibonacci sequence (usually denoted by F0, F1, F2.) is as follows:
0, 1, 1, 2,3,5,8, 13, 21, 34, 55

That is, F0 = 0 and F1 = 1 and each succeeding term is the sum of the two preceding terms.

Definition: (Fibonacci Sequence)
a) If n = 0 or n = 1, then $F_n = n$
b) If n > 1, then $F_n = F_{n-2} + F_{n-1}$

Here
(a) The base values are 0 and 1
(b) The value of F_n is defined in terms of smaller values of *n* which are closer to the base values.

A procedure for finding the nth term F_n of the Fibonacci sequence follows.

Procedure: FIBONACCI (FIB, N)

This procedure calculates F_N and returns the value in the first parameter FIB.
1. If N = 0 or N = 1, then: Set FIB: = N, and Return.
2. Call FIBONACCI (FIBA, N - 2).
3. Call FIBONACCI (FIBB, N - I).
4. Set FIB: = FIBA + FIBB.
5. Return.

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 maybe moved at a time. Only the top disk on anypeg maybe 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
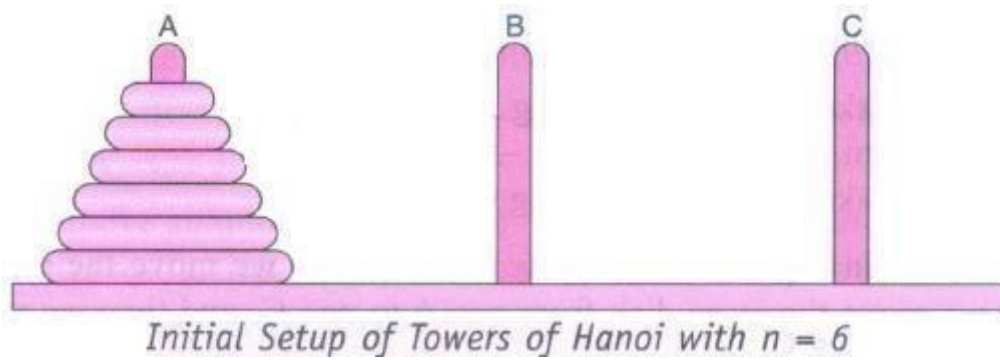
We write A→B to denote the instruction "Move top disk from peg A to peg B"

Example: Towers of Hanoi problem for n = 3.

Solution: Observe that it consists of the following seven moves
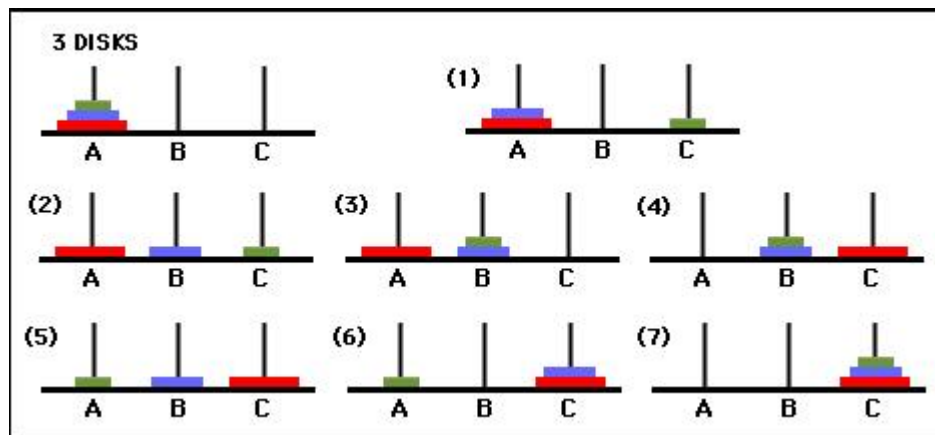
1. Move top disk from peg A to peg C.
2. Move top disk from peg A to peg B.
3. Move top disk from peg C to peg B.
4. Move top disk from peg A to peg C.
5. Move top disk from peg B to peg A.
6. Move top disk from peg B to peg C.
7. Move top disk from peg A to peg C.

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=1: 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


```
                                              TOWER(1, A, C, B) .... A → B
                          TOWER(2, A, B, C) _____ A → C ......... A → C
                                              TOWER(1, B, A, C) .... B → C
          TOWER(3, A, C, B) _____ A → B ....................... A → B
                                              TOWER(1, C, B, A) .... C → A
                          TOWER(2, C, A, B) _____ C → B ......... C → B
                                              TOWER(1, A, C, B) .... A → B
TOWER(4, A, B, C) _____ A → C ............................... A → C
                                              TOWER(1, B, A, C) ... B → C
                          TOWER(2, B, C, A) _____ B → A ......... B → A
                                              TOWER(1, C, B, A) ... C → A
          TOWER(3, B, A, C) _____ B → C ....................... B → C
                                              TOWER(1, A, C, B) ... A → B
                          TOWER(2, A, B, C) _____ A → C ......... A → C
                                              TOWER(1, B, A, C) ... B → C
```
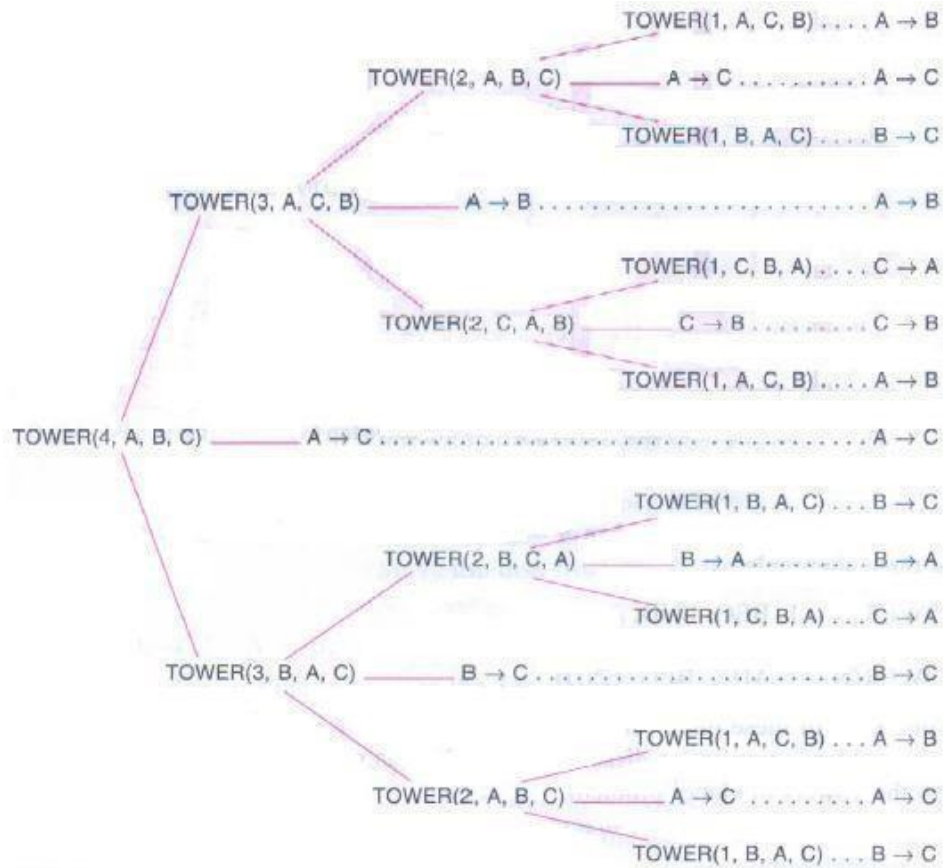
Ackermann function

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 ≠ 0 but n = 0, then A(m, n) = A(m - 1, 1)

(c) If m ≠ 0 and n ≠ 0, then A(m, n) = A(m - 1, A(m, n - 1))

MODULE 2: QUEUES

DEFINITION

- "A queue is an **ordered list** in which insertions (additions, pushes) and deletions (removals and pops) take place at different ends."
- The end at which new elements are added is called the **rear**, and that from which old elements are deleted is called the **front**.

If the elements are inserted A, B, C, D and E in this order, then A is the first element deleted from the queue. Since the first element inserted into a queue is the first element removed, queues are also known as **First-In-First-Out (FIFO) lists.**

QUEUE REPRESENTATION USING ARRAY

- Queues may be represented by one-way lists or linear arrays.
- Queues will be maintained by a linear array QUEUE and two pointer variables:
  FRONT-containing the location of the front element of the queue
  REAR-containing the location of the rear element of the queue.
- The condition FRONT = NULL will indicate that the queue is empty.

Figure indicates the way elements will be deleted from the queue and the way new elements will be added to the queue.

- Whenever an element is deleted from the queue, the value of FRONT is increased by 1; this can be implemented by the assignment FRONT := FRONT + 1
- When an element is added to the queue, the value of REAR is increased by 1; this can be implemented by the assignment REAR := REAR + 1
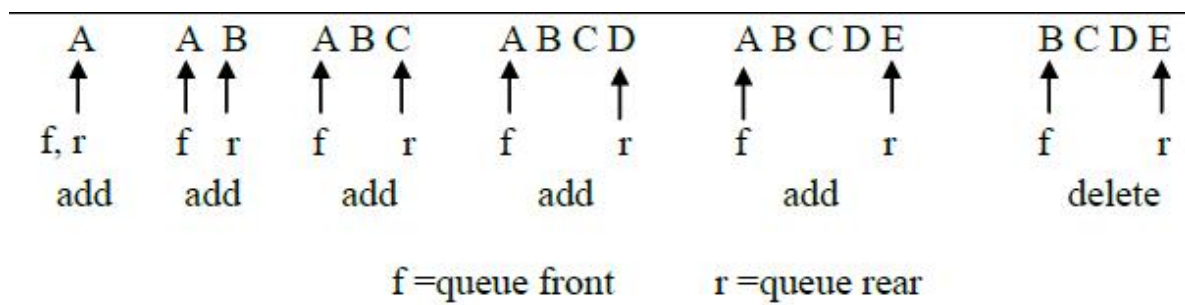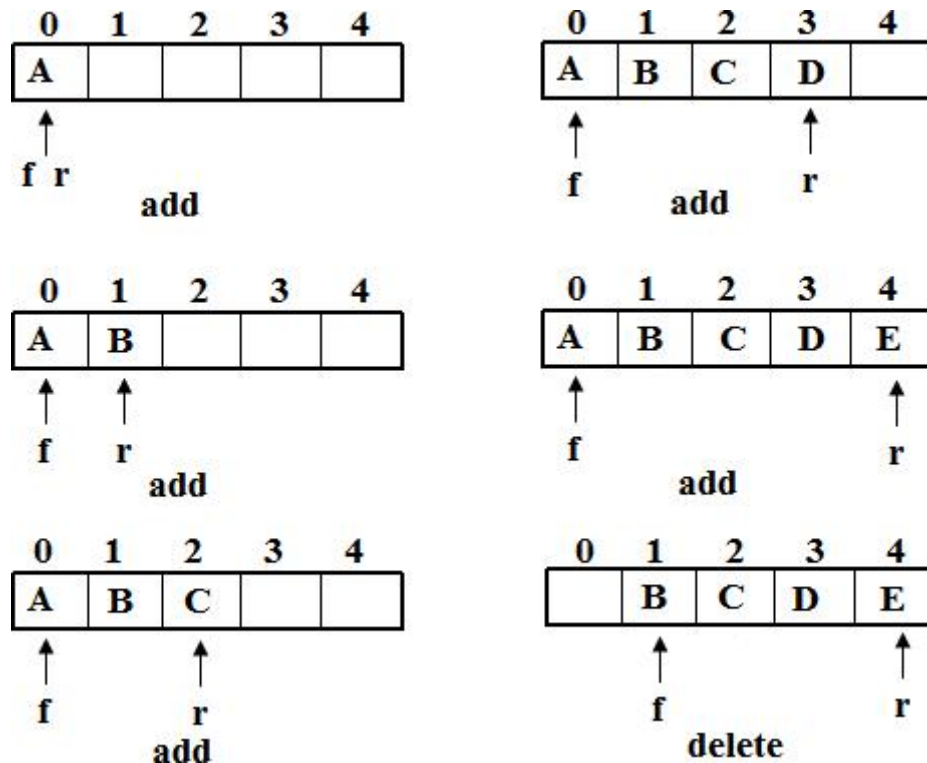
QUEUE OPERATIONS

Implementation of the queue operations as follows.

1. Queue Create

```
Queue CreateQ(maxQueueSize) ::=
        #define MAX_QUEUE_ SIZE  100          /* maximum queue size */
        typedef struct
            {
                    int key;                    /* other fields */
            } element;
        element  queue[MAX_QUEUE_ SIZE];
        int rear = -1;
        int front = -1;
```

2. Boolean IsEmptyQ(queue) ::= front==rear

3. Boolean IsFullQ(queue) ::= rear == MAX_QUEUE_SIZE-1

In the queue, two variables are used which are **front** and **rear.** The queue increments **rear** in addq( ) and **front** in delete( ). The function calls would be
addq (item); and **item =delete( );**

4. addq(item)

```
void addq(element item)
        {                       /* add an item to the queue */
            if (rear == MAX_QUEUE_SIZE-1)
                    queueFull();
            queue [++rear] = item;
        }
```

Program: Add to a queue

5. deleteq(   )

```
element deleteq()
        {                       /* remove element at the front of the queue */
            if (front == rear)
                    return queueEmpty( );           /* return an error key */
            return queue[++front];
        }
```
Program: Delete from a queue

6. queueFull( )
The **queueFull** function which prints an error message and terminates execution

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

Example: Job scheduling

- Queues are frequently used in creation of a **job queue** by an operating system. If the operating system does not use priorities, then the jobs are processed in the order they enter the system.
- Figure illustrates how an operating system process jobs using a sequential representation for its queue.

front	rear	Q[0]	Q[1]	Q[2]	Q[3]	Comments
-1	-1					Queue is empty
-1	0	J1				Job 1 is added
-1	1	J1	J2			Job 2 is added
-1	2	J1	J2	J3		Job 3 is added
0	2		J2	J3		Job 1 is deleted
1	2			J3		Job 2 is deleted

Figure: Insertion and deletion from a sequential queue

Drawback of Queue

When item enters and deleted from the queue, the queue gradually shifts to the right as shown in figure.
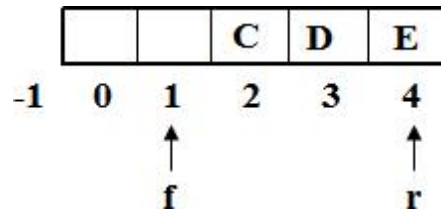
In this above situation, when we try to insert another item, which shows that the **queue is full** . This means that the **rear** index equals to MAX_QUEUE_SIZE -1. But even if the space is available at the front end, rear insertion cannot be done.

Overcome of Drawback using different methods

Method 1:

- When an item is deleted from the queue, move the entire queue to the **left** so that the first element is again at queue[0] and front is at **-1**. It should also recalculate **rear** so that it is correctly positioned.
- Shifting an array is very time-consuming when there are many elements in queue & queueFull has worst case complexity of **O(MAX_QUEUE_ SIZE)**

item A is deleted

item B is deleted
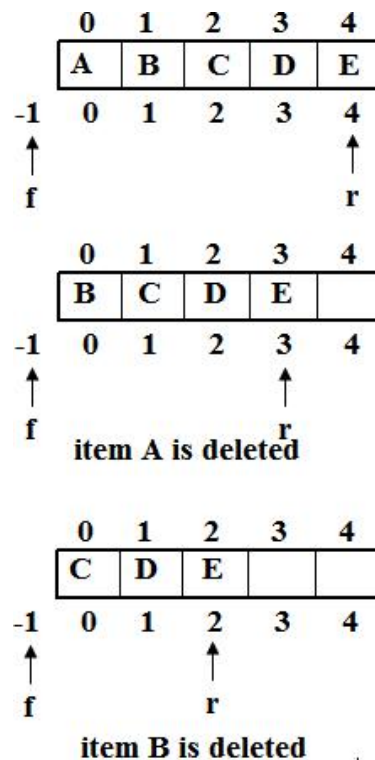
Method 2:

Circular Queue
- It is "The queue which **wrap around** the end of the array." The array positions are arranged in a circle.
- In this convention the variable **front** is changed. **front** variable points one position **counterclockwise** from the location of the front element in the queue. The convention for rear is unchanged.

CIRCULAR QUEUES

- It is "The queue which **wrap around** the end of the array." The array positions are arranged in a circle as shown in figure.
- In this convention the variable **front** is changed. **front** variable points one position **counterclockwise** from the location of the front element in the queue. The convention for rear is unchanged.

(a) Initial   (b) Addition   (c) Deletion
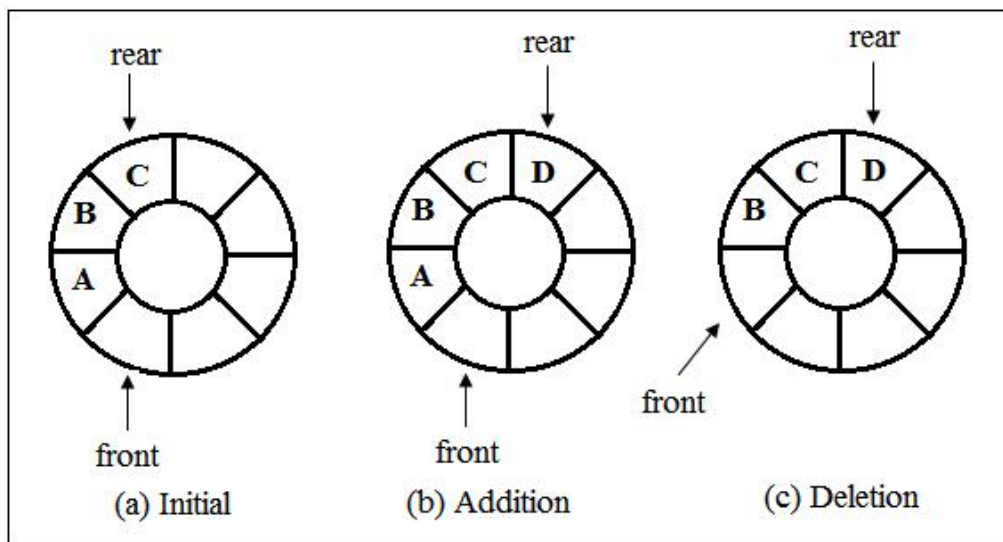
Implementation of Circular Queue Operations

- When the array is viewed as a circle, each array position has a **next** and a **previous** position. The position next to **MAX-QUEUE-SIZE -1** is **0**, and the position that precedes **0** is **MAX-QUEUE-SIZE -1**.
- When the queue **rear** is at MAX_QUEUE_SIZE-1, the next element is inserted at position **0**.
- In circular queue, the variables **front** and **rear** are moved from their current position to the next position in clockwise direction. This may be done using code

    if (rear == MAX_QUEUE_SIZE-1)

        rear = 0;

    else rear++;

Addition & Deletion

- To add an element, increment **rear** one position clockwise and insert at the new position. Here the **MAX_QUEUE_SIZE** is 8 and if all 8 elements are added into queue and that can be represented in below figure (a).

- To delete an element, increment **front** one position clockwise. The element **A** is deleted from queue and if we perform 6 deletions from the queue of Figure (b) in this fashion, then queue becomes empty and that **front=rear.**

- If the element **I** is added into the queue as in figure (c), then **rear** needs to increment by 1 and the value of rear is **8**. Since queue is circular, the next position should be 0 instead of 8.

  This can be done byusing the modulus operator, which computes remainders.

(rear +1) % MAX_QUEUE_SIZE

(a)   (b)   (c)
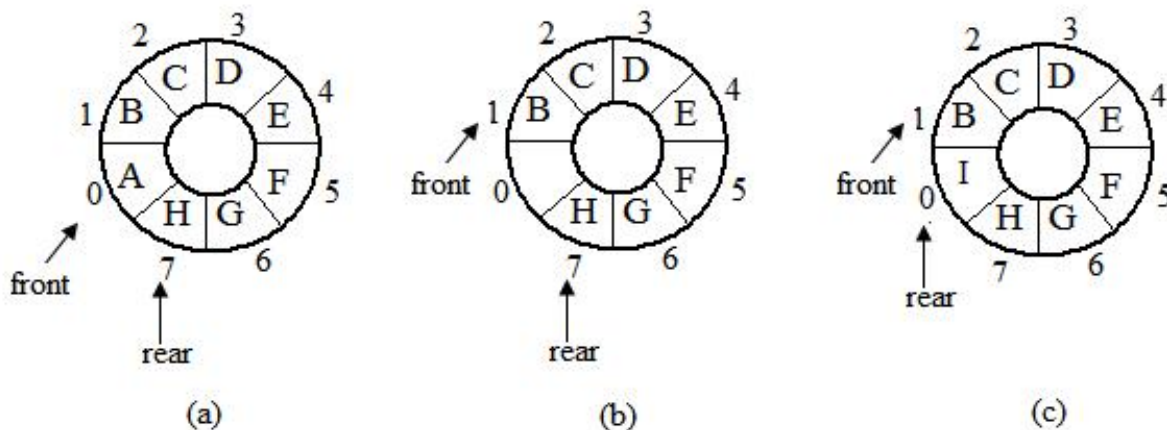
```
void addq(element item)
      {            /* add an item to the  queue  */
            rear = (rear +1) % MAX_QUEUE_SIZE;
            if (front == rear)
                  queueFull();            /* print error and exit */
            queue [rear] = item;

      }
```
Program: Add to a circular queue

```
element deleteq()
      {            /* remove front element from the queue */
            element item;
            if (front == rear)
                  return queueEmpty( );            /* return an error key */
            front = (front+1)% MAX_QUEUE_SIZE;
            return queue[front];

}
```
Program: Delete from a circular queue

Note:
- When queue becomes empty, then **front =rear.** When the queue becomes full and **front =rear.** It is difficult to distinguish between an empty and a full queue.
- To avoid the resulting confusion, increase the capacity of a queue just before it becomes full.

CIRCULAR QUEUES USING DYNAMIC ARRAYS

- A dynamically allocated array is used to hold the queue elements. Let **capacity** be the number of positions in the array queue.
- To add an element to a **full queue**, first increase the size of this array using a function **realloc**. As with dynamically allocated stacks, **array doubling** is used.

Consider the **full queue** of figure (a). This figure shows a queue with seven elements in an array whose capacity is 8. A circular queue is flatten out the array as in Figure (b).

(a) A full circular queue

(b) Flattened view of circular full queue
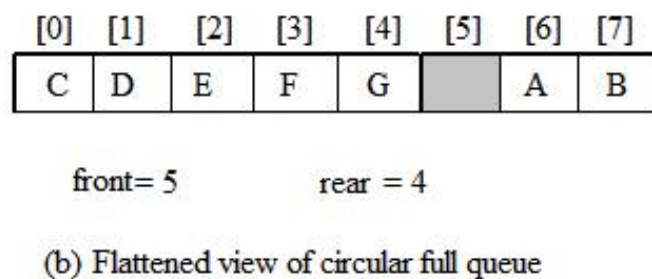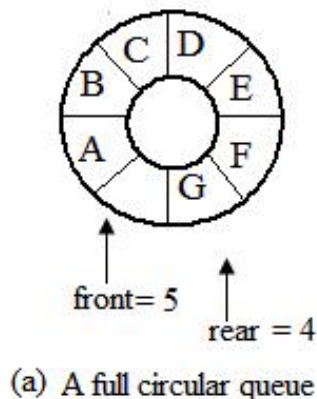
Figure (c) shows the array after array doubling by relloc

front= 5   rear = 4
(c) After array doubling
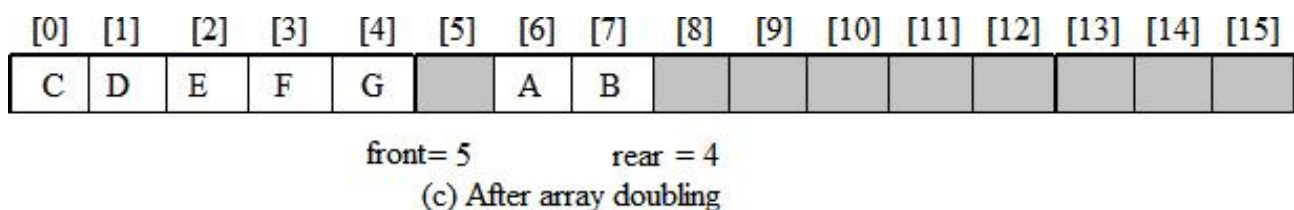
To get a proper circular queue configuration, slide the elements in the right segment (i.e., elements A and B) to the right end of the array as in figure (d)

front= 13   rear = 4
(d After shifting right segment
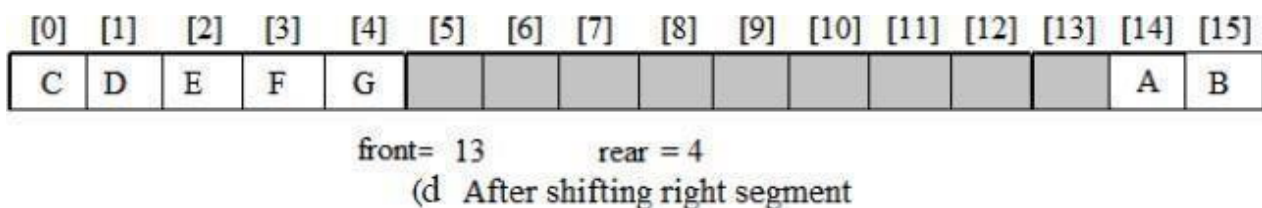
To obtain the configuration as shown in figure (e), follow the steps
1) Create a new array **newQueue** of twice the capacity.
2) Copy the second segment (i.e., the elements queue [front +1] through queue [capacity-1]) to positions in **newQueue** beginning at 0.
3) Copy the first segment (i.e., the elements queue [0] through queue [rear]) to positions in newQueue beginning at **capacity – front – 1.**

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
A	B	C	D	E	F	G									

front= 15   rear = 6

(e) Alternative configuration
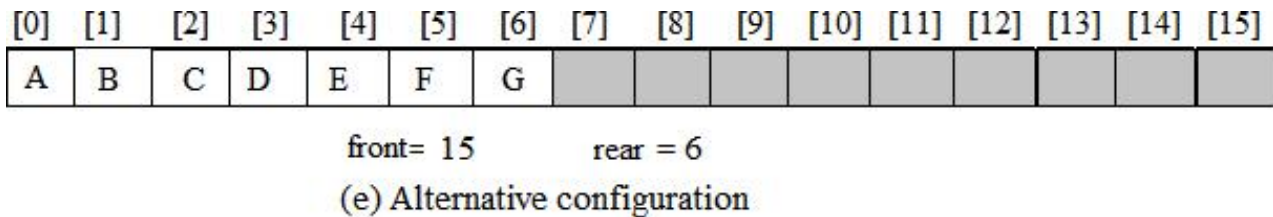
Below program gives the code to add to a circular queue using a dynamically allocated array.

```
void addq( elementitem)
        {               /* add an item to the queue
            rear  =  (rear  +1)  %  capacity;
            if(front == rear)
                    queueFull( );        /* double capacity */
            queue[rear] = item;
        }
```

Below program obtains the configuration of **figure (e)** and gives the code for queueFull. The function copy (a,b,c) copies elements from locations **a** through **b-1** to locations beginning at **c**.

```
void queueFull( )
        {                               /* allocate an array with twice the capacity */
            element *newQueue;
            MALLOC ( newQueue, 2 * capacity * sizeof(* queue));
            /* copy from queue to newQueue */

            int start = ( front + ) % capacity;
            if ( start < 2)                 /* no wrap around */
                    copy( queue+start, queue+start+capacity-1,newQueue);
            else
            {                               /* queue  wrap  around  */
                    copy(queue, queue+capacity, newQueue);
                    copy(queue, queue+rear+1, newQueue+capacity-start);
            }
        }
```

```
        /* switch to newQueue*/
        front  =  2*capacity  –  1;
        rear  =  capacity  –  2;
        capacity        *        =2;
        free(queue);
        queue= newQueue;

    }
```
Program: queueFull

DEQUEUES OR DEQUE

A deque (double ended queue) is a linear list in which elements can be added or removed at either end but not in the middle.

Representation

- Deque is maintained by a circular array DEQUE with pointers LEFT and RIGHT, which point to the two ends of the deque.
- Figure shows deque with 4 elements maintained in an array with $N = 8$ memory locations.
- The condition LEFT = NULL will be used to indicate that a deque is empty.

DEQUE

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

LEFT: 4   RIGHT: 7

There are two variations of a deque

1. **Input-restricted deque** is a deque which allows insertions at only one end of the list but allows deletions at both ends of thelist
2. **Output-restricted deque** is a deque which allows deletions at only one end of the list but allows insertions at both ends of the list.

PRIORITY QUEUES

A priority queue is a collection of elements such that each element has been assigned a priority and such that the order in which elements are deleted and processed comes from the following rules:

(1) An element of higher priority is processed before any element of lower priority.

(2) Twoelements with the same priority are processed according to the order in which theywere added to the queue.

A prototype of a priority queue is a timesharing system: programs of high priority are processed first, and programs with the same priority form a standard queue.

Representation of a Priority Queue

1. One-Way List Representation of a Priority Queue

One way to maintain a priority queue in memory is by means of a one-way list, as follows:

1. Each node in the list will contain three items of information: an information field INFO, a priority number PRN and a link number LINK.

2. A node X precedes a node Y in the list

a. When X has higher priority than Y

      b. When both have the same priority but X was added to the list before Y. This means that the order in the one-way list corresponds to the order of the priority queue.

Example:
- Below Figure shows the way the priority queue may appear in memory using linear arrays INFO, PRN and LINK with 7 elements.
- The diagram does not tell us whether BBB was added to the list before or after DDD. On the other hand, the diagram does tell us that BBB was inserted before CCC, because BBB and CCC have the same priority number and BBB appears before CCC in the list.
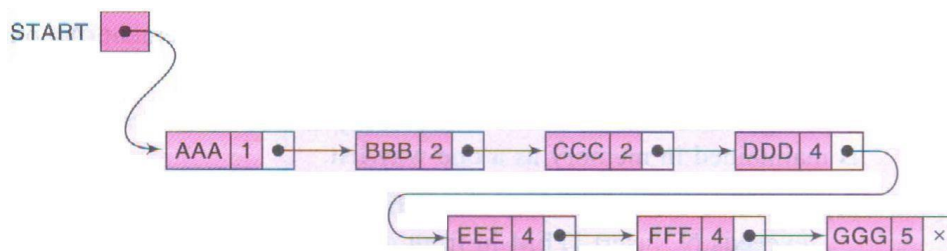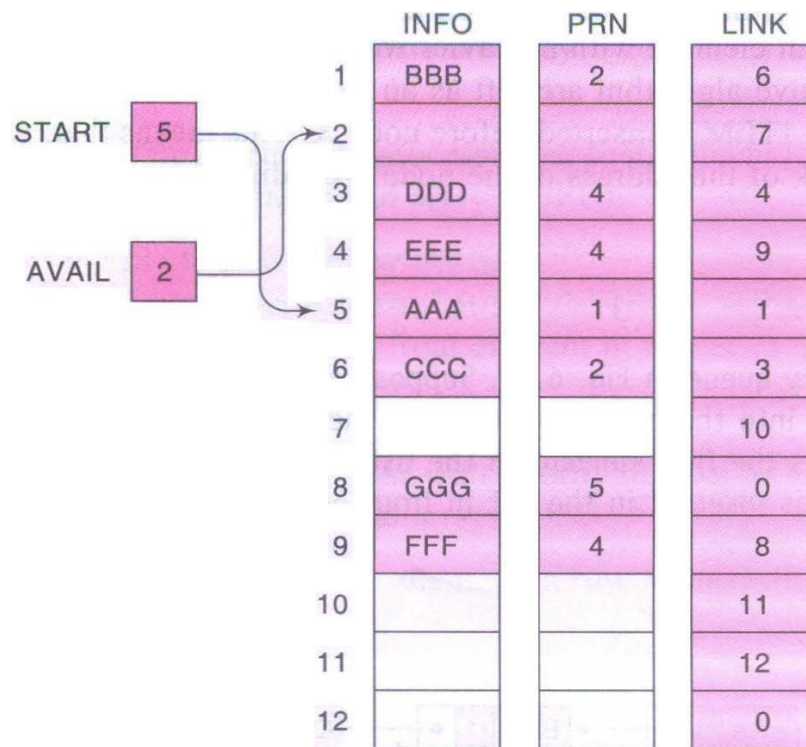
The main property of the one-way list representation of a priority queue is that the element in the queue that should be processed first always appears at the beginning of the one-way list. Accordingly, it is a very simple matter to delete and process an element from our priority queue.

Algorithm to deletes and processes the first element in a priority queue

Algorithm: This algorithm deletes and processes the first element in a priority queue which appears in memory as a one-way list.

1. Set ITEM:= INFO[START]   [This saves the data in the first node.]
2. Delete first node from the list.
3. Process ITEM.
4. Exit.

Algorithm to add an element to priority queue

Adding an element to priority queue is much more complicated than deleting an element from the queue, because we need to find the correct place to insert the element.

Algorithm: This algorithm adds an ITEM with priority number N to a priority queue which is maintained in memory as a one-way list.

1. Traverse the one-way list until finding a node X whose priority number exceeds N. Insert ITEM in front of node X.
2. If no such node is found, insert ITEM as the last element of the list.

The main difficulty in the algorithm comes from the fact that ITEM is inserted before node X. This means that, while traversing the list, one must also keep track of the address of the node preceding the node being accessed.

Example:
Consider the priority queue in Fig (a). Suppose an item XXX with priority number 2 is to be inserted into the queue. We traverse the list, comparing priority numbers.
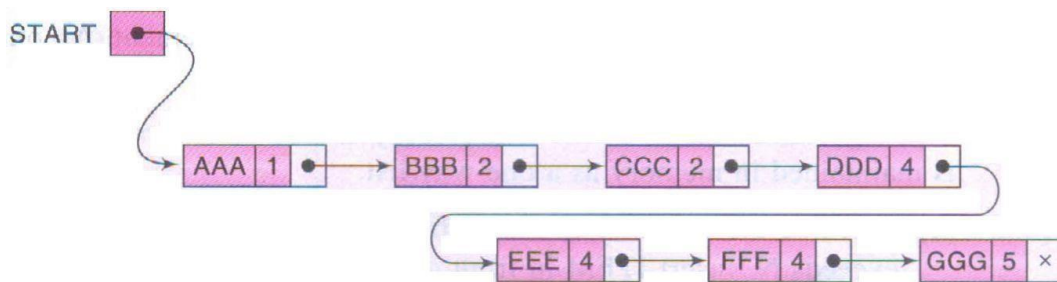
Fig (a)
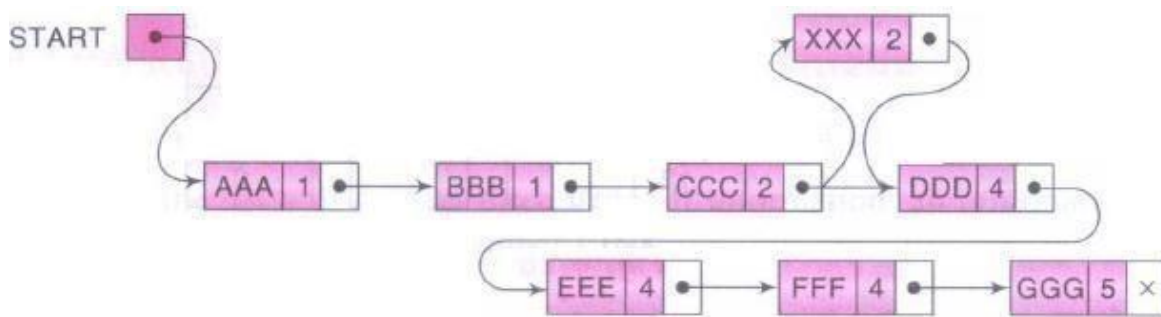
Fig(b)

Observe that DDD is the first element in the list whose priority number exceeds that of XXX. Hence XXX is inserted in the list in front of DDD, as pictured in Fig(b).

Observe that XXX comes after BBB and CCC, which have the same priority as XXX. Suppose now that an element is to be deleted from the queue. It will be AAA, the first element in the List. Assuming no other insertions, the next element to be deleted will be BBB, then CCC, then XXX, and so on.

Array Representation ofa Priority Queue

- Another way to maintain a priority queue in memory is to use a separate queue for each level of priority (or for each priority number).
- Each such queue will appear in its own circular array and must have its own pair of pointers, FRONT and REA R.
- If each queue is allocated the same amount of space, a two-dimensional array QUEUE can be used instead of the linear arrays.
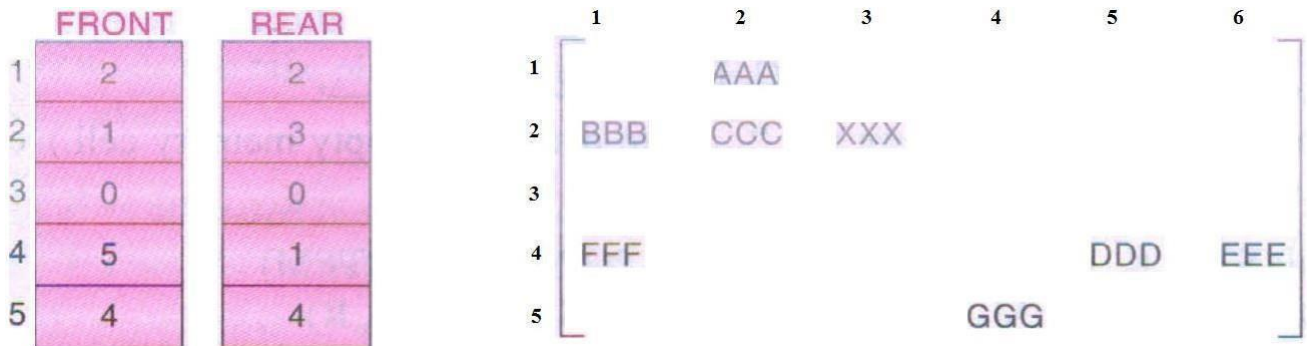
Observe that FRONT[K] and REAR[K] contain, respectively, the front and rear elements of row K of QUEUE, the row that maintains the queue of elements with priority number K.

The following are outlines or algorithms for deleting and inserting elements in a priority queue

Algorithm: This algorithm deletes and processes the first element in a priority queue maintained by a two-dimensional array QUEUE.

1. [Find the first non-empty queue.]
   Find the smallest K such that FRONT[K] ≠ NULL.
2. Delete and process the front element in row K of QUEUE.
3. Exit.

Algorithm: This algorithm adds an ITEM with priority number M to a priority queue maintained by a two-dimensional array QUEUE.

1. Insert ITEM as the rear element in row M ofQUEUE.
2. Exit.

MULTIPLE STACKS AND QUEUES

- In multiple stacks, we examine only **sequential mappings** of stacks into an array. The array is one dimensional which is **memory[MEMORY_SIZE]**. Assume **n** stacks are needed, and then divide the available memory into *n* segments. The array is divided in proportion if the expected sizes of the various stacks are known. Otherwise, divide the memory into equal segments.

- Assume that *i* refers to the stack number of one of the *n* stacks. To establish this stack, create indices for both the **bottom** and **top** positions of this stack. *boundary*[i] points to the position immediately to the left of the bottom element of stack *i*, *top[i]* points to the top element. Stack *i* is empty iff *boundary[i]=top[i]*.

The declarations are:

```
#define MEMORY_SIZE  100              /* size of memory */
#define MAX_STACKS  10               /* max number of stacks plus 1 */
element  memory[MEMORY_SIZE];        /* global  memory  declaration */
int top [MAX_STACKS];
int boundary [MAX_STACKS] ;
int n;                               /*number of stacks entered by the user */
```

To divide the array into roughly equal segments

```
top[0] = boundary[0] = -1;
      for (j= 1;j<n; j++)
              top[j] = boundary[j] = (MEMORY_SIZE / n) * j;
boundary[n] = MEMORY_SIZE  - 1;
```
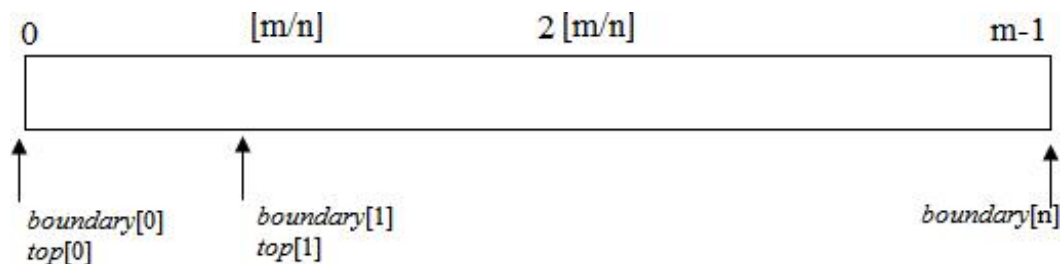

All stacks are empty and divided into roughly equal segments

Figure: Initial configuration for *n* stacks in *memory [m]*.

In the figure, **n** is the number of stacks entered by the user, n < MAX_STACKS, and m = MEMORY_SIZE. Stack **i** grow from **boundary[i] + 1** to **boundary [i + 1]** before it is full. A boundary for the last stack is needed, so set **boundary [n]** to **MEMORY_SIZE-1**.

Implementation of the add operation

```
void push(int i, element item)
        {               /* add an item to the ith stack */
            if (top[i] ==boundary[i+1])
                    stackFull(i);
            memory[++top[i]] = item;
        }
```
Program: Add an item to the ith stack

Implementation of the delete operation

```
element pop(int i)
        {               /* remove top element from the ith stack */
            if (top[i] == boundary[i])
                    return stackEmpty(i);
            return memory[top[i]--];
        }
```
Program: Delete an item from the ith stack

The top[i] == boundary[i+1] condition in push implies only that a particular stack ran out of memory, not that the entire memory is full. But still there may be a lot of unused space between other stacks in array memory as shown in Figure.

Therefore, create an error recovery function called **stackFull** , which determines if there is any free space in memory. If there is space available, it should shift the stacks so that space is allocated to the full stack.
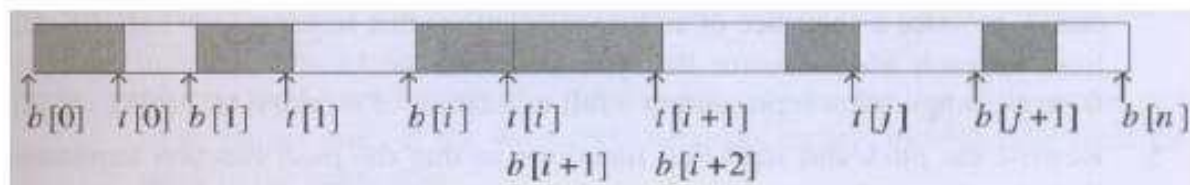
b boundary , t=top

Method to design stackFull

- Determine the least, *j,* i *< j < n,* such that there is free space between stacks *j* and j+1. That is, *top[j] < boundary[j+l].* If there is a j, then move stacks *i+l,i+2, ..,* j one position to the right (treating *memory[O]* as leftmost and *memory[MEMORY_SIZE -* 1] as rightmost). This creates a space between stacks *i* and *i+1.*

- If there is no j as in (1), then look to the left of stack *i.* Find the largest j such that $0 \le j \le i$ and there is space between stacks j and j+ 1 ie, *top[j] < boundary[j+l].* If there is a j, then move stacks *j+l, j+2, ... , i* one space to the left. This also creates space between stacks *i* and *i+1.*

- If there is no j satisfying either condition (1) or condition (2), then all *MEMORY_SIZE* spaces of memory are utilized and there is no free space. In this case *stackFull* terminates with an error message.

MODULE 3: LINKED LIST

DEFINITION

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

- The first part contains the information of the element, and
- The second part, called the *link field* or *nextpointer* field, contains the address of the next node in the list.
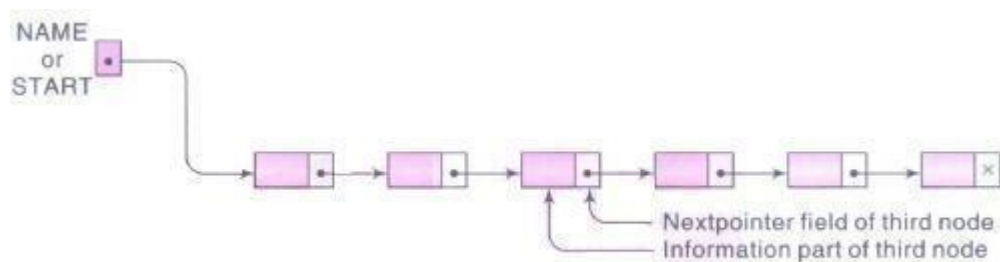
Fig: Linked list with 6 nodes

In the above figure each node is pictured with two parts.

- The left part represents the information part of the node, which may contain an entire record of data items.
- The right part represents the nextpointer field of the node
- An arrow drawn from a node to the next node in the list.
- The pointer of the last node contains a special value, called the *null pointer*, which is any invalid address.

A pointer variable called **START or FIRST** which contains the address of the first node.
A special case is the list that has no nodes, such a list is called the *null list or empty list* and is denoted by the null pointer in the variable START.

REPRESENTATION OF LINKED LISTS IN MEMORY

Let LIST be a linked list. Then LIST will be maintained in memory as follows.

1. LIST requires two linear arrays such as INFO and LINK-such that INFO[K] and LINK[K] contains the information part and the nextpointer field of a node of LIST.
2. LIST also requires a variable name such as START which contains the location of the beginning of the list, and a nextpointer sentinel denoted by NULL-which indicates the end of the list.
3. The subscripts of the arrays INFO and LINK will be positive, so choose NULL = 0, unless otherwise stated.

The following examples of linked lists indicate that the nodes of a list need not occupy adjacent

elements in the arrays INFO and LINK, and that more than one list may be maintained in the same linear arrays INFO and LINK. However, each list must have its own pointer variable giving the location of its first node.
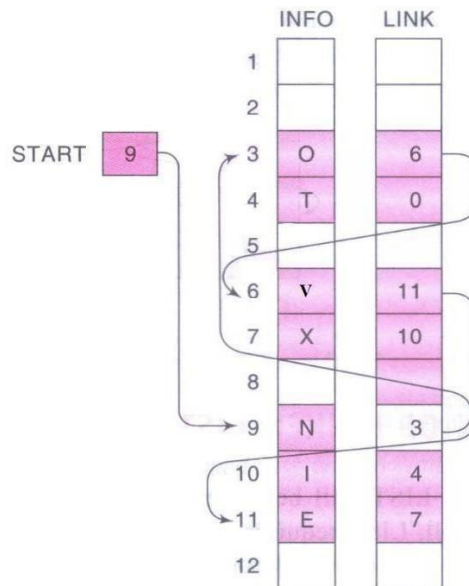
START=9   INFO[9]=N
LINK[3]=6   INFO[6]=V
LINK[6]=11   INFO[11]=E
LINK[11]=7   INFO[7]= X
LINK[7]=10   INFO[10]= I
LINK[10]=4   INFO[4]= T
LINK[4]= NULL value, So the list has ended

REPRESENTING CHAIN IN C

The following capabilities are needed to make linked representation

1. A mechanism for defining a node's structure, that is, the field it contains. So self-referential structures can be used
2. A way to create new nodes, so MALLOC functions can do this operation
3. A way to remove nodes that no longer needed. The FREE function handles this operation.

Defining a node structure

typedef struct listNode *listPointer typedef struct {
                      char data[4]; listPointer list;
            } listNode;

Create a New Empty list

listPointer first = NULL

To create a New Node

MALLOC (first, sizeof(*first));

To place the data into NODE

strcpy(first→ data,"BAT"); first→ link =
NULL
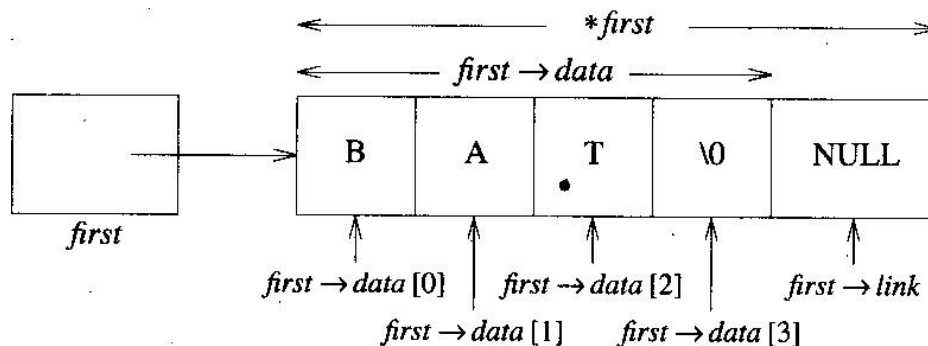
MEMORY ALLOCATION - GARBAGE COLLECTION

- The maintenance of linked lists in memory assumes the possibility of inserting new nodes into the lists and hence requires some mechanism which provides unused memory space for the new nodes.
- Mechanism is required whereby the memory space of deleted nodes becomes available for future use.
- Together with the linked lists in memory, a special list is maintained which consists of unused memory cells. This list, which has its own pointer, is called ***the list of available space or the free storage list or the free pool***.

Suppose linked lists are implemented by parallel arrays and insertions and deletions are to be performed linked lists. Then the unused memory cells in the arrays will also be linked together to form a linked list using AVAIL as its list pointer variable. Such a data structure will be denoted by LIST (INFO, LINK, START, AVAIL)
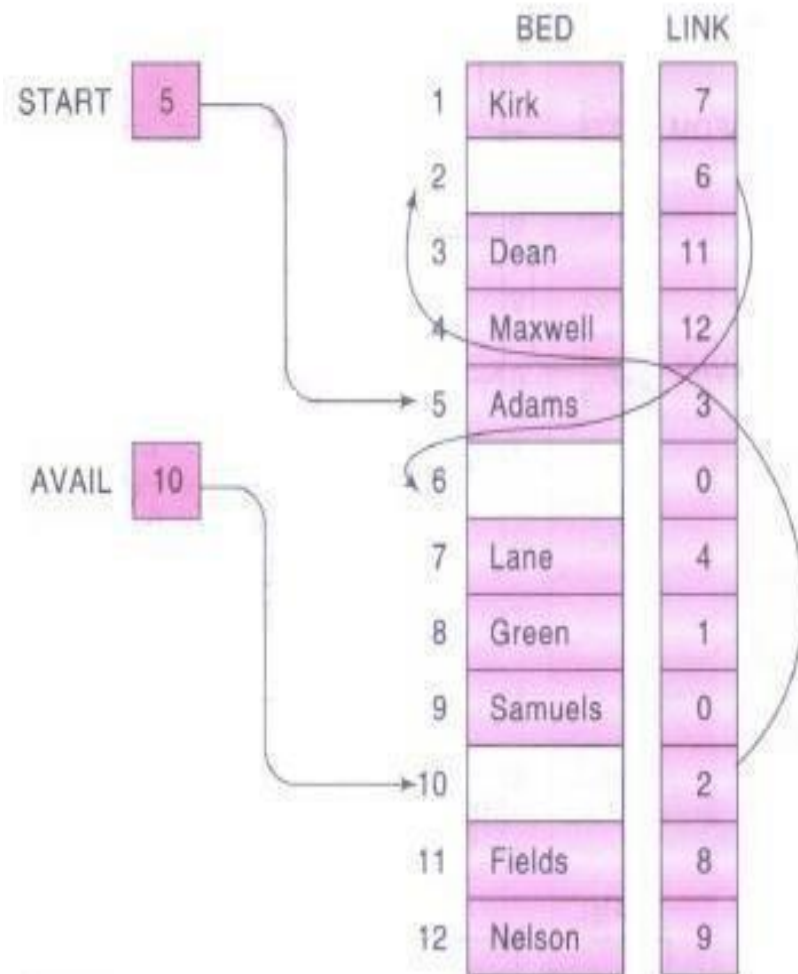
Garbage Collection

- Suppose some memory space becomes reusable because a node is deleted from a list or an entire list is deleted from a program. So space is need to be available for future use.
- One way to bring this is to immediately reinsert the space into the free-storage list. However, this method may be too time-consuming for the operating system of a computer, which may choose an alternative method, as follows.

The operating system of a computer may periodically collect all the deleted space onto the freestorage list. Any technique which does this collection is called garbage collection.
Garbage collection takes place in two steps.
1. First the computer runs through all lists, tagging those cells which are currently in use
2. And then the computer runs through the memory, collecting all untagged space onto the free-storage list.

The garbage collection may take place when there is only some minimum amount of space or no space at all left in the free-storage list, or when the CPU is idle and has time to do the collection.
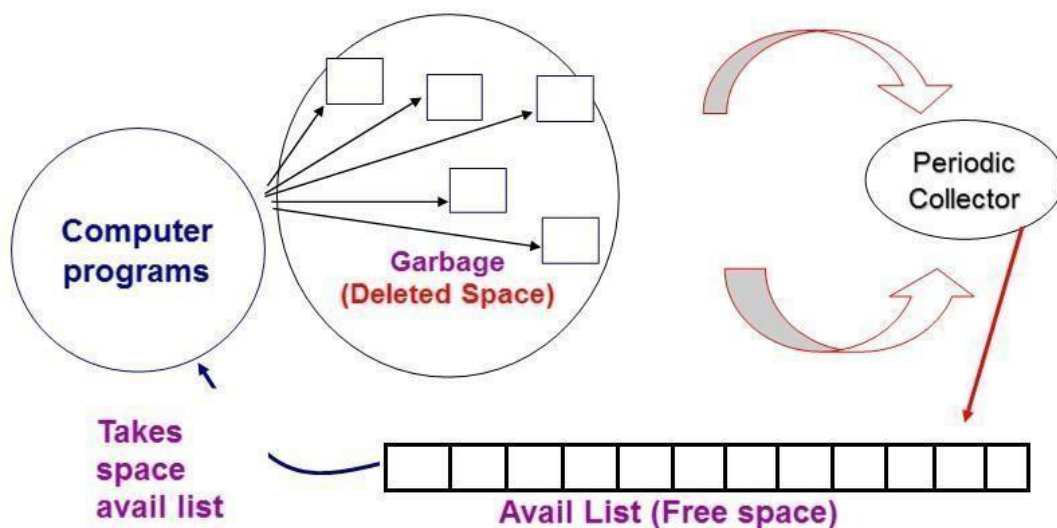
Overflow

- Sometimes new data are to be inserted into a data structure but there is no available space, i.e., the free-storage list is empty. This situation is usually called *overflow*.
- The programmer mayhandle overflow byprinting the message OVERFLOW. In such a case, the programmer may then modify the program by adding space to the underlying arrays.
- Overflow will occur with linked lists when AVAIL = NULL and there is an insertion.

Underflow

- The term underflow refers to the situation where one wants to delete data from a data structure that is empty.
- The programmer mayhandle underflow byprinting the message UNDERFLOW.
- The underflow will occur with linked lists when START= NULL and there is a deletion.

LINKED LIST OPERATIONS

1. Traversing a Linkedlist

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

- Traversing algorithm uses a pointer variable PTR which points to the node that is currently being processed.
- PTR→LINK points to the next node to be processed.
- Thus the assignment PTR= PTR→LINK moves the pointer to the next node in the list, as pictured in below figure

PTR : = LINK[PTR]
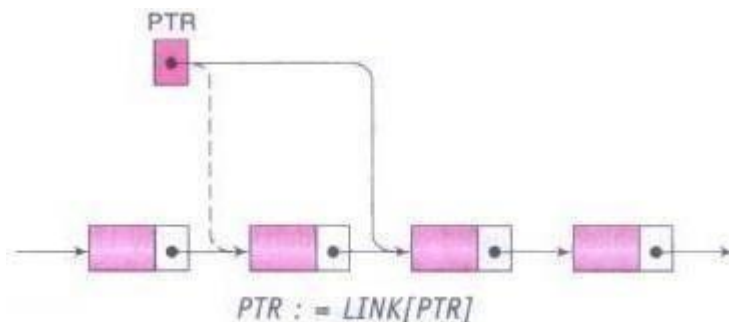
Algorithm: (Traversing a Linked List) Let LIST be a linked list in memory. This algorithm traverses LIST, applying an operation PROCESS to each element of LIST.

The variable PTR points to the node currently being processed.

    1. Set PTR = START
    2. Repeat Steps 3 and 4 while PTR ≠ NULL
    3.   ApplyPROCESS to PTR→INFO
    4.   Set PTR = PTR→LINK
    5. Exit.

The details of the algorithm are as follows.

- Initialize PTR or START.
- Then process PTR→INFO, the information at the first node.
- Update PTR by the assignment PTR = PTR→LINK, so that PTR points to the second node. Then process PTR→INFO, the information at the second node. Again update PTR by the assignment PTR = PTR→LINK, and then process PTR→INFO, the information at the third node. And so on. Continue until PTR = NULL, which signals the end of the list.

Example:
The following procedure prints the information at each node of a linked list. Since the procedure must traverse the list.

Procedure: PRINT (INFO, LINK, START)
1. Set PTR = START.
2. Repeat Steps 3 and 4 while PTR ≠ NULL:
3.   Write: PTR→INFO
4.   Set PTR = PTR→LINK
5. Return.

2. Searching a Linkedlist
There are two searching algorithm for finding location LOC of the node where ITEM first appears in LIST.

Let LIST be a linked list in memory. Suppose a specific ITEM of information is given.
If ITEM is actually a key value and searching through a file for the record containing ITEM, then ITEM can appear only once in LIST.

LIST Is Unsorted
Suppose the data in LIST are not sorted. Then search for ITEM in LIST by traversing through the list using a pointer variable PTR and comparing ITEM with the contents PTR→INFO of each node, one by one, of LIST. Before updating the pointer PTR by
PTR = PTR→LINK
It requires two tests.
First check whether we have reached the end of the list, i.e.,
PTR == NULL
If not, then check to see whether
PTR→INFO == ITEM

Algorithm: SEARCH (INFO, LINK, START, ITEM, LOC)

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

    1. Set PTR: = START.
    2. Repeat Step 3 while PTR ≠ NULL
    3.   If ITEM = PTR→INFO, then:
           Set LOC: = PTR, and Exit.
       Else
           Set PTR: = PTR→LINK
       [End of If structure.]
       [End of Step 2 loop.]
    4. [Search is unsuccessful.] Set LOC: = NULL.
    5. Exit.

The complexity of this algorithm for the worst-case running time is proportional to the number n of elements in LIST, and the average-case running time is approximately proportional to $n/2$ (with the condition that ITEM appears once in LIST but with equal probability in any node of LIST).

LIST is Sorted

Suppose the data in LIST are sorted. Search for ITEM in LIST by traversing the list using a pointer variable PTR and comparing ITEM with the contents PTR→INFO of each node, one byone, of LIST. Now, searching can stop once ITEM exceeds PTR→INFO.

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

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

    1. Set PTR: = START.
    2. Repeat Step 3 while PTR ≠ NULL
    3.   If ITEM < PTR→INFO, then:
           Set PTR: = PTR→LINK
       Else if ITEM = PTR→INFO, then:
           Set LOC: = PTR, and Exit. [Search is successful.]
       Else:
           Set LOC: = NULL, and Exit. [ITEM now exceeds PTR→INFO]
       [End of If structure.]
       [End of Step 2 loop.]
    4. Set LOC: = NULL.
    5. Exit.

The complexity of this algorithm for the worst-case running time is proportional to the number n of elements in LIST, and the average-case running time is approximately proportional to $n/2$

3. Insertion into a Linked list

Let LIST be a linked list with successive nodes A and B, as pictured in Fig. (a). Suppose a node N is to be inserted into the list between nodes A and B. The schematic diagram of such an insertion appears in Fig. (b). That is, node A now points to the new node N, and node N points to node B, to which A previously pointed.

(a). Before Insertion

(b). After Insertion
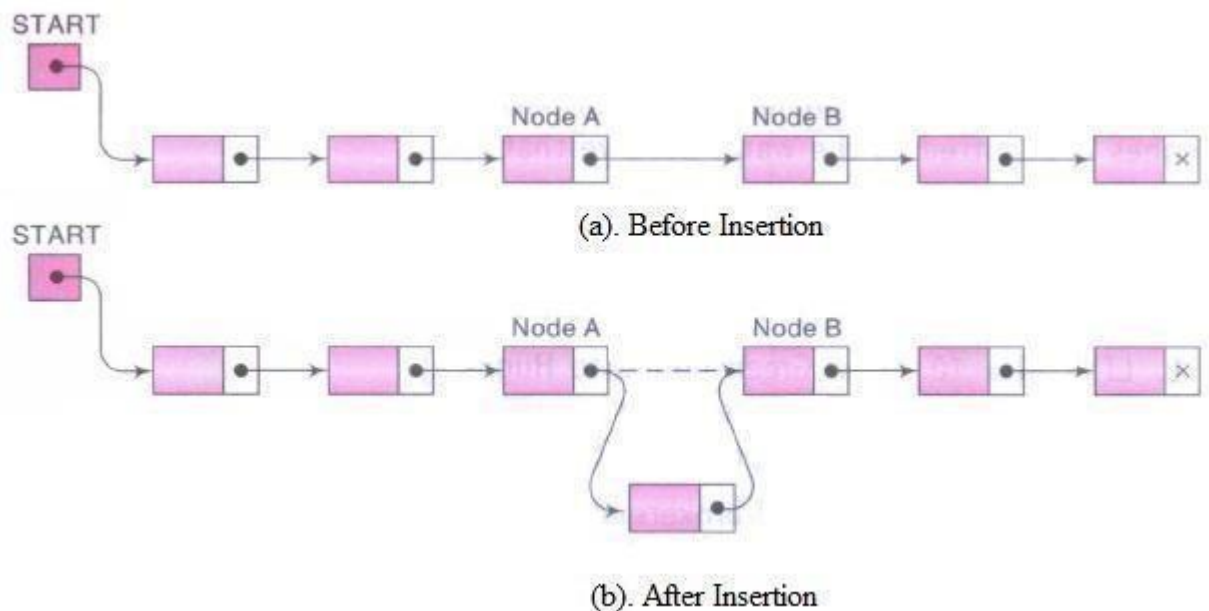
The above figure does not take into account that the memory space for the new node N will come from the AVAIL list.

Specifically, for easier processing, the first node in the AVAIL list will be used for the new node N. Thus a more exact schematic diagram of such an insertion is that in below Fig.
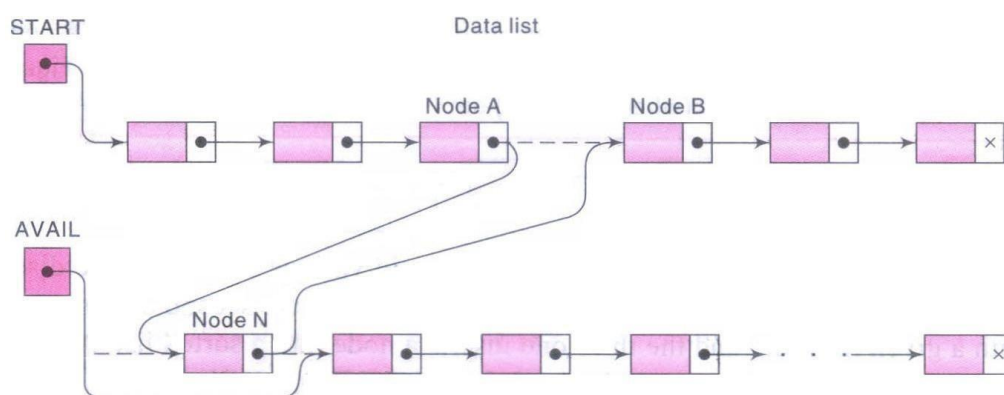
Observe that three pointer fields are changed as follows:

1. The nextpointer field of node A now points to the new node N, to which AVAIL previously pointed.
2. AVAIL now points to the second node in the free pool, to which node N previously pointed.
3. The nextpointer field of node N now points to node B, to which node A previously pointed.

There are also two special cases.

1. If the new node N is the first node in the list, then START will point to N
2. If the new node N is the last node in the list, then N will contain the null pointer.

Insertion Algorithms
Algorithms which insert nodes into linked lists come up in various situations.

1. Inserts a node at the beginning of the list,
2. Inserts a node after the node with a givenlocation
3. Inserts a node into a sorted list.

1. Inserting at the Beginning of a List
Inserting the node at the beginning of the list.

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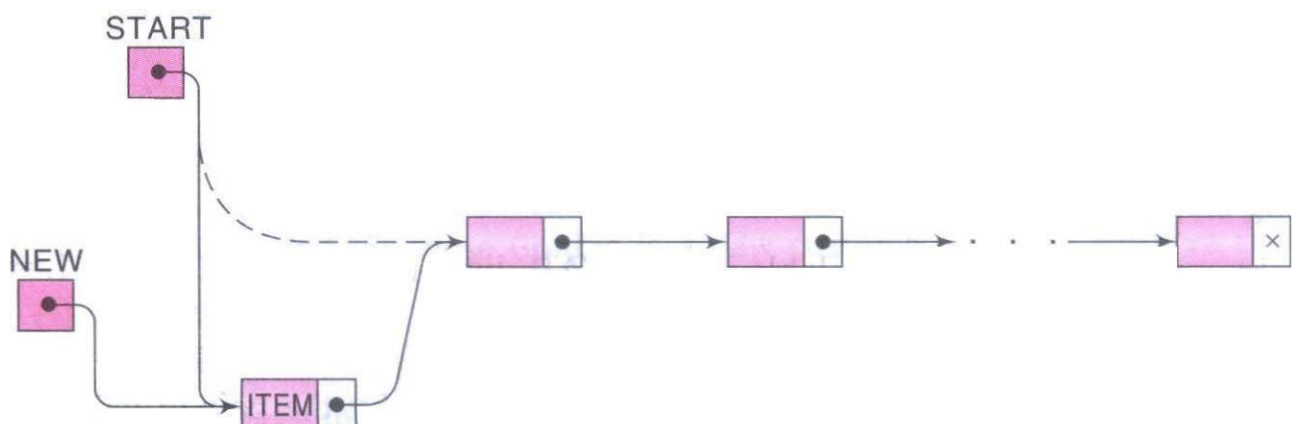
Fig: Inserting at the Beginning of a List

2. **Inserting after a Given Node**

Suppose the value of LOC is given where either LOC is the location of a node A in a linked LIST or LOC = NULL.

The following is an algorithm which inserts ITEM into LIST so that ITEM follows node A or, when LOC = NULL, so that ITEM is the first node.

Let N denote the new node. If LOC = NULL, then N is inserted as the first node in LIST. Otherwise, let node N point to node B by the assignment NEW→LINK:= LOC→LINK and let node A point to the new node N by the assignment LOC→LINK:= NEW

Algorithm: INSLOC (INFO, LINK, START, AVAIL, LOC, ITEM)

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

1. [OVERFLOW?] If AVAIL = NULL, then: Write: OVERFLOW, and Exit
2. [Remove first node from AVAIL list.]
   Set NEW: = AVAIL and AVAIL: = AVAIL→LINK
3. Set NEW→INFO:= ITEM [Copies new data into new node]
4. If LOC = NULL, then: [Insert as first node]
   Set NEW→LINK:= START and START:= NEW.
   Else: [Insert after node with location LOC]
   Set NEW→LINK:= LOC→LINK and LOC→LINK:= NEW
   [End of If structure.]
5. Exit.

---

3. **Inserting into a Sorted Linked List**

- Suppose ITEM is to be inserted into a sorted linked LIST. Then ITEM must be inserted between nodes A and B sothat

INFO(A) < ITEM < INFO(B)

- The following is a procedure which finds the location LOC of node A, that is, which finds the location LOC of the last node in LIST whose value is less than ITEM.
- Traverse the list, using a pointer variable PTR and comparing ITEM with PTR→INFO at each node. While traversing, keep track of the location of the preceding node by using a pointer variable SAVE, as pictured in below Fig. Thus SAVE and PTR are updated by the assignments

SAVE: = PTR and PTR: = PTR→LINK

- The traversing continues as long as PTR→INFO > ITEM, or in other words, the traversing stops as-soon as ITEM ≤ PTR→INFO. Then PTR points to node B, so SAVE will contain the location of the node A.

Procedure: FINDA (INFO, LINK, START, ITEM, LOC)

This procedure finds the location LOC of the last node in a sorted list such that LOC→INFO < ITEM, or sets LOC = NULL.

1. [List empty?] If START = NULL, then: Set LOC: = NULL, and Return.
2. [Special case?] If ITEM < START→INFO, then: Set LOC: = NULL, and Return.
3. Set SAVE: = START and PTR: = START→LINK. [Initializes pointers.]
4. Repeat Steps 5 and 6 while PTR ≠ NULL.
5.   If ITEM < PTR→INFO, then:
          Set LOC: = SAVE, and Return.
      [End of If structure.]
6.   Set SAVE: = PTR and PTR: = PTR→LINK. [Updates pointers.]
   [End of Step 4 loop.]
7. Set LOC: = SAVE.
8. Return.

Below algorithm which inserts ITEM into a linked list. The simplicity of the algorithm comes from using the previous two procedures.

Algorithm: INSERT (INFO, LINK, START, AVAIL, ITEM)

This algorithm inserts ITEM into a sorted linked list.
1. [Use Procedure to find the location of the node preceding ITEM.]
        Call FINDA (INFO, LINK, START, ITEM, LOC).
2. [Use Algorithm to insert ITEM after the node with location LOC.]
        Call INSLOC (INFO, LINK, START, AVAIL, LOC, ITEM).
3. Exit.

3. <u>Deletion into a Linkedlist</u>

- Let LIST be a linked list with a node N between nodes A and B, as pictured in below Fig.(a). Suppose node N is to be deleted from the linked list. The schematic diagram of such a deletion appears in Fig.(b).
- The deletion occurs as soon as the nextpointer field of node A is changed so that it points to node B.
- Linked list is maintained in memory in the form

LIST (INFO, LINK, START, AVAIL)

(a) Before deletion

(b) After deletion
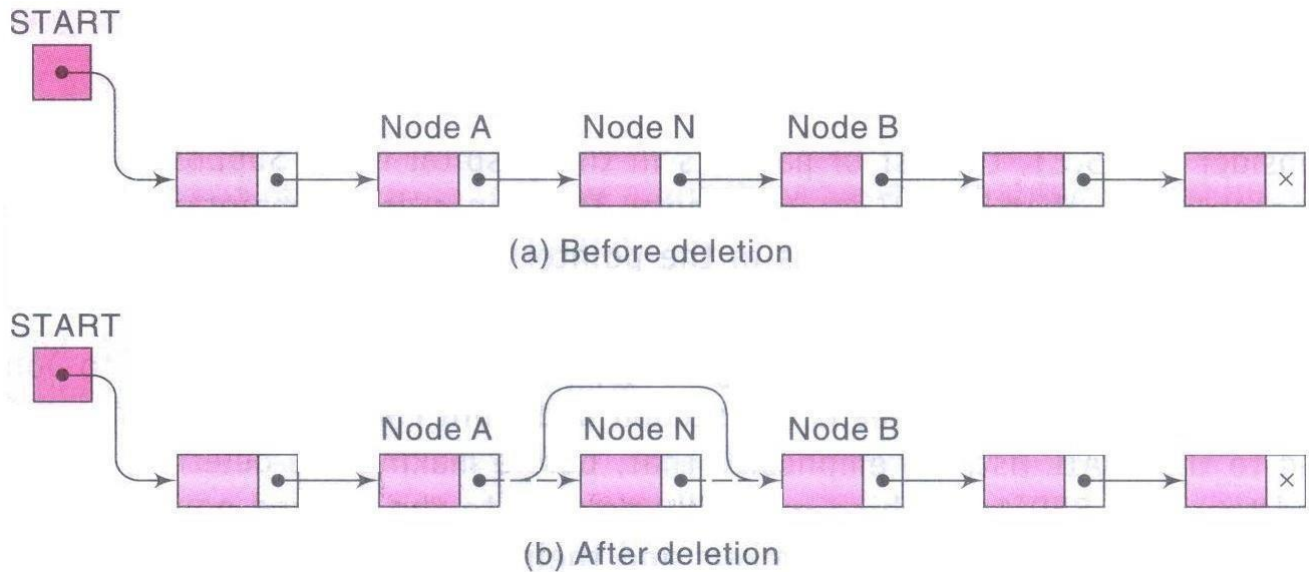
The above figure does not take into account the fact that, when a node N is deleted from our list, immediately return its memory space to the AVAIL list. So for easier processing, it will be returned to the beginning of the AVAIL list. Thus a more exact schematic diagram of such a deletion is the one in below Fig.
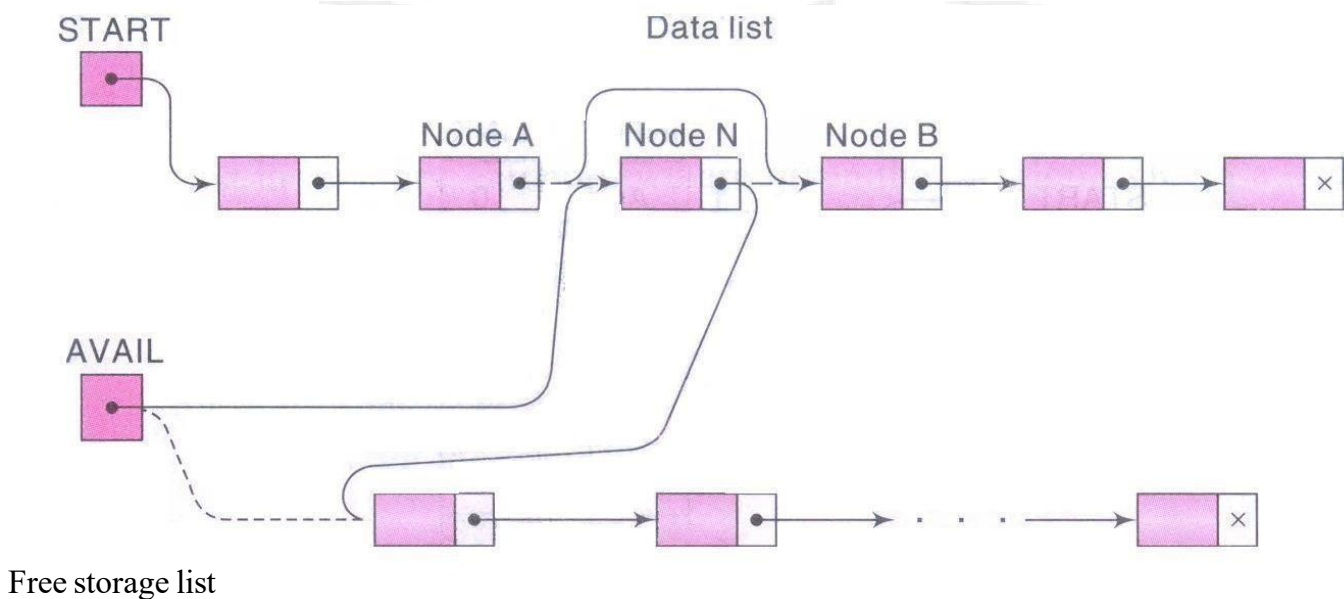
Free storage list

Observe that three pointer fields are changed as follows:

1. The nextpointer field of node A now points to node B, where node N previously pointed.
2. The nextpointer field of N now points to the original first node in the free pool, where AVAIL previously pointed.
3. AVAIL now points to the deleted node N.

Deletion Algorithms

Deletion of nodes from linked lists come up in various situations.

1. Deletes the node following a given node
2. Deletes the node with a given ITEM of information.

All deletion algorithms will return the memory space of the deleted node N to the beginning of the AVAIL list.

Deleting the Node Following a Given Node

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

The following algorithm deletes N from the list.

Algorithm: DEL (INFO, LINK, START, AVAIL, LOC, LOCP)

This algorithm deletes the node N with location LOC. LOCP is the location of the node which precedes N or, when N is the first node, LOCP = NULL.

1. If LOCP = NULL, then:
   Set START: = START→LINK.   [Deletes first node.]
   Else:
   Set LOCP→LINK:= LOC→LINK   [Deletes node N.]
   [End of If structure.]
2. [Return deleted node to the AVAIL list.]
Set LOC→LINK:= AVAIL and AVAIL: = LOC
3. Exit.

Deleting the Node with a Given ITEM of Information

- Consider a given an ITEM of information and wants to delete from the LIST the first node N which contains ITEM. Then it is needed to know the location of the node preceding N. Accordingly, first finds the location LOC of the node N containing ITEM and the location LOCP of the node preceding node N.

- If N is the first node, then set LOCP = NULL, and if ITEM does not appear in LIST, then set LOC = NULL.

- Traverse the list, using a pointer variable PTR and comparing ITEM with PTR→INFO at each node. While traversing, keep track of the location of the preceding node by using a pointer variable SAVE. Thus SAVE and PTR are updated by the assignments SAVE:=PTR and PTR:= PTR→LINK

- The traversing continues as long as PTR→INFO ≠ ITEM, or in other words, the traversing stops as soon as ITEM = PTR→INFO. Then PTR contains the location LOC of node N and SAVE contains the location LOCP of the node preceding N

Procedure: FINDB (INFO, LINK, START, ITEM, LOC, LOCP)

This procedure finds the location LOC of the first node N which contains ITEM and the location LOCP of the node preceding N. If ITEM does not appear in the list, then the procedure sets LOC = NULL; and if ITEM appears in the first node, then it sets LOCP = NULL.

1. [List empty?] If START = NULL, then:
   Set LOC: = NULL and LOCP: = NULL, and Return.
   [End of If structure.]
2. [ITEM in first node?] If START→INFO = ITEM, then:
   Set LOC: = START and LOCP = NULL, and Return.
   [End of If structure.]
3. Set SAVE: = START and PTR: = START→LINK. [Initializespointers.]
4. Repeat Steps 5 and 6 while PTR ≠ NULL.
5. If PTR→INFO = ITEM, then:
   Set LOC: = PTR and LOCP: = SAVE, and Return.
   [End of If structure.]
6. Set SAVE: = PTR and PTR: = PTR→LINK. [Updates pointers.] [End of Step 4 loop.]
7. Set LOC: = NULL. [Search unsuccessful.]
8. Return.

DOUBLY LINKED LIST

1. The difficulties with single linked lists is that, it is possible to traversal only in one direction, ie., direction of the links.
2. The only way to find the node that precedes p is to start at the beginning of the list. The same problem arises when one wishes to delete an arbitrary node from a singly linked list. Hence the solution is to use doubly linkedlist

Doubly linked list: It is a linear collection of data elements, called nodes, where each node N is divided into three parts:

1. An information field INFO which contains the data of N
2. A pointer field LLINK (FORW) which contains the location of the next node in the list
3. A pointer field RLINK (BACK) which contains the location of the preceding node in the list

The declarations are:

```
typedef   struct   node   *nodePointer;
typedef struct {
                nodePointer llink;
                element       data;
                nodePointer rlink;
        } node;
```
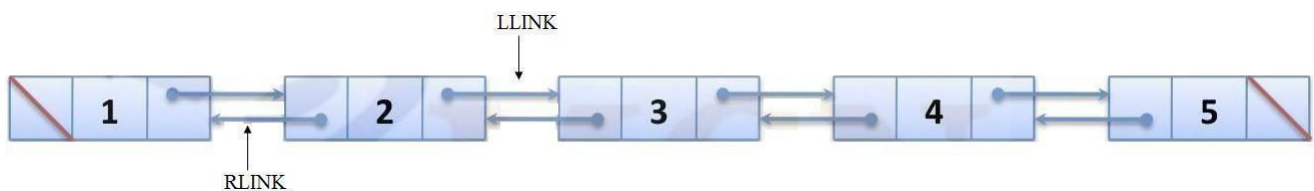

Insertion into a doubly linked list

Insertion into a doubly linked list is fairly easy. Assume there are two nodes, node and newnode, node may be either a header node or an interior node in a list. The function dinsert performs the insertion operation in constant time.

```
void dinsert(nodePointer node, nodePointer newnode)
        {/* insert newnode to the right of node */
                newnode→llink      =      node;
                newnode→rlink   =   node→rlink;
                node→rlink→llink   =   newnode;
                node→rlink = newnode;
        }
```
Program: Insertion into a doubly linked circular list

<u>Deletion from a doubly linked list</u>

Deletion from a doubly linked list is equally easy. The function *ddelete* deletes the node deleted from the list pointed to by node.

To accomplish this deletion, we only need to change the link fields of the nodes that precede (deleted→llink→rlink) and follow (deleted→rlink→llink) the node we want to delete.

```
void ddelete(nodePointer node, nodePointer  deleted)
        {/* delete from the doubly linked list */
                if (node == deleted)
                        printf("Deletion of header node not permitted.\n");
else {

                        deleted→llink→rlink  =  deleted→rlink;
                        deleted→rlink→llink  =  deleted→llink;
                        free(deleted) ;
                }
}
```

Program: Deletion from a doubly linked circular list

HEADER LINKED LISTS

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

The following are two kinds of widely used header lists:

1. A grounded header list is a header list where the last node contains the null pointer.
2. A circular header list is a header list where the last node points back to the headernode.

Below figure contains schematic diagrams of these header lists.

(a) Grounded header list

(b) Circular header list
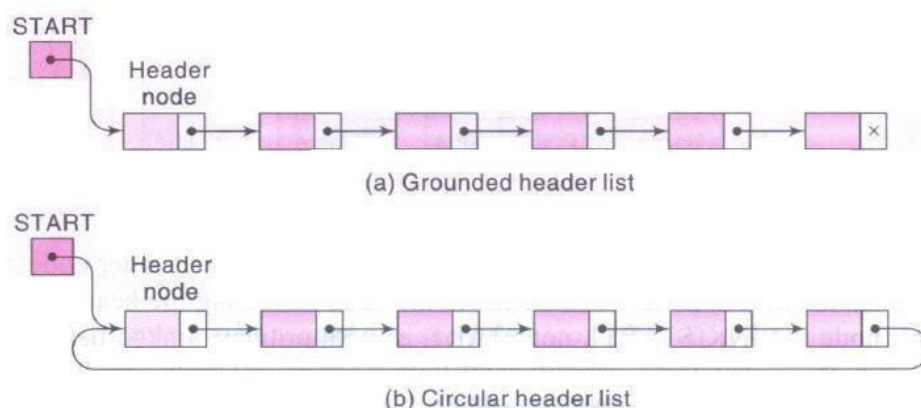
Observe that the list pointer START always points to the header node.

- If START→LINK = NULL indicates that a grounded header list is empty
- If START→LINK = START indicates that a circular header list is empty.

The first node in a header list is the node following the header node, and the location of the first node is START→LINK, not START, as with ordinary linked lists.

Below algorithm, which uses a pointer variable PTR to traverse a circular header list

1. Begins with PTR = START→LINK (not PTR = START)
2. Ends when PTR = START (not PTR = NULL).

Algorithm: (Traversing a Circular Header List) Let LIST be a circular header list in memory. This algorithm traverses LIST, applying an operation PROCESS to each node of LIST.

1. Set PTR: = START→LINK. [Initializes the pointer PTR.]
2. Repeat Steps 3 and 4 while PTR ≠ START:
3.   Apply PROCESS to PTR→INFO.
4.   Set PTR: = PTR→LINK. [PTR now points to the next node.]
   [End of Step 2 loop.]
5. Exit.

Algorithm: SRCHHL (INFO, LINK, START, ITEM, LOC)

LIST is a circular header list in memory. This algorithm finds the location LOC of the node where ITEM first appears in LIST or sets LOC = NULL.

1. Set PTR: = START→LINK
2. Repeat while PTR→INFO [PTR] ≠ ITEM and PTR ≠ START:
    Set PTR: =PTR→LINK.   [PTR now points to the next node.]
   [End of loop.]
3. If PTR→INFO = ITEM, then:
    Set LOC: = PTR.
   Else:
    Set LOC: = NULL.
   [End of If structure.]
4. Exit.

The two properties of circular header lists:

1. The null pointer is not used, and hence all pointers contain valid addresses.
2. Every (ordinary) node has a predecessor, so the first node may not require a special case.

There are two other variations of linked lists

1. A linked list whose last node points back to the first node instead of containing the null pointer, called a circular list
2. A linked list which contains both a special header node at the beginning of the list and a special trailer node at the end of the list

(a) Circular linked list

(b) Linked list with header and trailer nodes
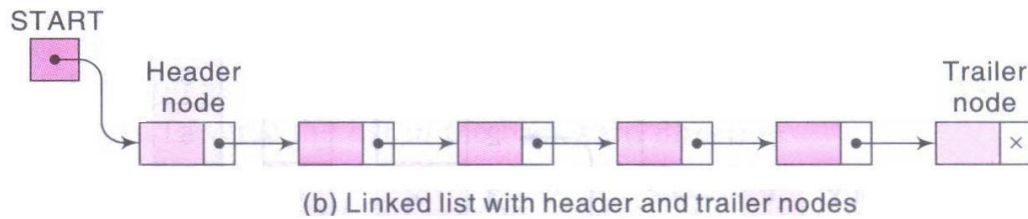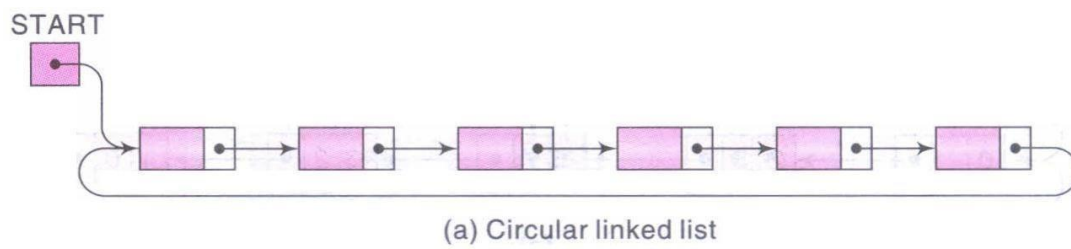
LINKED STACKS AND QUEUES

The below figure shows stacks and queues using linked list. Nodes can easily add or delete a node from the top of the stack. Nodes can easily add a node to the rear of the queue and add or delete a node at the front

(a) Linked stack

(b) Linked queue
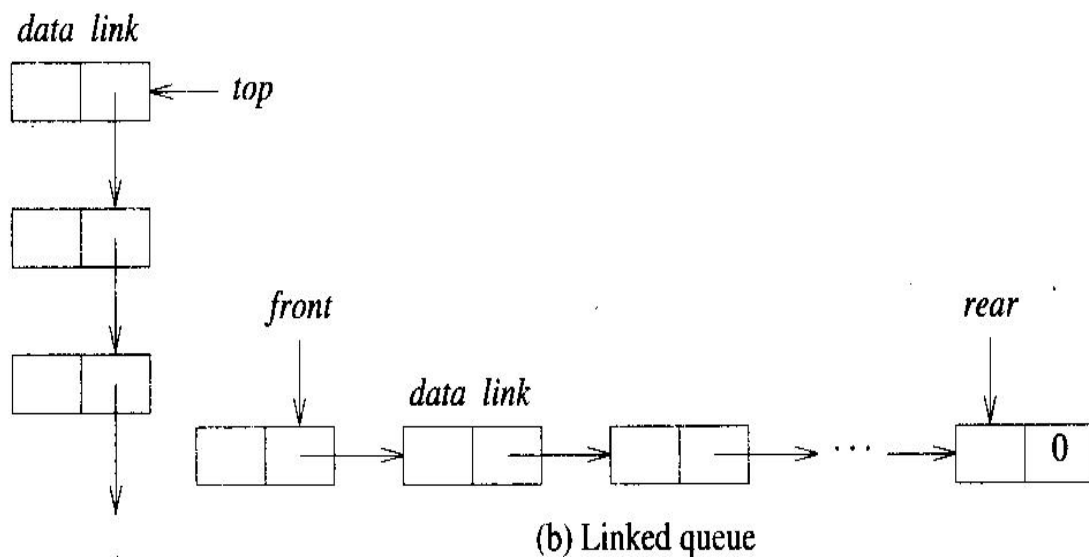
Linked Stack

```
#define MAX_STACKS 10        /* maximum number of stacks */
typedef struct {
            int key;
            /* other fields */
        }element;
typedef struct stack *stackPointer; typedef
struct {
            element data;
            stackPointer link;
        } stack;
stackPointer top[MAX_STACKS];
```

The representation of $n \leq MAX\_STACKS$ stacks, below is the declarations:

The initial condition for the stacks is:

$top[i] = NULL, 0 \leq i < MAX\_STACKS$

The boundary condition is:

$top [i] = NULL$ iff the ith stack is empty

Functions push and pop add and delete items to/from a stack.

```
void push(int i, element item)
    {                                   /* add item to the ith stack */
        stackPointer           temp;
        MALLOC(temp,    sizeof(*temp));
        temp→data = item;
        temp→link   =   top[i];
        top[i] = temp;
    }
```

Program: Add to a linked stack

Function push creates a new node, temp, and places item in the data field and top in the link field. The variable top is then changed to point to temp. A typical function call to add an element to the ith stack would be push (i,item).

```
element pop(int i)
    {                                   /* remove top element from the ith stack */
        stackPointer temp = top[i];element item;
```

```
                    if (! temp)
                            return stackEmpty();
                    item = temp→data;
                    top[i]  =  temp→link;
                    free (temp) ;
                    return item;
            }
```

Program: Delete from a linked stack

Function pop returns the top element and changes top to point to the address contained in its link field. The removed node is then returned to system memory. A typical function call to delete an element from the ith stack would be item = pop (i);

Linked Queue

The representation of m ≤ MAX_QUEUES queues, below is the declarations:

```
            #define MAX-QUEUES 10 /* maximum number of queues */
            typedef.struct queue *queuePointer;
            typedef struct {
                            element data;
                            queuePointer link;
             } queue;
            queuePointer front[MAX_QUEUES], rear[MAX_QUEUES];
```

The initial condition for the queues is:
front[i] = NULL, $0 \leq i < MAX\_QUEUES$

The boundary condition is:
front[i] = NULL iff the ith queue is empty

Functions addq and deleteq implement the add and delete operations for multiple queues.

```
        void addq(i, item)
                {               /* add item to the rear of queue i */
                queuePointer temp;
                MALLOC(temp,    sizeof(*temp));
                temp→data = item;
                temp→link = NULL;

                if (front[i])
                        rear[i] →link = temp;
```

```
                        else
                                front[i] = temp;
                        rear[i] = temp;
                }
```

Program: Add to the rear of a linked queue

Function addq is more complex than push because we must check for an empty queue. If the queue is empty, then change front to point to the new node; otherwise change rear's link field to point to the new node. In either case, we then change rear to point to the new node.

```
        element deleteq(int i)
                {                               /* delete an element from queue i */
                        queuePointer temp =  front[i];
                        element item;
                        if (! temp)
                                return queueEmpty();
                        item      =      temp→data;
                        front[i]= temp→link;
                        free   (temp)   ;
                        return item;
                }
```

Program: Delete from the front of a linked queue

Function deleteq is similar to pop since nodes are removing that is currently at the start of the list. Typical function calls would be addq (i, item); and item = deleteq (i);

APPLICATIONS OF LINKED LISTS – POLYNOMIALS

Representation of the polynomial:

$$A(x) = a_{m-1}x^{e_{m-1}} + \cdots + a_0x^{e_0}$$

where the a_i are nonzero coefficients and the e_i are nonnegative integer exponents such that $e_{m-1} > e_{m-2} > ... > e_1 > e_0 \geq 0$.

Present each term as a node containing coefficient and exponent fields, as well as a pointer to the next term.

Assuming that the coefficients are integers, the type declarations are:

```
typedef   struct   polyNode   *polyPointer;
typedef struct {
                int    coef;
                int expon;
                polyPointer link;
        } polyNode;
polyPointer a,b;
```

coef	expon	link

Figure shows how we would store the polynomials

$$a = 3x^{14} + 2x^8 + 1$$

and

$$b = 8x^{14} - 3x^{10} + 10x^6$$

(a)

(b)
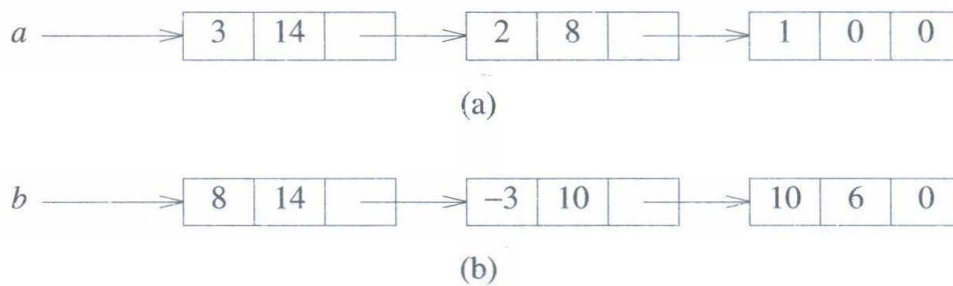
Figure: Representation of $3x^{14} + 2x^8 + 1$ and $8x^{14} - 3x^{10} + 10x^6$

Adding Polynomials

To add two polynomials, examine their terms starting at the nodes pointed to by **a** and **b**.

- If the exponents of the two terms are equal, then add the two coefficients and create a new term for the result, and also move the pointers to the next nodes in **a** and **b**.
- If the exponent of the current term in **a** is less than the exponent of the current term in **b**, then create a duplicate term of **b**, attach this term to the result, called **c**, and advance the pointer to the next term in **b**.
- If the exponent of the current term in **b** is less than the exponent of the current term in **a**, then create a duplicate term of **a**, attach this term to the result, called **c**, and advance the pointer to the next term in **a**

Below figure illustrates this process for the polynomials addition.
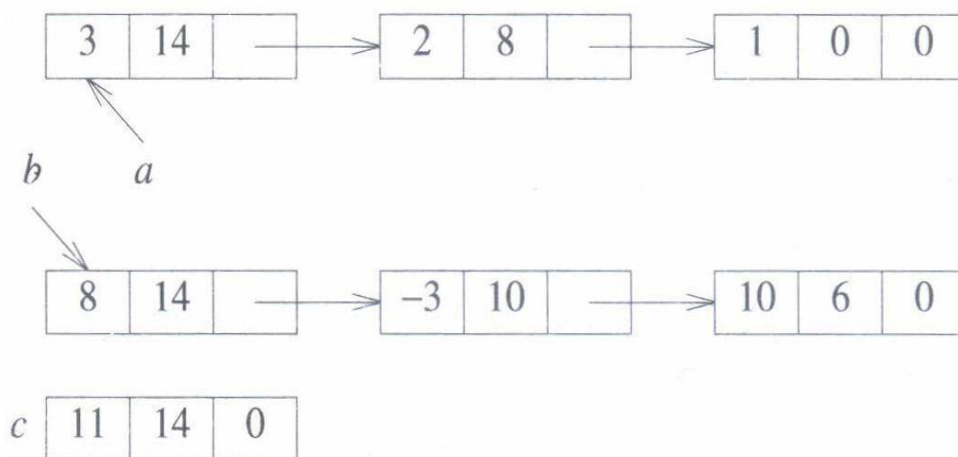
$$(i)\ a \rightarrow expon == b \rightarrow expon$$
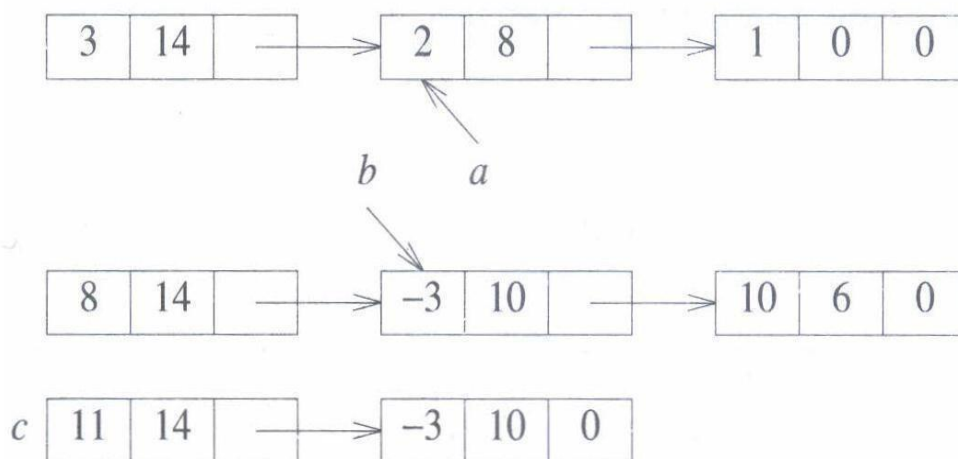
$$(ii)\ a \rightarrow expon < b \rightarrow expon$$

(iii) $a \rightarrow expon > b \rightarrow expon$
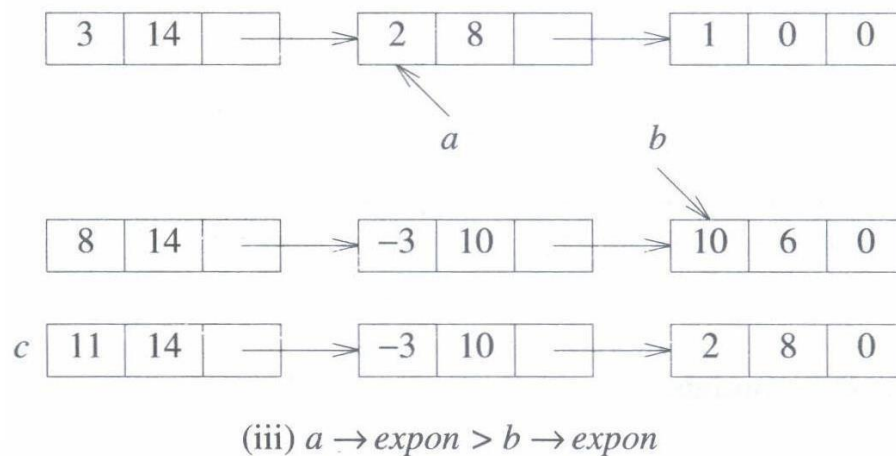
Figure: Generating the first three terms of $c = a + b$

The complete addition algorithm is specified by padd( )

```
polyPointer padd(polyPointer a, polyPointer b)
{/* return a polynomial which is the sum of a and b */
   polyPointer c, rear, temp;
   int sum;
   MALLOC(rear, sizeof(*rear));
   c = rear;
   while (a && b)
      switch (COMPARE(a→expon,b→expon)) {
         case -1: /* a→expon < b→expon */
            attach(b→coef,b→expon,&rear);
            b = b→link;
            break;
         case 0: /* a→expon = b→expon */
            sum = a→coef + b→coef;
            if (sum) attach(sum,a→expon,&rear);
            a = a→link;   b = b→link; break;
         case 1: /* a→expon > b→expon */
            attach(a→coef,a→expon,&rear);
            a = a→link;
      }
   /* copy rest of list a and then list b */
   for (; a; a = a→link) attach(a→coef,a→expon,&rear);
   for (; b; b = b→link) attach(b→coef,b→expon,&rear);
   rear→link = NULL;
   /* delete extra initial node */
   temp = c; c = c→link;  free(temp);
   return c;
}
```

Program : Add two polynomials

```
void attach(float coefficient, int exponent,
            polyPointer *ptr)
{/* create a new node with coef = coefficient and expon =
    exponent, attach it to the node pointed to by ptr.
    ptr is updated to point to this new node */
   polyPointer temp;
   MALLOC(temp, sizeof(*temp));
   temp→coef = coefficient;
   temp→expon = exponent;
   (*ptr)→link = temp;
   *ptr = temp;
}
```

Program : Attach a node to the end of a list

Analysis of padd:

To determine the computing time of padd, first determine which operations contribute to the cost. For this algorithm, there are three cost measures:

(l) Coefficient additions

(2) Exponent comparisons

(3) Creation of new nodes for c

The maximum number of executions of any statement in padd is bounded above by m + n. Therefore, the computing time is O(m+n). This means that if we implement and run the algorithm on a computer, the time it takes will be $C_1m + C_2n + C_3$, where C1, C2, C3 are constants. Since any algorithm that adds two polynomials must look at each nonzero term at least once, padd is optimal to within a constant factor.

SPARSE MATRIX REPRESENTATION

A linked list representation for sparse matrices.

In data representation, each column of a sparse matrix is represented as a circularly linked list with a header node. A similar representation is used for each row of a sparse matrix.
Each node has a tag field, which is used to distinguish between header nodes and entry nodes.

Header Node:
- Each header node has three fields: down, right, and next as shown in figure (a).
- The down field is used to link into a column list and the right field to link into a row list.
- The next field links the header nodes together.
- The header node for row i is also the header node for column i, and the total number of header nodes is max {number of rows, number of columns}.

Element node:

- Each element node has five fields in addition in addition to the tag field: row, col, down, right, value as shown in figure (b).
- The down field is used to link to the next nonzero term in the same column and the right field to link to the next nonzero term in the same row. Thus, if $a_{ij} \neq 0$, there is a node with tag field = entry, value = a_{ij}, row = i, and col = j as shown infigure (c).
- We link this node into the circular linked lists for row i and column j. Hence, it is simultaneously linked into two different lists.

(a) header node   (b) element node

head field is not shown
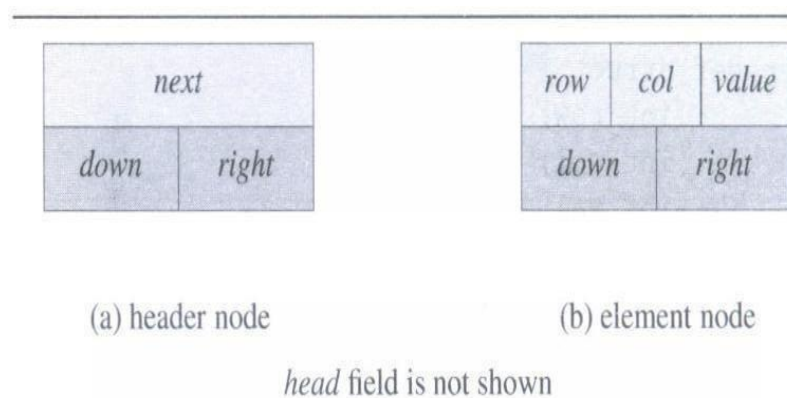
Figure: Node structure for sparse matrices

Consider the sparse matrix, as shown in below figure (2).

Figure(2): 4×4 sparse matrix a

Figure (3) shows the linked representation of this matrix. Although we have not shown the value of the tag fields, we can easily determine these values from the node structure.

For each nonzero term of a, have one entry node that is in exactly one row list and one column list. The header nodes are marked HO-H3. As the figure shows, we use the right field of the header node list header to link into the list of headernodes.

To represent a *numRows x numCols* matrix with *numTerms* nonzero terms, then we need max

{numRows, numCols} + *numTerms* + *1* nodes. While each node may require several words of memory, the total storage will be less than *numRows x numCols* when *numTerms* is sufficiently small.
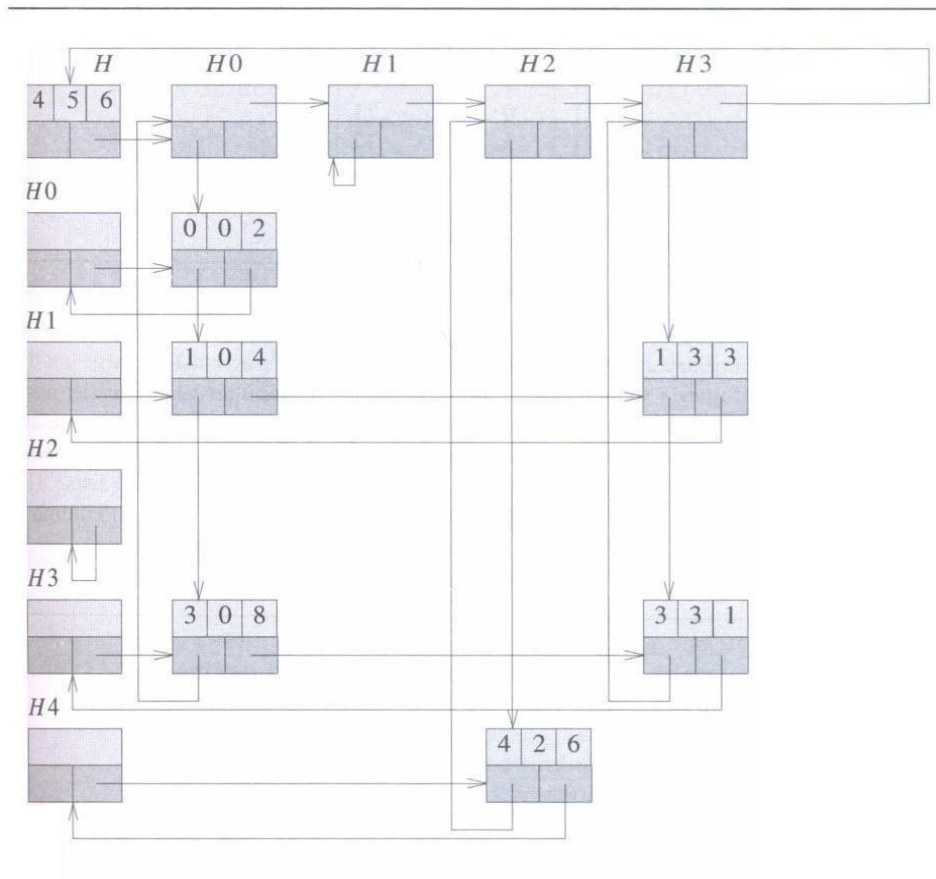
Figure (3) : Linked representation of the sparse matrix of Figure (2)   (the *head* field of a node is not shown)

There are two different types of nodes in representation, so unions are used to create the appropriate data structure. The C declarations are as follows:

```
#define MAX-SIZE 50 /*size of largest matrix*/ typedef enum {head, entry} tagfield;
    typedef struet matrixNode *matrixPointer;

    typedef strue {
                        int row; int
                        eol; intvalue;
                    } entryNode;

    typedef struet {
                        matrixPointer down;
                        matrixPointerright; tagfield
                        tag;

union {
                            matrixPointernext;
                            entryNode entry;
```

MODULE 4: TREES

DEFINITION

A *tree* is a finite set of one or more nodes such that

- There is a specially designated node called *root*.
- The remaining nodes are partitioned into $n >= 0$ disjoint set $T_1,\ldots,T_n$, where each of these sets is a tree. $T_1,\ldots,T_n$ are called the *subtrees* of the root.
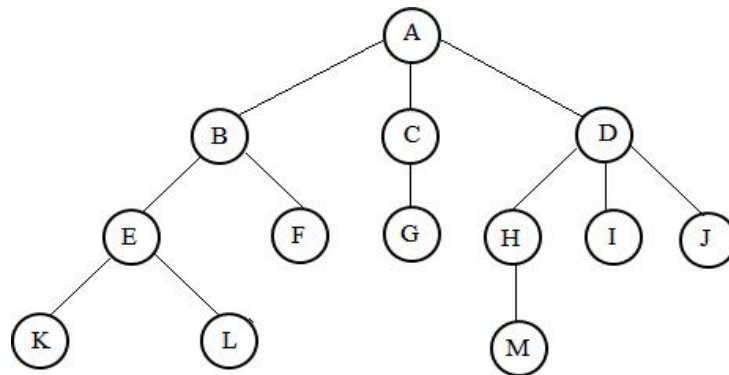
Every node in the tree is the root of some subtree

TERMINOLOGY

- *Node:* The item of information plus the branches to other nodes

- *Degree:* The number of subtrees of a node

- *Degree of a tree:* The maximum of the degree of the nodes in the tree.

- *Terminal nodes (or leaf):* nodes that have degree zero or node with no successor

- *Nonterminal nodes*: nodes that don't belong to terminal nodes.

- *Parent and Children:* Suppose N is a node in T with left successor S1 and right successor S2, then N is called the Parent (or father) of S1 and S2. Here, S1 is called left child (or Son) and S2 is called right child (or Son) of N.

- *Siblings:* Children of the same parent are said to be siblings.

- *Edge:* A line drawn from node N of a T to a successor is called an edge

- *Path:* A sequence of consecutive edges from node N to a node M is called a path.

- *Ancestors of a node:* All the nodes along the path from the root to that node.

- *The level of a node:* defined by letting the root be at level zero. If a node is at level l, then it children are at level $l+1$.

- *Height (or depth):* The maximum level of anynode in the tree

Example
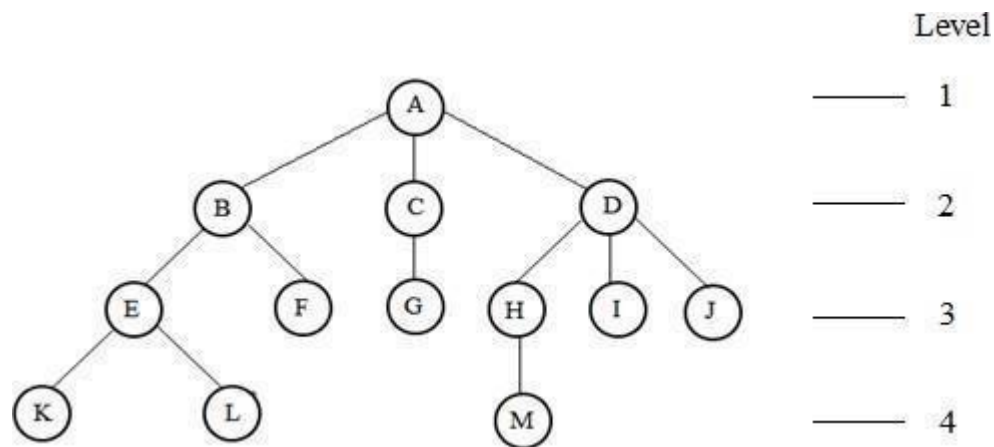
A is the root node
B is the parent of E and F
C and D are the sibling of B
E and F are the children of B
K, L, F, G, M, I, J are external nodes, or leaves
A, B, C, D, E, H are internal nodes
The level of E is 3
The height (depth) of the tree is 4
The degree of node B is 2
The degree of the tree is 3
The ancestors of node M is A, D, H
The descendants of node D is H, I, J, M

Representation of Trees

There are several ways to represent a given tree such as:
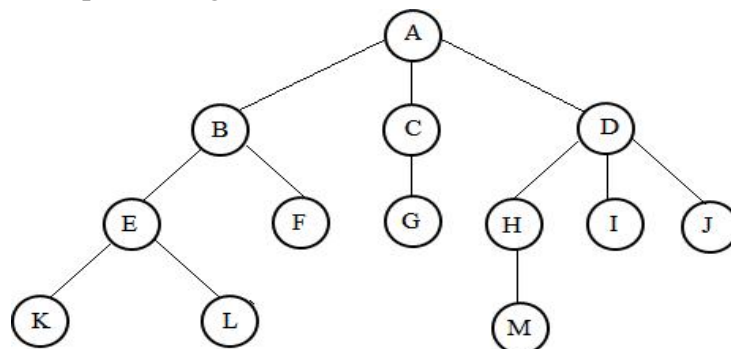
Figure (A)

1. List Representation
2. Left Child- Right Sibling Representation
3. Representation as a Degree-Two tree

List Representation:

The tree can be represented as a List. The tree of **figure (A)** could be written as the list.
(A (B (E (K, L), F), C (G), D (H (M), I, J)))

- The information in the root node comes first.
- The root node is followed by a list of the subtrees of that node.

Tree node is represented by a memory node that has fields for the data and pointers to the tree node's children
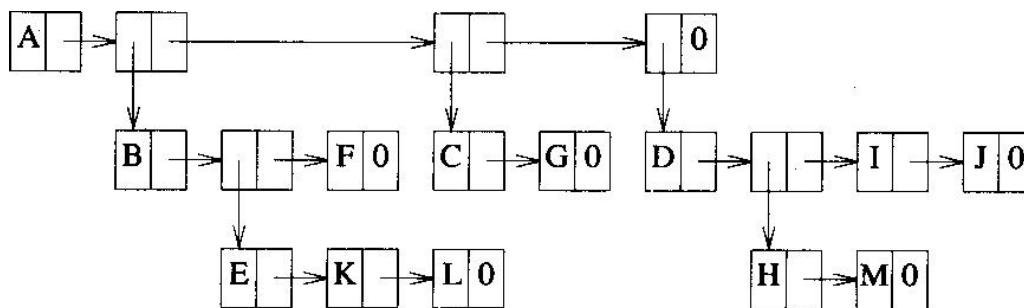
Figure (B): List representation of the tree of figure (A)

Since the degree of each tree node may be different, so memory nodes with a varying number of pointer fields are used.
For a tree of degree k, the node structure can be represented as below figure. Each child field is used to point to a subtree.

Left Child-Right Sibling Representation

The below figure show the node structure used in the left child-right sibling representation
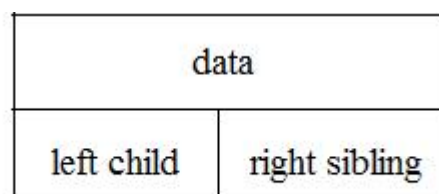
Figure (c): Left child right sibling node structure

To convert the tree of Figure (A) into this representation:
1. First note that every node has at most one leftmost child
2. At most one closest right sibling.

Ex:
- In Figure (A), the leftmost child of A is B, and the leftmost child of D is H.
- The closest right sibling of B is C, and the closest right sibling of H is I.
- Choose the nodes based on how the tree is drawn. The left child field of each node points to its leftmost child (if any), and the right sibling field points to its closest right sibling (if any).

Figure (D) shows the tree of Figure (A) redrawn using the left child-right sibling representation.
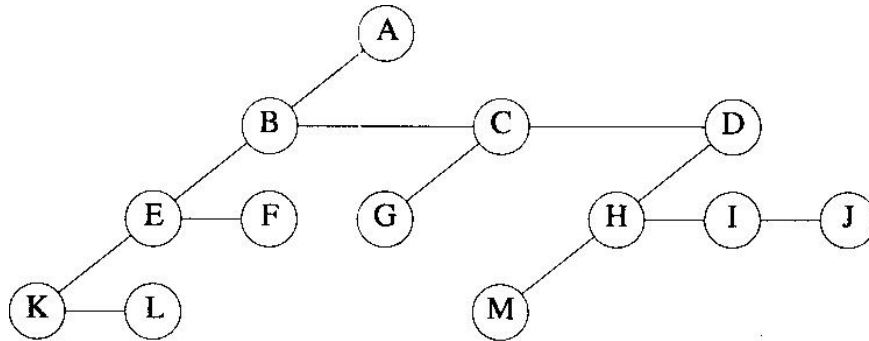
Figure (D): Left child-right sibling representation of tree of figure (A)

Representation as a Degree-Two Tree

To obtain the degree-two tree representation of a tree, simply rotate the right-sibling pointers in a left child-right sibling tree clockwise by 45 degrees. This gives us the degree-two tree displayed in Figure (E).
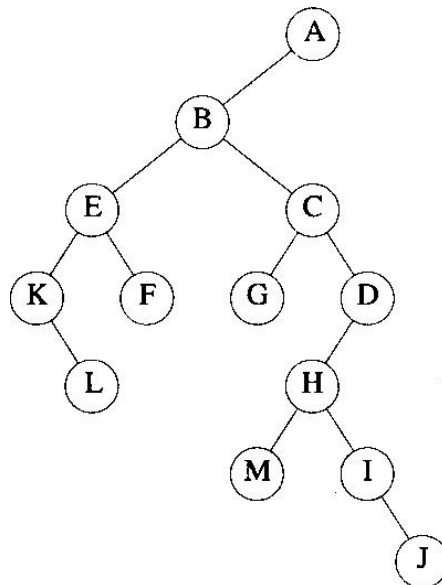
Figure (E): degree-two representation

In the degree-two representation, a node has two children as the left and right children.

BINARY TREES

Definition: A binary tree T is defined as a finite set of nodes such that,

- T is empty or
- T consists of a root and two disjoint binary trees called the left subtree and the right subtree.
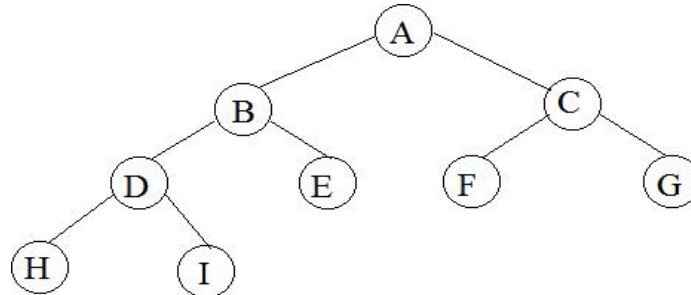
Figure: Binary Tree

Different kinds of Binary Tree

1. Skewed Tree

A skewed tree is a tree, skewed to the left or skews to the right.

                                    or

It is a tree consisting of only left subtree or only right subtree.

- A tree with only left subtrees is called Left Skewed Binary Tree.
- A tree with only right subtrees is called Right Skewed Binary Tree.

2. Complete Binary Tree

A binary tree T is said to complete if all its levels, except possibly the last level, have the maximum number node 2^i, $i \geq 0$ and if all the nodes at the last level appears as far left as possible.
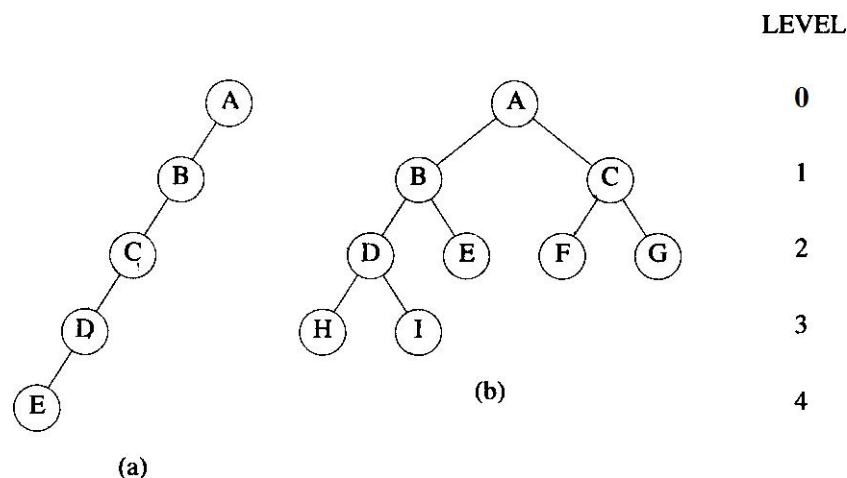
Figure (a): Skewed binary tree   Figure (b): Complete binarytree

3. Full Binary Tree

A full binary tree of depth 'k' is a binary tree of depth k having $2^k - 1$ nodes, $k \geq 1$.
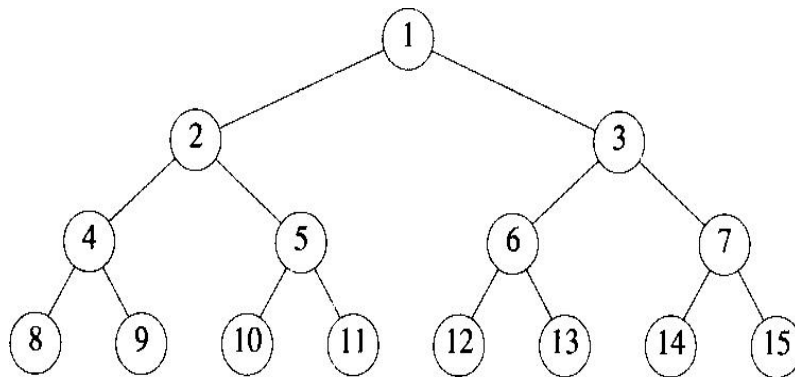
Figure: Full binary tree of level 4 with sequential node number

4. Extended Binary Trees or 2-trees

An *extended binary tree* is a transformation of any binary tree into a complete binary tree. This transformation consists of replacing every null subtree of the original tree with "special nodes." The nodes from the original tree are then *internal nodes*, while the special nodes are *external nodes*.
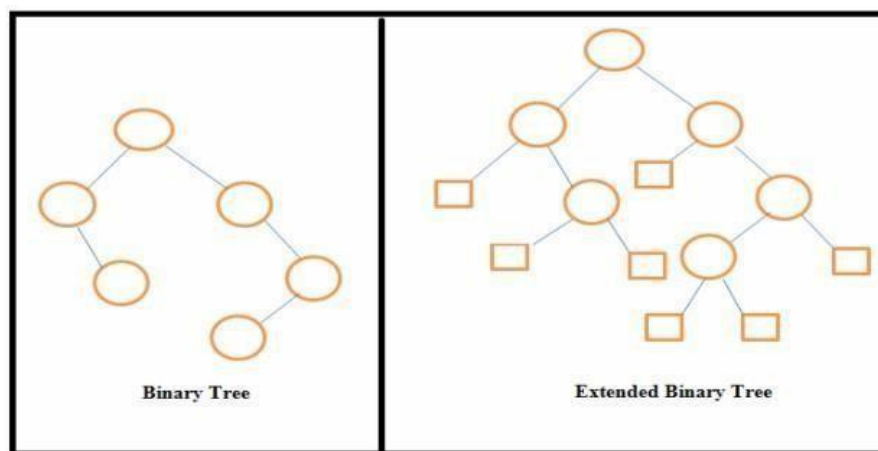
For instance, consider the following binary tree.

The following tree is its extended binary tree. The circles represent internal nodes, and square represent external nodes.

Every internal node in the extended tree has exactly two children, and every external node is a leaf. The result is a complete binary tree.

PROPERTIES OF BINARY TREES

Lemma 1: [Maximum number of nodes]:
(1) The maximum number of nodes on level i of a binary tree is 2^{i-1}, $i \geq 1$.
(2) The maximum number of nodes in a binary tree of depth k is $2^k - 1$, $k \geq 1$.

Proof:
(1) The proof is by induction on i.
Induction Base: The root is the only node on level i = 1. Hence, the maximum number of nodes on level i = 1 is $2^{i-1} = 2^0 = 1$.
Induction Hypothesis: Let i be an arbitrary positive integer greater than 1. Assume that the maximum number of nodes on level i - 1 is 2^{i-2}
Induction Step: The maximum number of nodes on level i - 1 is 2^{i-2} by the induction hypothesis. Since each node in a binary tree has a maximum degree of 2, the maximum number of nodes on level i is two times the maximum number of nodes on level i-1, or 2^{i-1}

(2) The maximum number of nodes in a binary tree of depth k is

$$\sum_{i=0}^{k} (\text{maximum number of nodes on level i}) = \sum_{i=0}^{k} 2^{i-1} = 2^{k-1}$$

Lemma 2: [Relation between number of leaf nodes and degree-2 nodes]:
For any nonempty binary tree, T, if n_0 is the number of leaf nodes and n_2 the number of nodes of degree 2, then $n_0 = n_2 + 1$.

Proof: Let n_1 be the number of nodes of degree one and n the total number of nodes.
Since all nodes in T are at most of degree two, we have

$$n = n_0 + n_1 + n_2 \qquad (1)$$

Count the number of branches in a binary tree. If B is the number of branches, then

$$n = B + 1.$$

All branches stem from a node of degree one or two. Thus,

$$B = n_1 + 2n_2.$$

Hence, we obtain

$$n = B + 1 = n_1 + 2n_2 + 1 \qquad (2)$$

Subtracting Eq. (2) from Eq. (1) and rearranging terms, we get

$$\mathbf{n_0 = n_2 + 1}$$

Consider the figure:
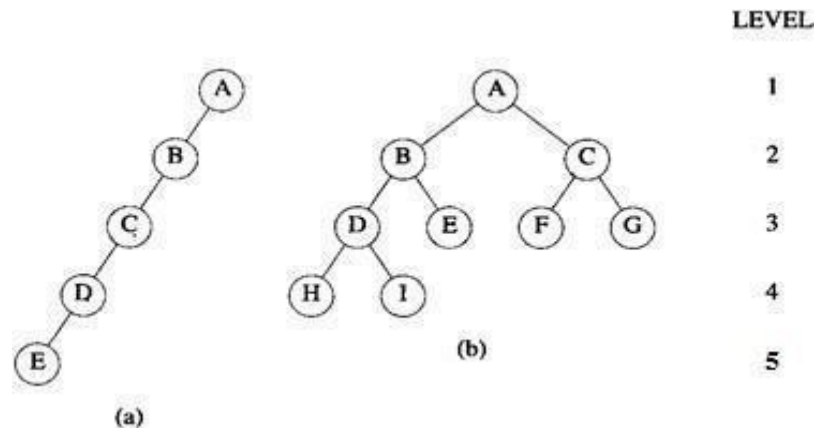
Here, For Figure (b) $n_2=4$, $n_0= n_2+1= 4+1=5$
Therefore, the total number of leaf node=5

BINARY TREE REPRESENTATION
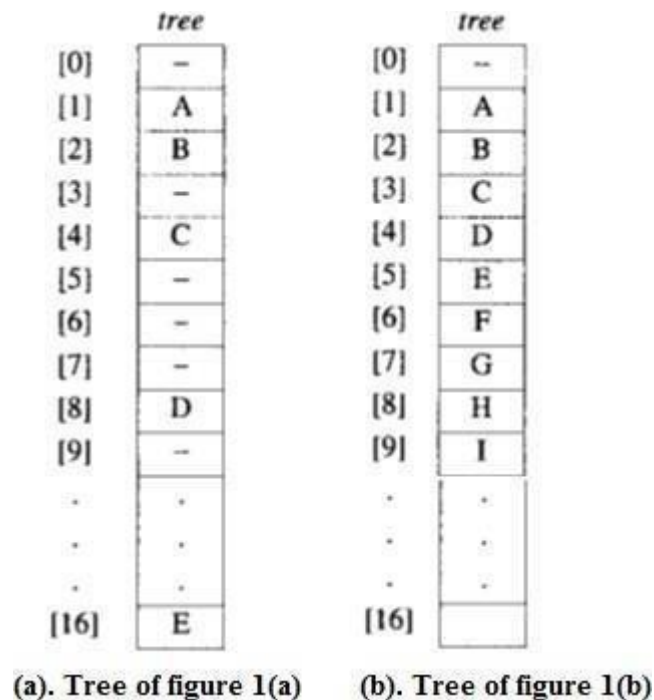The storage representation of binary trees can be classified as
1. Array representation
2. Linked representation.

Array representation:
- A tree can be represented using an array, which is called sequential representation.
- The nodes are numbered from 1 to n, and one dimensional array can be used to store the nodes.
- Position 0 of this array is left empty and the node numbered *i* is mapped to position *i* of the array.

Below figure shows the array representation for both the trees of figure (a).

Figure 1(a) Skewed binary tree

Figure 1(b) Complete binary tree

	tree			tree
[0]	–		[0]	--
[1]	A		[1]	A
[2]	B		[2]	B
[3]	–		[3]	C
[4]	C		[4]	D
[5]	–		[5]	E
[6]	–		[6]	F
[7]	–		[7]	G
[8]	D		[8]	H
[9]	–		[9]	I
.	.		.	.
.	.		.	.
.	.		.	.
[16]	E		[16]	

(a). Tree of figure 1(a)   (b). Tree of figure 1(b)
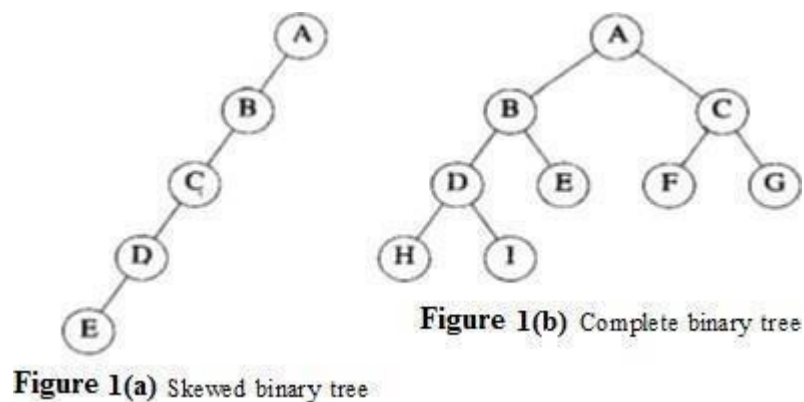
- For complete binary tree the array representation is ideal, as no space is wasted.
- For the skewed tree less than half the array is utilized.

Linked representation:

The problems in array representation are:
- It is good for complete binary trees, but more memory is wasted for skewed and many other binary trees.
- The insertion and deletion of nodes from the middle of a tree require the movement of many nodes to reflect the change in level number of these nodes.

These problems can be easily overcome by linked representation

Each node has three fields,
- LeftChild   - which contains the address of left subtree
- RightChild - which contains the address of right subtree.
- Data   - which contains the actual information

C Code for node:

```
typedef  struct  node  *treepointer;
typedef struct {
              int data;
              treepointer leftChild, rightChild;
        }node;
```
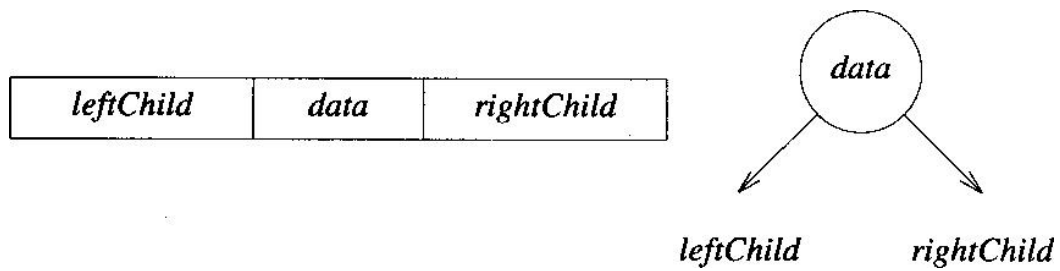
Figure: Node representation

Figure 1(a) Skewed binary tree

Figure 1(b) Complete binary tree

(a)

(b)
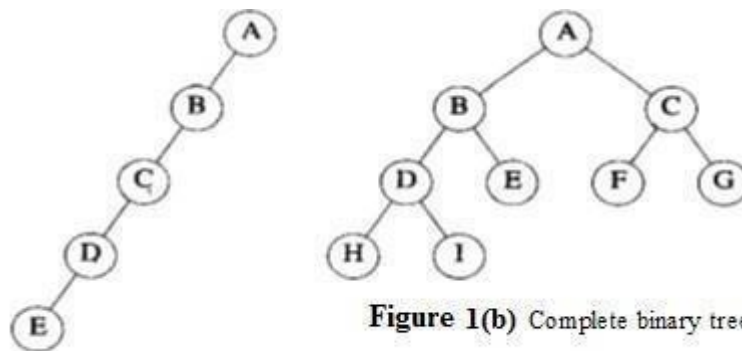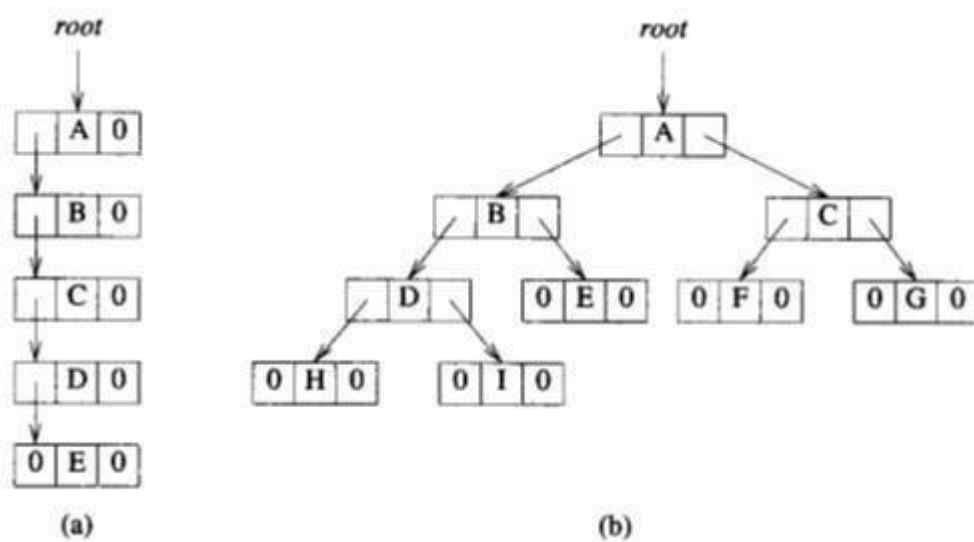
Linked representation of the binary tree

BINARY TREE TRAVERSALS

Visiting each node in a tree exactly once is called tree traversal

The different methods of traversing a binary tree are:
1. Preorder
2. Inorder
3. Postorder
4. Iterative inorder Traversal
5. Level-Order traversal

1. **Inorder:** Inorder traversal calls for moving down the tree toward the left until you cannot go further. Then visit the node, move one node to the right and continue. If no move can be done, then go back one more node.

Let ptr is the pointer which contains the location of the node N currently being scanned.
L(N) denotes the leftchild of node N and R(N) is the right child of node N
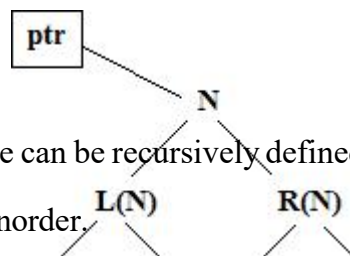
Recursion function:
The inorder traversal of a binary tree can be recursively defined as

- Traverse the left subtree in inorder.
- Visit the root.
- Traverse the right subtree in inorder.

```
void inorder(treepointerptr)
{
        if(ptr)
        {
                inorder    (ptr→leftchild);
                printf    ("%d",ptr→data);
                inorder (ptr→rightchild);
        }
}
```

2. **Preorder:** Preorder is the procedure of visiting a node, traverse left and continue. When you cannot continue, move right and begin again or move back until you can move right and resume.

Recursion function:

The Preorder traversal of a binary tree can be recursively defined as

- Visit the root
- Traverse the left subtree in preorder.
- Traverse the right subtree in preorder

```
void preorder (treepointerptr)
{
      if(ptr)
      {
              printf      ("%d",ptr→data)
              preorder    (ptr→leftchild);
              preorder (ptr→rightchild);
          }
  }
```

3. **Postorder:** Postorder traversal calls for moving down the tree towards the left until you can go no further. Then move to the right node and then visit the node and continue.

Recursion function:

The Postorder traversal of a binary tree can be recursively defined as

- Traverse the left subtree in postorder.
- Traverse the right subtree in postorder.
- Visit the root

```
void postorder(treepointerptr)
{
      if(ptr)
      {
              postorder   (ptr→leftchild);
              postorder (ptr→rightchild);
              printf ("%d",ptr→data);
          }
  }
```

4. Iterative inorder Traversal:

Iterative inorder traversal explicitly make use of stack function.

The left nodes are pushed into stack until a null node is reached, the node is then removed from the stack and displayed, and the node's right child is stacked until a null node is reached. The traversal then continues with the left child. The traversal is complete when the stack is empty.

```
void iterInorder(treePointer node)
{
    int top = -1; /* initialize stack */
    treePointer stack[MAX_STACK_SIZE];
    for (;;) {
        for(; node; node = node→leftChild)
            push(node); /* add to stack */
        node = pop(); /* delete from stack */
        if (!node) break; /* empty stack */
        printf("%d", node→data);
        node = node→rightChild;
    }
}
```

Program   : Iterative inorder traversal

5. Level-Order traversal:

Visiting the nodes using the ordering suggested by the node numbering is called level ordering traversing.

The nodes in a tree are numbered starting with the root on level 1 and so on.

Firstly visit the root, then the root's left child, followed by the root's right child. Thus continuing in this manner, visiting the nodes at each new level from the leftmost node to the rightmost node.
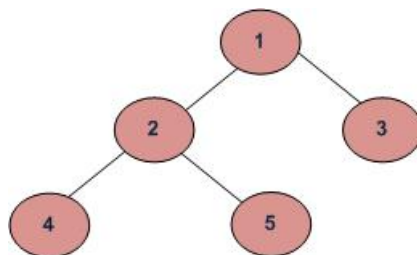
Level order traversal: 1 2 3 4 5

        Initially in the code for level order add the root to the queue. The function operates by deleting the node at the front of the queue, printing the nodes data field and adding the nodes left and right children to the queue.

Function for level order traversal of a binary tree:

```
void levelOrder(treePointer ptr)
{/* level order tree traversal */
    int front = rear = 0;
    treePointer queue[MAX_QUEUE_SIZE];
    if (!ptr) return; /* empty tree */
    addq(ptr);
    for (;;) {
        ptr = deleteq();
        if (ptr) {
            printf("%d",ptr→data);
            if(ptr→leftChild)
                addq(ptr→leftChild);
            if (ptr→rightChild)
                addq(ptr→rightChild);
        }
        else break;
    }
}
```

Program   : Level-order traversal of a binary tree

ADDITIONAL BINARY TREE OPERATIONS

1. Copying a Binarytree
This operations will perform a copying of one binary tree to another.

C function to copy a binary tree:

```
treepointer copy(treepointer original)
{ if(original)
    {   MALLOC(temp,sizeof(*temp));
        temp→leftchild=copy(original→leftchild);
        temp→rightchild=copy(original→rightchild);
        temp→data=original→data;
        return temp;
    }
    return NULL;
}
```

2. Testing Equality
This operation will determin the equivalance of two binary tree. Equivalance binary tree have the same strucutre and the same information in the corresponding nodes.

C function for testing equality of a binary tree:

int equal(treepointer first,treepointer second)

{

return((!first && !second) || (first && second && (first→data==second→data)
&& equal(first→leftchild,second→leftchild) && equal(first→rightchild,
second→rightchild))

}

This function will return TRUE if two trees are equivalent and FALSE if they are not.

3. The Satisfiability problem

- Consider the formula that is constructed by set of variables: x_1, x_2, …, x_n and operators $\Box$(*and*), $\Box$(*or*), $\neg$ (*not*).
- The variables can hold only of two possible values, *true* or *false*.
- The expression can form using these variables and operators is defined by the following rules.
    - A variable is an expression
    - If *x* and *y* are expressions, then $\neg x$, $x \Box y$, $x \Box y$ are expressions
    - Parentheses can be used to alter the normal order of evaluation ($\neg > \Box > \Box$)

Example: $x_1 \Box (x_2 \Box \neg x_3)$ If x_1 and x_3 are *false* and x_2 is *true*

$= false \Box (true \Box \neg false)$

$= false \Box\ true$

$= true$

The satisfiablity problem for formulas of the propositional calculus asks if there is an assignment of values to the variable that causes the value of the expression to be *true.*

Let's assume the formula in a binary tree

$(x_1 \Box \neg x_2) \Box (\neg x_1 \Box x_3) \Box \neg x_3$
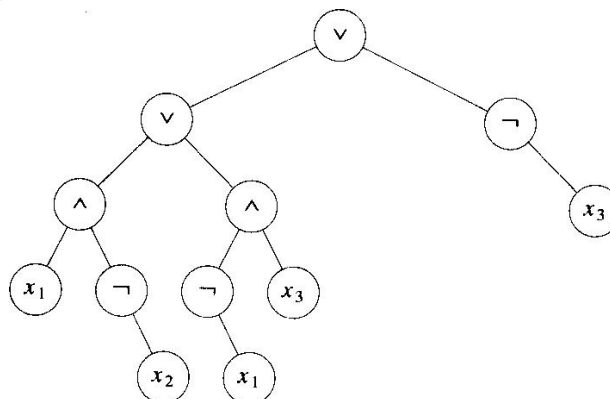
Figure : Propositional formula in a binary tree

The inorder traversal of this tree is

$$x_1 \ \Box \ \neg x_2 \ \Box \ \neg \ x_1 \ \Box \ x_3 \ \Box \ \neg x_3$$

The algorithm to determine satisfiablity is to let (x_1, x_2, x_3) takes on all the possible combination of true and false values to check the formula for each combination.

For *n* value of an expression, there are 2^n possible combinations of *true* and *false*

For example n=3, the eight combinations are (t,t,t), (t,t,f), (t,f,t), (t,f,f), (f,t,t), (f,t,f), (f,f,t), (f,f,f).

The algorithm will take $O(g \ 2^n)$, where g is the time to substitute values for $x_1, x_2, \ldots x_n$ and evaluate the expression.

Node structure:

For the purpose of evaluation algorithm, assume each node has four fields:

left_child	data	value	right_child

Figure : Node structure for the satisfiability problem

Define this node structure in C as:

```
typedef enum {not,and,or,true,false} logical;
typedef struct node *tree_pointer;
typedef struct node {
        tree_pointer left_child;
        logical      data;
        short int    value;
        tree_pointer right_child;
        } ;
```

Satisfiability function: The first version of Satisfiability algorithm

```
for (all 2ⁿ possible combinations) {
   generate the next combination;
   replace the variables by their values;
   evaluate root by traversing it in postorder;
   if (root→value) {
      printf(<combination>);
      return;
   }
}
printf("No satisfiable combination\n");
```

THREADED BINARY TREE

The limitations of binary tree are:
- In binary tree, there are n+1 null links out of 2n totallinks.
- Traversing a tree with binary tree is time consuming.

These limitations can be overcome by threaded binary tree.

In the linked representation of any binary tree, there are more null links than actual pointers. <u>These null links are replaced by the pointers, called **threads**, which points to other nodes in the tree.</u>

To construct the threads use the following rules:
1. Assume that **ptr** represents a node. If ptr→leftChild is null, then replace the null link with a pointer to the inorder predecessor of ptr.
2. If ptr →rightChild is null, replace the null link with a pointer to the inorder successor of ptr.

Ex: Consider the binary tree as shown in below figure:
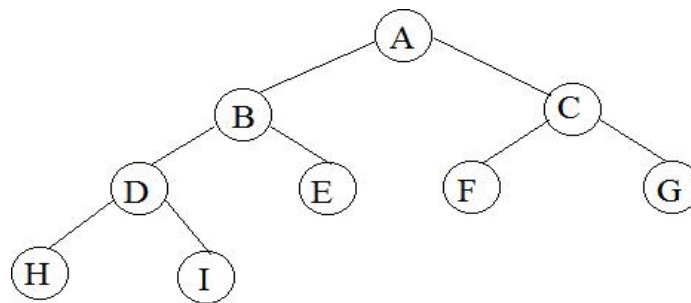
Figure A: Binary Tree

There should be no loose threads in threaded binary tree. But in **Figure B** two threads have been left dangling: one in the left child of *H,* the other in the right child of G.
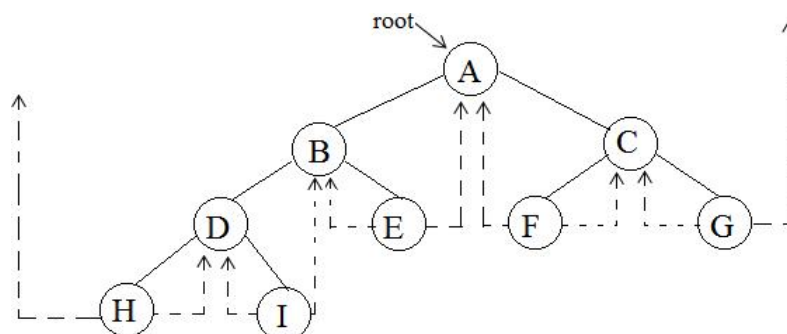
Figure B: Threaded tree corresponding to Figure A

In above figure the new threads are drawn in broken lines. This tree has 9 node and 10 0 -links which has been replaced by threads.

When trees are represented in memory, it should be able to distinguish between threads and pointers. This can be done by adding two additional fields to node structure, ie., *leftThread* and *rightThread*

- If ptr→leftThread = TRUE, then ptr→leftChild contains a thread, otherwise it contains a pointer to the left child.
- If ptr→rightThread = TRUE, then ptr→rightChild contains a thread, otherwise it contains a pointer to the rightchild.

Node Structure:
The node structure is given in C declaration

```
typedef struct threadTree *threadPointer typedef struct{
            short int leftThread; threadPointer
            leftChild; char data;
            threadPointer    rightChild;    short    int
            rightThread;
        }threadTree;
```
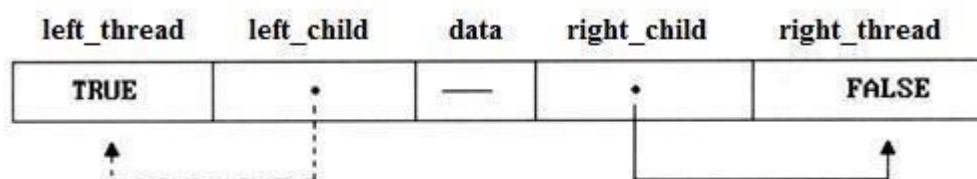

Figure An empty threaded tree

The complete memory representation for the tree of figure is shown in Figure C
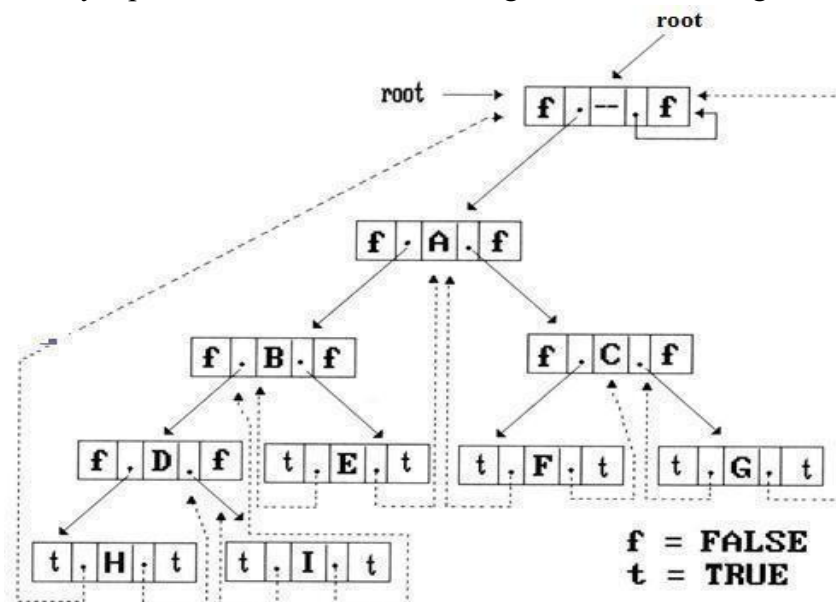
The variable *root* points to the header node of the tree, while **root →leftChild** points to the start of the first node of the actual tree. This is true for all threaded trees. Here the problem of the loose threads is handled by pointing to the head node called *root*.

Inorder Traversal of a Threaded Binary Tree

- Byusing the threads, an inorder traversal can be performed without making use of a stack.
- For any node, **ptr,** in a threaded binary tree, if **ptr→rightThread =TRUE,** the inorder successor of **ptr** is **ptr →rightChild** by definition of the threads. Otherwise we obtain the inorder successor of **ptr** by following a path of **left-child links** from the **right- child** of **ptr** until we reach a node with **leftThread = TRUE.**
- The function insucc **( )** finds the inorder successor of any node in a threaded tree without using a stack.

```
threadedpointer insucc(threadedPointer tree)
{       /* find the inorder successor of tree in a threaded binary tree */
         threadedpointer temp;
        temp = tree→rightChild;
        if (!tree→rightThread)
           while (!temp→leftThread)
                   temp = temp→leftChild;
        return temp;
```
Program: Finding inorder successor of a node

To perform inorder traversal make repeated calls to insucc ( ) function

```
void tinorder (threadedpointer tree)
{
Threadedpointer temp = tree;
for(; ;){
                temp = insucc(temp);
                if (temp == tree)      break;
                printf("%3c", temp→data
     }
}
```
**Program: Inorder traversal of a threaded binary tree**

Inserting a Node into a Threaded Binary Tree

In this case, the insertion of **r** as the right child of a node **s** is studied.

The cases for insertion are:

- If *s* has an **empty** right subtree, then the insertion is simple and diagrammed in Figure
- If the right subtree of *s* is **not empty**, then this right subtree is made the right subtree of *r* after insertion. When this is done, r becomes the inorder predecessor of a node that has a **leftThread == true** field, and consequently there is a thread which has to be updated to point to r. The node containing this thread was previously the inorder successor of s.

(a)

before   (b)   after
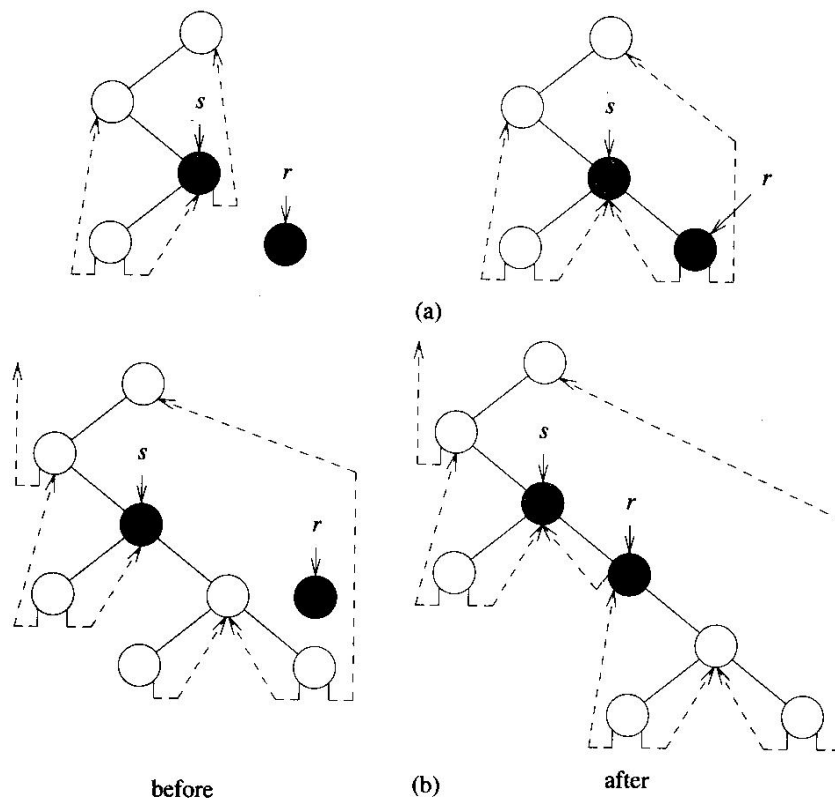
```
void insertRight(threadedPointer Sf threadedPointer r)
{    /* insert r as the right child of s */
        threadedpointer              temp;
        r→rightChild = parent→rightChild;
        r→rightThread     =     parent→rightThread;
        r→leftChild = parent;
        r→leftThread   =   TRUE;
        s→rightChild   =   child;
        s→rightThread = FALSE;
        if (!r→rightThread) {
                temp   =   insucc(r);
                temp→leftChild = r;
        }
}
```

Graphs

Definitions

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as **vertices,** and the links that connect the vertices are called **edges**.

Formally, a graph is a pair of sets **(V, E),** where **V** is the set of vertices and **E** is the set of edges, connecting the pairs of vertices. Take a look at the following graph −
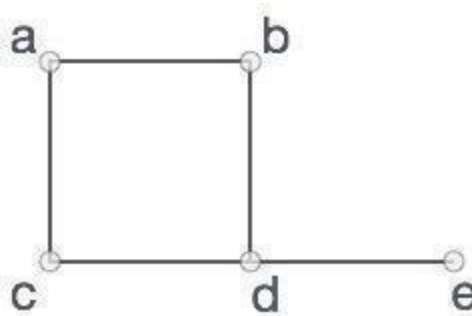

In the

above

graph,

V = {a,

b, c, d, e}

E = {ab, ac, bd, cd, de}

Terminologies and Matrix and Adjacency List Representation of graphs

Graph Data Structure

Mathematical graphs can be represented in data-structure. We can represent a graph using an array of vertices and a two dimensional array of edges. Before we proceed further, let's familiarize ourselves with some important terms −

- **Vertex** − Each node of the graph is represented as a vertex. In example given below, labeled circle represents vertices. So A to G are vertices. We can represent them using an array as shown in image below. Here A can be identified by index 0. B can be identified using index 1 and so on.

- **Edge** − Edge represents a path between two vertices or a line between two vertices. In example given below, lines from A to B, B to C and so on represents edges. We can use a two dimensional array to represent array as shown in image below. Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as 0.

- **Adjacency** − Two node or vertices are adjacent if they are connected to each other through an edge. In example given below, B is adjacent to A, C is adjacent to B and so on.

- **Path** − Path represents a sequence of edges between two vertices. In example given below, ABCD represents a path from A to D.
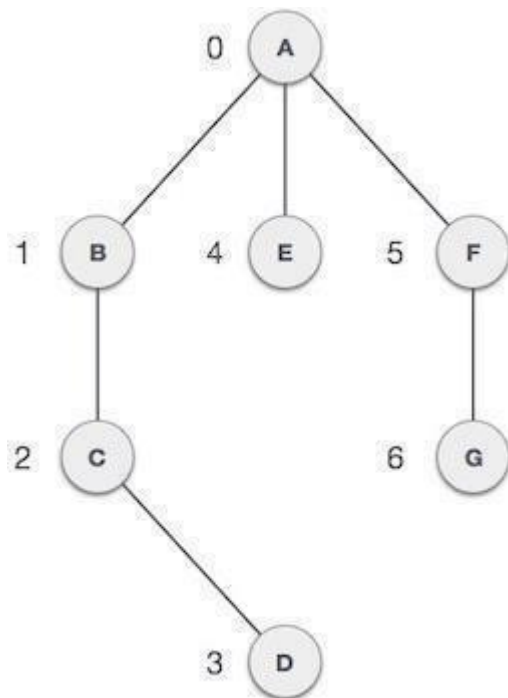
Elementary Graph Operations
Basic Operations

Following are basic primary operations of a Graph which are following.
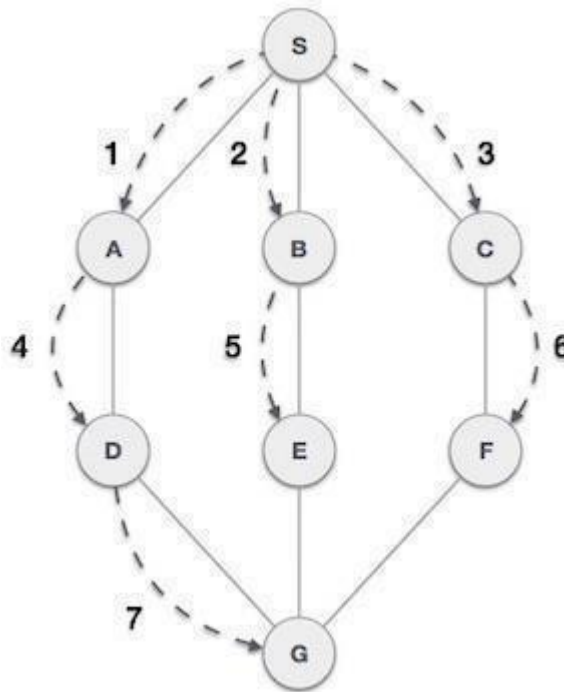
- **Add Vertex** − add a vertex to a graph.

- **Add Edge** − add an edge between two vertices of a graph.

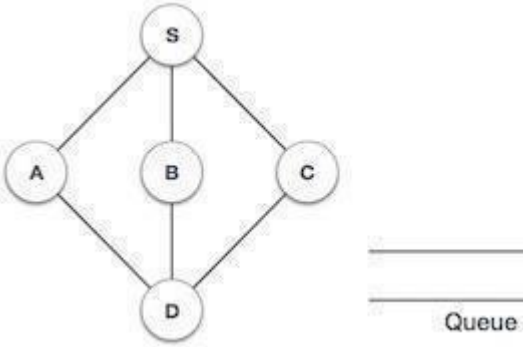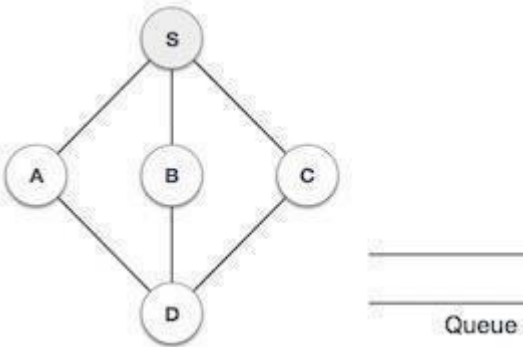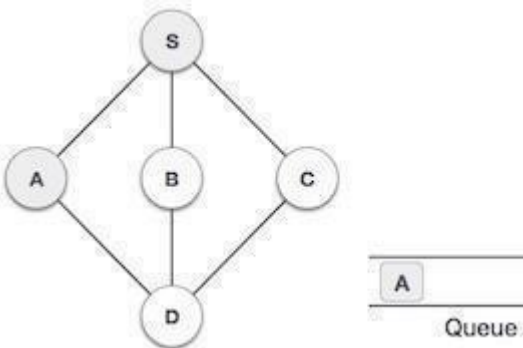- **Display Vertex** − display a vertex of a graph.

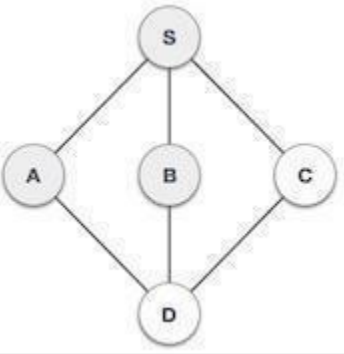## *Traversal methods*

Breadth First Search
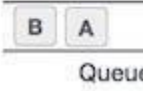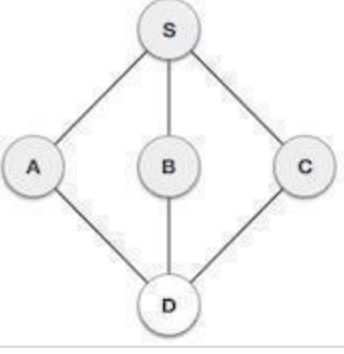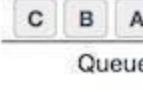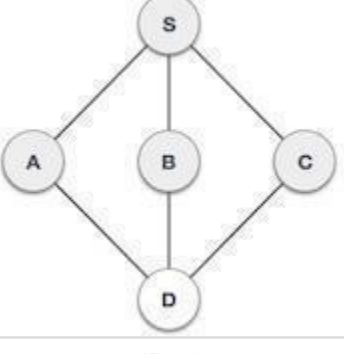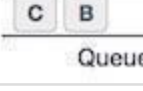
Breadth First Search algorithm(BFS) traverses a graph in a breadthwards motion and uses a queue to remember to get the next vertex to start a search when a dead end occurs in any iteration.

As in example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs following rules.

- **Rule 1** − Visit adjacent unvisited vertex. Mark it visited. Display it. Insert it in a queue.

- **Rule 2** − If no adjacent vertex found, remove the first vertex from queue.

- **Rule 3** − Repeat Rule 1 and Rule 2 until queue is empty.

Traversal	Description
	Initialize the queue.
	We start from visiting **S**(starting node), and mark it visited.
	We then see unvisited adjacent node from **S**. In this example, we have three nodes but alphabetically we choose **A** mark it visited and enqueue it.

	  B A Queue	Next unvisited adjacent node from **S** is **B**. We mark it visited and enqueue it.
	  C B A Queue	Next unvisited adjacent node from **S** is **C**. We mark it visited and enqueue it.
	  C B Queue	Now **S** is left with no unvisited adjacent nodes. So we dequeue and find **A**.
	  D C B Queue	From **A** we have **D** as unvisited adjacent node. We mark it visited and enqueue it.

At this stage we are left with no unmarked (unvisited) nodes. But as per algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied the program is over.
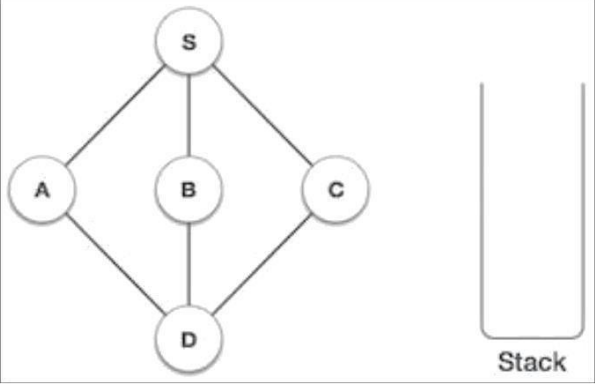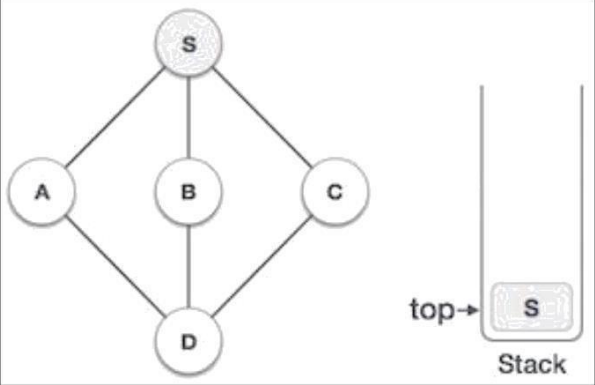
Depth First Search

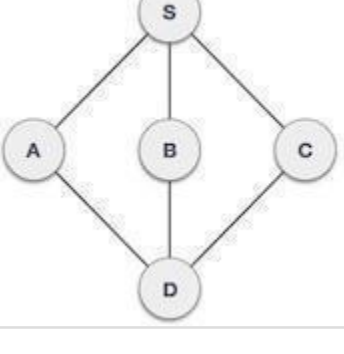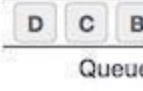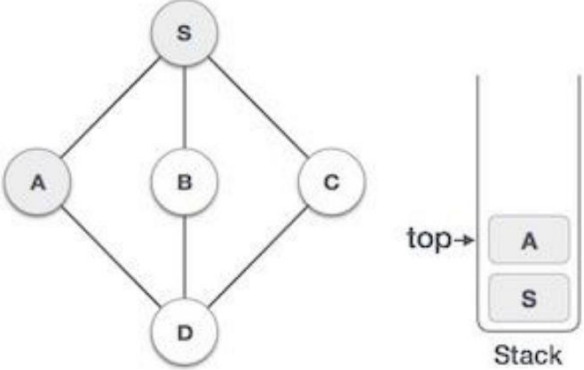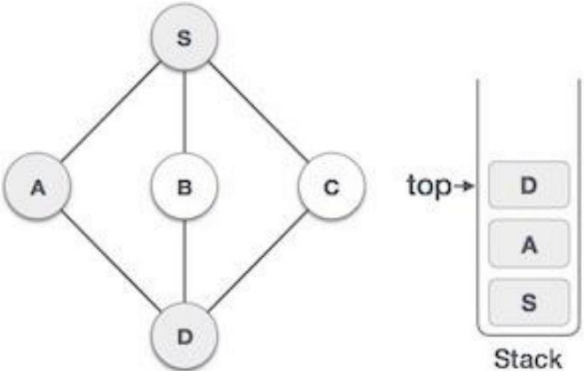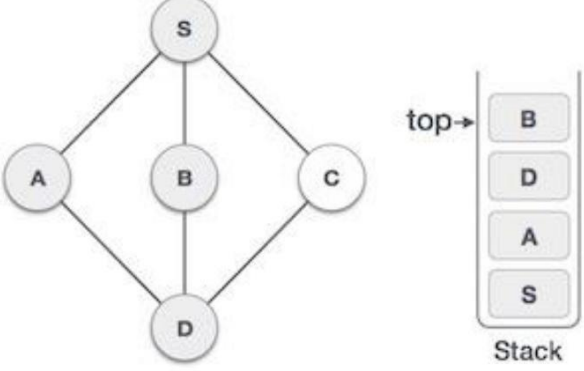Depth First Search algorithm(DFS) traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search when a dead end occurs in any iteration.
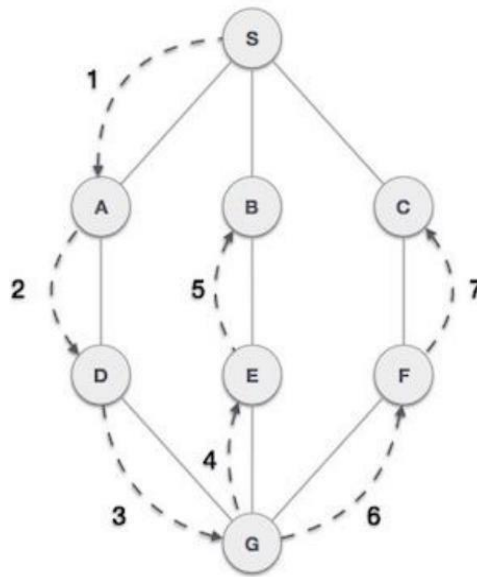
As in example given above, DFS algorithm traverses from A to B to C to D first then to E, then to F and lastly to G. It employs following rules.

- **Rule 1** − Visit adjacent unvisited vertex. Mark it visited. Display it. Push it in a stack.

- **Rule 2** − If no adjacent vertex found, pop up a vertex from stack. (It will pop up all the vertices from the stack which do not have adjacent vertices.)

- **Rule 3** − Repeat Rule 1 and Rule 2 until stack is empty.

Traversal	Description
	Initialize the stack
	Mark **S** as visited and put it onto the stack. Explore any unvisited adjacent node from **S**. We have three nodes and we can pick any of them. For this example, we shall take the node in alphabetical order.

	Mark **A** as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both **S** and **D** are adjacent to **A** but we are concerned for unvisited nodes only.
	Visit **D** and mark it visited and put onto the stack. Here we have **B** and **C** nodes which are adjacent to **D** and both are unvisited. But we shall again choose in alphabetical order.
	We choose **B**, mark it visited and put onto stack. Here **B** does not have any unvisited adjacent node. So we pop **B** from the stack.
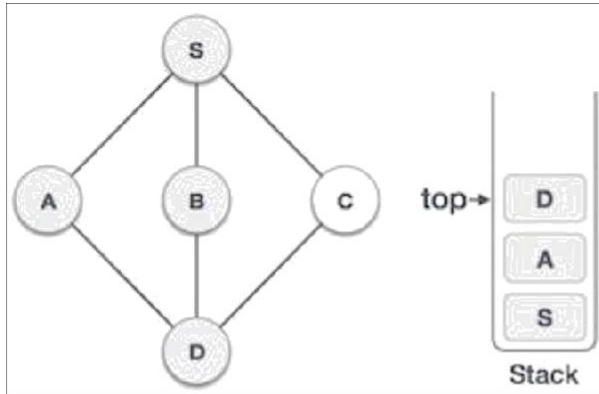
6.		We check stack top for return to previous node and check if it has any unvisited nodes. Here, we find **D** to be on the top of stack.
7.		Only unvisited adjacent node is from **D** is **C** now. So we visit **C**, mark it visited and put it onto the stack.
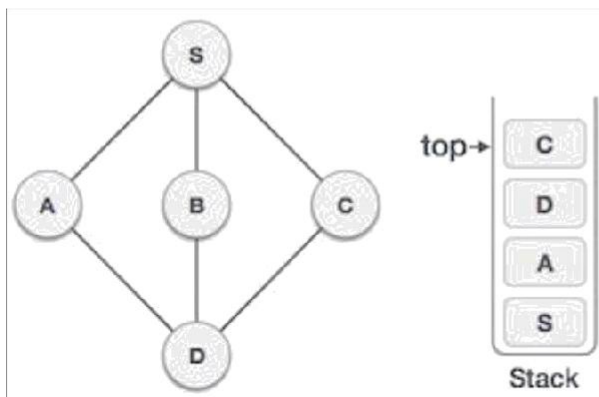
As **C** does not have any unvisited adjacent node so we keep popping the stack until we find a node which has unvisited adjacent node. In this case, there's none and we keep popping until stack is empty.

MODULE 5: HASHING

The Hash Table organizations

If we have a collection of **n** elements whose keys are unique integers in (1,**m**), where **m** >= **n**, then we can store the items in a *direct address* table, **T[m]**,

where **Ti** is either empty or contains one of the elements of our collection.

Searching a direct address table is clearly an **O(1)** operation: for a key, **k**, we access **Tk**,

- if it contains an element, return it,
  - if it doesn't then

return a NULL. There are

two constraints here:

1. the keys must be unique, and
2. the range of the key must be severely bounded.

direct access table
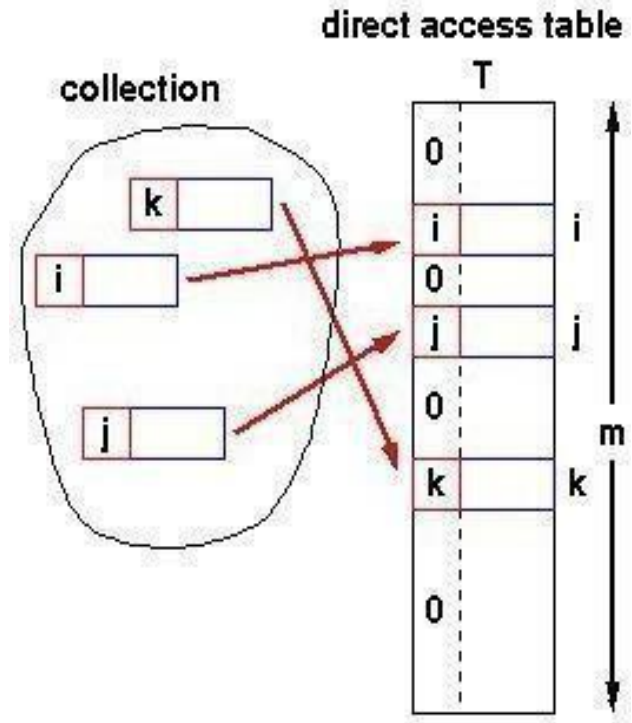
collection

If the keys are not unique, then we can simply construct a set of **m** lists and store the heads of these lists in the direct address table. The time to find an element matching an input key will still be O (1).

However, if each element of the collection has some other distinguishing feature (other than its key), and if the maximum number mofaxduplicates

duplicates are the exception rather than the rule, tdhuepnn max is much sma s table
performance. But if n_{dup}^{max} approaches **n**, then the time to find a specific element is
O(n) and a tree structure will be more efficient.

max). If

is ndup , then searching for a specific element isduOp (n
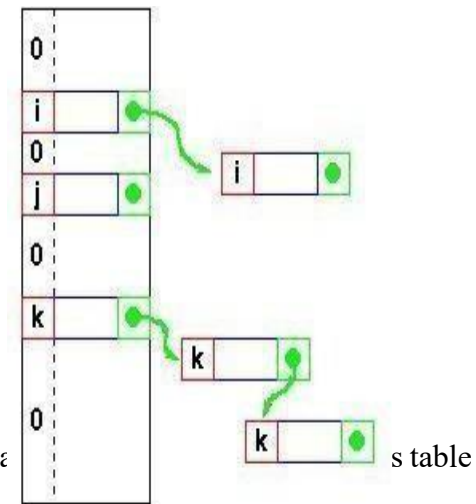
The range of the key determines the size of the direct address table and may be too large to be practical. For instance it's not likely that you'll be able to use a direct address table to store elements which have arbitrary 32- bit integers as their keys for a few years yet!

Direct addressing is easily generalised to the case where there is a function,

h(k) => (1,**m**)

which maps each value of the key, **k**, to the range (1,**m**). In this case, we place the element in **T[h(k)]** rather than **T[k]** and we can search in **O(1)** time as before.

Hashing Functions

The following functions map a single integer key (k) to a small integer bucket value h(k). m is the size of the hash table (number of buckets).

Division method (Cormen) Choose a prime that isn't close to a power of 2. h(k) = k mod m. Works badly for many types of patterns in the input data.

Knuth Variant on Division h(k) = k(k+3) mod m. Supposedly works much better than the raw division method.

Multiplication Method (Cormen). Choose m to be a power of 2. Let A be some random-looking real number. Knuth suggests M = 0.5*(sqrt(5) - 1). Then do the following:

s = k*A
x = fractional part
of s h(k) =
floor(m*x)

This seems to be the method that the theoreticians like.

To do this quickly with integer arithmetic, let w be the number of bits in a word (e.g. 32) and suppose m is 2^p. Then compute:

s =
floor(A *
2^w) x =
k*s
h(k) = x >> (w-p) // i.e. right shift x by (w-p) bits
// i.e. extract the p most significant
// bits from x

Static and Dynamic Hashing

The good functioning of a hash table depends on the fact that the table size is proportional to the number of entries. With a fixed size, and the common structures, it is similar to linear search, except with a better constant factor. In some cases, the number of entries may be definitely known in advance, for example keywords in a language. More commonly, this is not known for sure, if only due to later changes in code and data. It is one serious, although common, mistake to not provide *any* way for the table to resize. A general-purpose hash table "class" will almost always have some way to resize, and it is good practice even for simple "custom" tables. An implementation should check the load factor, and do something if it becomes too large (this needs to be done only on inserts, since that is the only thing that would increase it).

To keep the load factor under a certain limit, e.g., under 3/4, many table implementations expand the table when items are inserted. For example, in Java's HashMap class the default load factor threshold for table expansion is 3/4 and in Python's dict , table size is resized when load factor is greater than 2/3.

Since buckets are usually implemented on top of a dynamic array and any constant proportion for resizing greater than 1 will keep the load factor under the desired limit, the exact choice of the constant is determined by the same space-time tradeoff as for dynamic arrays.

Resizing is accompanied by a full or incremental table *rehash* whereby existing items are mapped to new bucket locations.

To limit the proportion of memory wasted due to empty buckets, some implementations also shrink the size of the table—followed by a rehash—when items are deleted. From the point of space-time tradeoffs, this operation is similar to the deallocation in dynamic arrays.

Resizing by copying all entries

A common approach is to automatically trigger a complete resizing when the load factor exceeds some
threshold r_{max}. Then a new larger table is allocated, all the entries of the old table are removed and inserted into this new table, and the old table is returned to the free storage pool. Symmetrically, when the load factor falls below a second threshold r_{min}, all entries are moved to a new smaller table.

For hash tables that shrink and grow frequently, the resizing downward can be skipped entirely. In this case, the table size is proportional to the maximum number of entries that ever were in the hash table at one time, rather than the current number. The disadvantage is that memory usage will be higher, and thus cache behavior may be worse. For best control, a "shrink-to-fit" operation can be provided that does this only on request.

If the table size increases or decreases by a fixed percentage at each expansion, the total cost of these resizings, amortized over all insert and delete operations, is still a constant, independent of the number

of entries n and of the number m of operations performed.

For example, consider a table that was created with the minimum possible size and is doubled each time the load ratio exceeds some threshold. If m elements are inserted into that table, the total number of extra re- insertions that occur in all dynamic resizings of the table is at most $m - 1$. In other words, dynamic resizing roughly doubles the cost of each insert or delete operation.

Incremental resizing

Some hash table implementations, notably in real-time systems, cannot pay the price of enlarging the hash table all at once, because it may interrupt time-critical operations. If one cannot avoid dynamic resizing, a solution is to perform the resizing gradually:

- During the resize, allocate the new hash table, but keep the old table unchanged.
- In each lookup or delete operation, check both tables.
- Perform insertion operations only in the new table.
- At each insertion also move r elements from the old table to the new table.
- When all elements are removed from the old table, deallocate it.

To ensure that the old table is completely copied over before the new table itself needs to be enlarged, it is necessary to increase the size of the table by a factor of at least $(r + 1)/r$ during resizing.

Disk-based hash tables almost always use some scheme of incremental resizing, since the cost of rebuilding the entire table on disk would be too high.

Monotonic keys

If it is known that key values will always increase (or decrease) monotonically, then a variation of consistent hashing can be achieved by keeping a list of the single most recent key value at each hash table resize operation. Upon lookup, keys that fall in the ranges defined by these list entries are directed to the appropriate hash function—and indeed hash table—both of which can be different for each range. Since it is common to grow the overall number of entries by doubling, there will only be$O(\log(N))$ ranges to check, and binary search time for the redirection would be $O(\log(\log(N)))$. As with consistent hashing, this approach guarantees that any key's hash, once issued, will never change, even when the hash table is later grown.

Other solutions

Linear hashing is a hash table algorithm that permits incremental hash table expansion. It is implemented using a single hash table, but with two possible lookup functions.
Another way to decrease the cost of table resizing is to choose a hash function in such a way that the hashes of most values do not change when the table is resized. This approach, called consistent hashing, is prevalent in disk-based and distributed hash tables, where rehashing is prohibitively costly.

SEARCHING AND SORTING

INSERTION SORT

The basic step in this method is to insert a new record into a sorted sequence of *i* records in such a way that the resulting sequence of size *i* + 1 is also ordered.
Function *insert* accomplishes this insertion.

```
void insert(element e, element all, int i)
        {/* insert e into the ordered list a[1 : i] such that the resulting list a[1: i+1] is also
                ordered, the array a must have space allocated for at least i+2 elements */
                a[0] = e;
                while (e.key < a[i].key)
                {
                        a[i+1] = a[i] ;
                        i--;
                }
                a[i+1] = e;
        }
```

Program: Insertion into a sorted list

The use of *a*[0] enables us to simplify the while loop, avoiding a test for end of list (*i* < 1). In insertion sort, begin with the ordered sequence *a* [1] and successively insert the records *a* [2], *a* [3], ... , *a [n]*. Since each insertion leaves the resultant sequence ordered, the list with *n* records can be ordered making *n* - 1 insertions.
The details are given in function *insertionSort*.

```
void insertionSort(element all, int n)
        {       /* sort a[1: n] into nondecreasing order */
                int j;
                for (j = 2i j <= n : j++)
                {
                        element   temp   =   a[j];
                        insert (temp, a, j-1);
                }
        }
```

Program: Insertion sort

Analysis of insertion Sort: In the worst case insert (e, a, i) makes i + 1 comparisons before making the insertion. Hence the complexity of Insert is O(i). Function insertionSort invokes insert for i = j - 1 = 1, 2, ... , n - 1. So, the complexity of insertionSort is

$$O(\sum_{i=1}^{n-1} (i+1)) = O(n^2).$$

The average time for insertionSort is $O(n^2)$

Example: Assume that n = 5 and the input key sequence is 5, 4, 3, 2, 1. After each iteration we have

j	[1]	[2]	[3]	[4]	[5]
–	5	4	3	2	1
2	4	5	3	2	1
3	3	4	5	2	1
4	2	3	4	5	1
5	1	2	3	4	5

Example: Assume that n = 5 and the input key sequence is 2, 3, 4, 5, 1. after each iteration we have

j	[1]	[2]	[3]	[4]	[5]
–	2	3	4	5	1
2	2	3	4	5	1
3	2	3	4	5	1
4	2	3	4	5	1
5	1	2	3	4	5

RADIX SORT

Radix sort is the method that many people intuitively use or begin to use when alphabetizing a large list of names. (Here the radix is 26, the 26 letters of the alphabet.) Specifically, the list of names is first sorted according to the first letter of each name. That is, the names are arranged in 26 classes, where the first class consists of those names that begin with "A," the second class consists of those names that begin with "B," and so on. During the second pass, each class is alphabetized according to the second letter of the name. And so on. If no name contains, for example, more than 12 letters, the names are alphabetized with at most 12 passes.

The radix sort is the method used by a card sorter. A card sorter contains 13 receiving pockets labelled as follows:

9, 8, 7, 6, 5, 4, 3, 2, 1, 0, 11, 12, R (reject)

Each pocket other than R corresponds to a row on a card in which a hole can be punched. Decimal numbers, where the radix is 10, are punched in the obvious way and hence use only the first 10 pockets of the sorter. The sorter uses a radix reverse-digit sort on numbers. That is, suppose a card sorter is given a collection of cards where each card

contains a 3-digit number punched in columns 1 to 3. The cards are first sorted according to the unit's digit. On the second pass, the cards are sorted according to the tens digit. On the third and last pass, the cards are sorted according to the hundreds digit.

Illustration with an example:

Suppose 9 cards are punched as follows:
348, 143, 361, 423, 538, 128, 321, 543, 366

Given to a card sorter, the numbers would be sorted in three phases, as pictured in

Input	0	1	2	3	4	5	6	7	8	9
348									348	
143				143						
361		361								
423				423						
538									538	
128									128	
321		321								
543				543						
366							366			

(a) First pass

Input	0	1	2	3	4	5	6	7	8	9
361							361			
321			321							
143					143					
423			423							
543					543					
366					543					
366							366			
348					348					
538				538						
128			128							

(b) Second pass

Input	0	1	2	3	4	5	6	7	8	9
321				321						
423					423					
128		128								
538						538				
143		143								
543						543				
348				348						
361				361						
366				366						

(c) Third pass

Files and Their Organization

DATA HIERARCHY

Every file contains data which can be organized in a hierarchy to present a systematic organization.

The data hierarchy includes data items such as fields, records, files, and database. These terms are defined below.

- *Data field:* A *data field* is an elementary unit that stores a single fact. A data field is usually characterized by its type and size.
  Example: student's name is a data field that stores the name of students.

- *Record:* A *record* is a collection of related data fields which is seen as a single unit from the application point of view.
  Example: The student's record may contain data fields such as name, address, phone number, roll number, marks obtained, and so on.

- *File:* A *file* is a collection of related records.
  Example: A file of all the employees working in an organization

- *Directory*: A *directory* stores information of related files. A directory organizes information so that users can find it easily.
  Example: Below fig. shows how multiple related files are stored in a student directoy.

Student's
Personal Info
File

Roll_no
Name
Address
Phone No

Student's
Academic Info
File

Roll_no
Name
Course
Marks
Grade in Sports

Student's
Fees Info
File

Roll_no
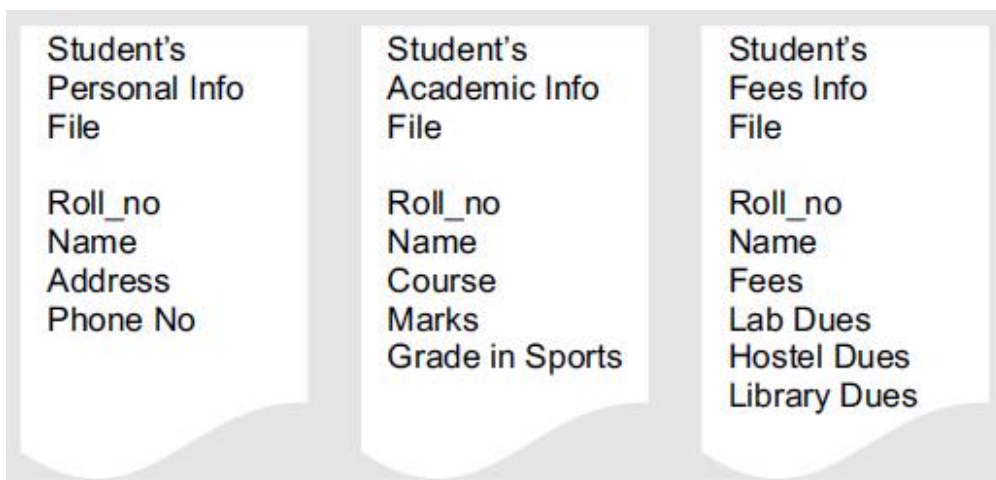Name
Fees
Lab Dues
Hostel Dues
Library Dues

Figure Student directory

FILE ATTRIBUTES

File has a list of attributes associated with it that gives the operating system and the application software information about the file and how it is intended to be used.

The attributes are explained below

- **File name:** It is a string of characters that stores the name of a file. File naming conventions vary from one operating system to the other.
- **File position:** It is a pointer that points to the position at which the next read/write operation will be performed.
- **File structure:** It indicates whether the file is a text file or a binary file. In the text file, the numbers are stored as a string of characters. A binary file stores numbers in the same way as they are represented in the main memory.
- **File Access Method:** It indicates whether the records in a file can be accessed sequentially or randomly.
  In sequential access mode, records are read one by one. That is, if 60 records of students are stored in the STUDENT file, then to read the record of 39th student, you have to go through the record of the first 38 students.
  In random access, records can be accessed in any order.

- **Attributes Flag:** A file can have six additional attributes attached to it. These attributes are usually stored in a single byte, with each bit representing a specific attribute. If a particular bit is set to '1' then this means that the corresponding attribute is turned on.

Table Attribute flag

Attribute	Attribute Byte
Read-Only	00000001
Hidden	00000010
System	00000100
Volume Label	00001000
Directory	00010000
Archive	00100000

Above figure shows the list of attributes and their position in the attribute flag or attribute byte.

- **Read-only:** A file marked as read-only cannot be deleted or modified. Example: if an attempt is made to either delete or modify a read-only file, then a message 'access denied' is displayed on the screen.
- **Hidden:** A file marked as hidden is not displayed in the directory listing.
- **System:** A file marked as a system file indicates that it is an important file used by the system and should not be altered or removed from the disk.
- **Volume Label:** Every disk volume is assigned a label for identification. The label can be assigned at the time of formatting the disk or later through various tools such as the DOS command LABEL.
- **Directory:** In directory listing, the files and sub-directories of the current directory are differentiated by a directory-bit. This means that the files that have the directory-bit turned on are actually sub-directories containing one or more files.
- **Archive:** The archive bit is used as a communication link between programs that modify files and those that are used for backing up files. Most backup programs allow the user to do an incremental backup.

TEXT AND BINARY FILES

Text Files
- A text file, also known as a flat file or an ASCII file, is structured as a sequence of lines of alphabet, numerals, special characters.
- The data in a text file, whether numeric or non-numeric, is stored using its corresponding ASCII code.
- The end of a text file is denoted by placing a special character, called an end-of-file marker, after the last line in the text file.
- It is possible for humans to read text files which contain only ASCII text.
- Text files can be manipulated by any text editor, they do not provide efficient storage.

Binary Files
- A binary file contains any type of data encoded in binary form for computer storage and processing purposes.
- A binary file can contain text that is not broken up into lines.
- A binary file stores data in a format that is similar to the format in which the data is stored in the main memory. Therefore, a binary file is not readable by humans.
- Binary files contain formatting information that only certain applications or processors can understand.
- Binary files must be run on an appropriate software or processor so that the software or processor can transform the data in order to make it readable.
- Binary files provide efficient storage of data, but they can be read only through an appropriate program.

BASIC FILE OPERATIONS

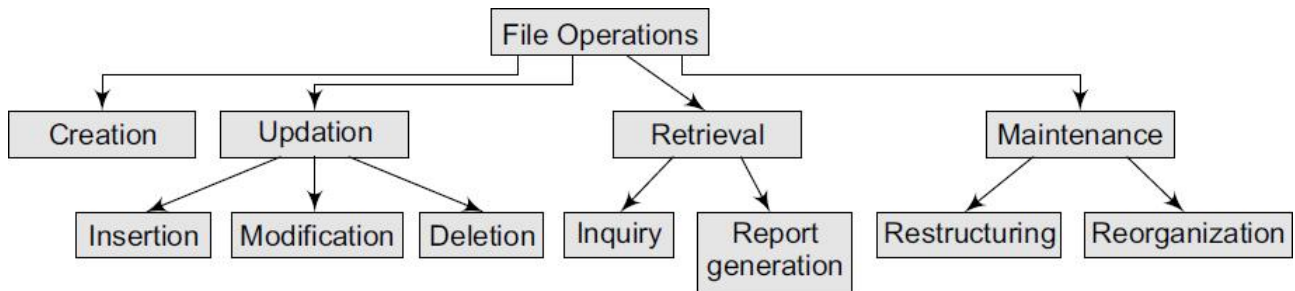The basic operations that can be performed on a file are given in below figure

Figure File operations

Creating a File

A file is created by specifying its name and mode. Then the file is opened for writing records that are read from an input device. Once all the records have been written into the file, the file is closed. The file is now available for future read/write operations by any program that has been designed to use it in some way or the other.

Updating a File

Updating a file means changing the contents of the file to reflect a current picture of reality.
A file can be updated in the following ways:
- Inserting a new record in the file. For example, if a new student joins the course, we need to add his record to the STUDENT file.
- Deleting an existing record. For example, if a student quits a course in the middle of the session, his record has to be deleted from the STUDENT file.
- Modifying an existing record. For example, if the name of a student was spelt incorrectly, then correcting the name will be a modification of the existingrecord.

Retrieving from a File

It means extracting useful data from a given file. Information can be retrieved from a file either for an inquiry or for report generation. An inquiry for some data retrieves low volume of data, while report generation may retrieve a large volume of data from the file.

Maintaining a File

It involves restructuring or re-organizing the file to improve the performance of the programs that access this file.
Restructuring a file keeps the file organization unchanged and changes only the structural aspects of the file.
Example: changing the field width or adding/deleting fields.

File reorganization may involve changing the entire organization of the file

FILE ORGANIZATION

Organization of records means the logical arrangement of records in the file and not the physical layout of the file as stored on a storage media.

The following considerations should be kept in mind before selecting an appropriate file organization method:
- Rapid access to one or more records
- Ease of inserting/updating/deleting one or more records without disrupting the speed of accessing record
- Efficient storage of records
- Using redundancy to ensure data integrity

1. Sequential Organization

A sequentially organized file stores the records in the order in which they were entered. Sequential files can be read only sequentially, starting with the first record in the file. Sequential file organization is the most basic way to organize a large collection of records in a file
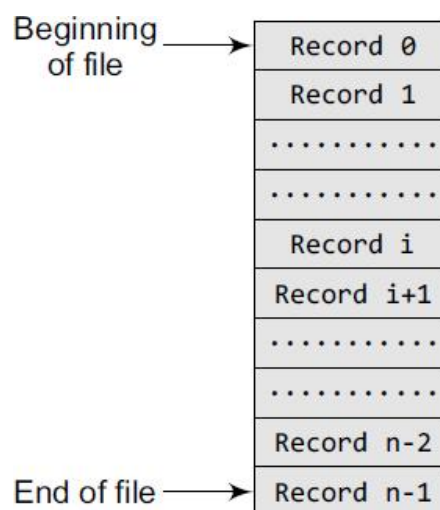

Figure Sequential file organization

Features
- Records are written in the order in which they are entered
- Records are read and written sequentially
- Deletion or updation of one or more records calls for replacing the original file with a new file that contains the desired changes
- Records have the same size and the same field format
- Records are sorted on a key value
- Generally used for report generation or sequential reading

Advantages
- Simple and easy to Handle
- No extra overheads involved
- Sequential files can be stored on magnetic disks as well as magnetic tapes
- Well suited for batch– oriented applications

Disadvantages
- Records can be read only sequentially. If ith record has to be read, then all the i–1 records must be read
- Does not support update operation. A new file has to be created and the original file has to be replaced with the new file that contains the desired changes
- Cannot be used for interactive applications

2. Relative File Organization

Figure shows a schematic representation of a relative file which has been allocated space to store 100 records

Relative record number	Records stored in memory
0	Record 0
1	Record 1
2	FREE
3	FREE
4	Record 4
....	....
98	FREE
99	Record 99

Figure Relative file organization

If the records are of fixed length and we know the base address of the file and the length of the record, then any record i can be accessed using the following formula:

$$\text{Address of i}^{th}\text{ record} = \text{base\_address} + (i–1) * \text{record\_length}$$

Consider the base address of a file is 1000 and each record occupies 20 bytes, then the address of the 5th record can be given as:

$$1000 + (5–1) * 20$$
$$= 1000 + 80$$
$$= 1080$$

Features

- Provides an effective way to access individual records
- The record number represents the location of the record relative to the beginning of the file
- Records in a relative file are of fixed length
- Relative files can be used for both random as well as sequential access
- Every location in the table either stores a record or is marked as FREE

Advantages

- Ease of processing
- If the relative record number of the record that has to be accessed is known, then the record can be accessed instantaneously
- Random access of records makes access to relative files fast
- Allows deletions and updations in the same file
- Provides random as well as sequential access of records with low overhead
- New records can be easily added in the free locations based on the relative record number of the record to be inserted
- Well suited for interactive applications

Disadvantages

- Use of relative files is restricted to disk devices
- Records can be of fixed length only
- For random access of records, the relative record number must be known in advance

3. Indexed Sequential File Organization

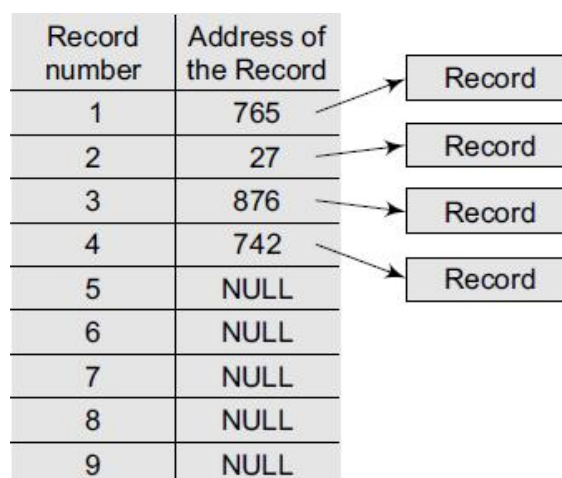The index sequential file organization can be visualized as shown in figure

Figure Indexed sequential file organization

Features
- Provides fast data retrieval
- Records are of fixed length
- Index table stores the address of the records in the file
- The ith entry in the index table points to the ith record of the file
- While the index table is read sequentially to find the address of the desired record, a direct access is made to the address of the specified record in order to access it randomly
- Indexed sequential files perform well in situations where sequential access as well as random access is made to the data

Advantages
- The key improvement is that the indices are small and can be searched quickly, allowing the database to access only the records it needs
- Supports applications that require both batch and interactive processing
- Records can be accessed sequentially as well as randomly
- Updates the records in the same file

Disadvantages
- Indexed sequential files can be stored only on disks
- Needs extra space and overhead to store indices
- Handling these files is more complicated than handling sequential files
- Supports only fixed length records

INDEXING

the indexing technique based on factors such as access type, access time, insertion time, deletion time, and space overhead involved. There are two kinds of indices:
- *Ordered indices* that are sorted based on one or more key values
- *Hash indices* that are based on the values generated by applying a *hash function*

1. Ordered Indices

Indices are used to provide fast random access to records. An index of a file may be a primary index or a secondary index.

Primary Index

In a sequentially ordered file, the index whose search key specifies the sequential order of the file is defined as the primary index.

Example: suppose records of students are stored in a STUDENT file in a sequential order starting from roll number 1 to roll number 60. Now, if we want to search a record for, say, roll number 10, then the student's roll number is the primary index.

Secondary Index

An index whose search key specifies an order different from the sequential order of the file is called as the secondary index.

Example: If the record of a student is searched by his name, then the name is a secondary index. Secondary indices are used to improve the performance of queries on non-primary keys.

2. **Dense and Sparse Indices**

Dense index

- In a dense index, the index table stores the address of every record in the file.
- Dense index would be more efficient to use than a sparse index if it fits in the memory
- By looking at the dense index, it can be concluded directly whether the record exists in the file or not.

Sparse index

- In a sparse index, the index table stores the address of only some of the records in the file.
- Sparse indices are easy to fit in the main memory,
- In a sparse index, to locate a record, first find an entry in the index table with the largest search key value that is either less than or equal to the search key value of the desired record. Then, start at that record pointed to by that entry in the index table and then proceed searching the record using the sequential pointers in the file, until the desired record is obtained.

Example: If we need to access record number 40, then record number 30 is the largest key value that is less than 40. So jump to the record pointed by record number 30 and move along the sequential pointer to reach record number 40.

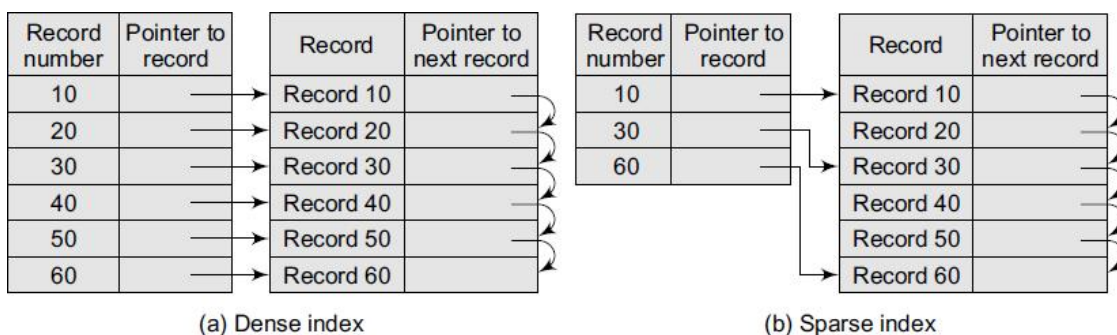Below figure shows a dense index and a sparse index for an indexed sequential file.

Record number	Pointer to record	Record	Pointer to next record		Record number	Pointer to record	Record	Pointer to next record
10		Record 10			10		Record 10	
20		Record 20			30		Record 20	
30		Record 30			60		Record 30	
40		Record 40					Record 40	
50		Record 50					Record 50	
60		Record 60					Record 60	

(a) Dense index   (b) Sparse index

Figure Dense index and sparse index

3. <u>Cylinder Surface Indexing</u>

Cylinder surface indexing is a very simple technique used only for the primary key index of a sequentially ordered file.
The index file will contain two fields—cylinder index and several surface indices.
There are multiple cylinders, and each cylinder has multiple surfaces. If the file needs m cylinders for storage then the cylinder index will contain m entries.
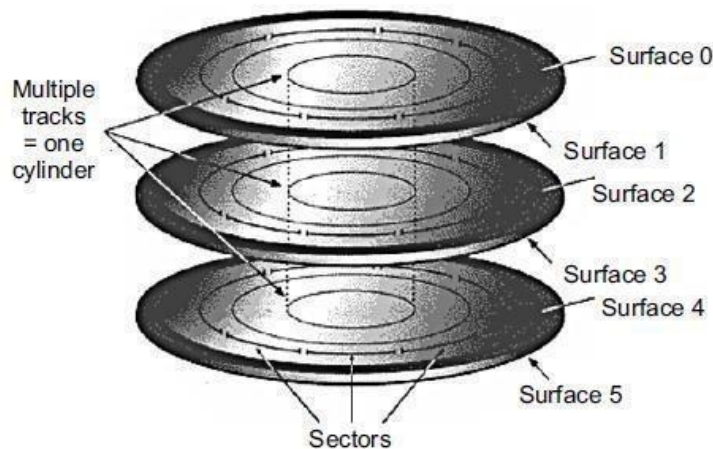

Figure Physical and logical organization of disk

When a record with a particular key value has to be searched, then the following steps are performed:

- First the cylinder index of the file is read into memory.
- Second, the cylinder index is searched to determine which cylinder holds the desired record. For this, either the binary search technique can be used or the cylinder index can be made to store an array of pointers to the starting of individual key values. In either case the search will take O (log m) time.
- After the cylinder index is searched, appropriate cylinder is determined.
- Depending on the cylinder, the surface index corresponding to the cylinder is then retrieved from the disk.
- Since the number of surfaces on a disk is very small, linear search can be used to determine surface index of the record.
- Once the cylinder and the surface are determined, the corresponding track is read and searched for the record with the desired key.

Hence, the total number of disk accesses is three—first, for accessing the cylinder index, second for accessing the surface index, and third for getting the track address.

4. Multi-level Indices

Consider very large files that may contain millions of records. For such files, a simple indexing technique will not suffice. In such a situation, we use multi-level indices.

Below figure shows a two-level multi-indexing. Three-level indexing and so, can also be used

In the figure, the main index table stores pointers to three inner index tables. The inner index tables are sparse index tables that in turn store pointers to the records.
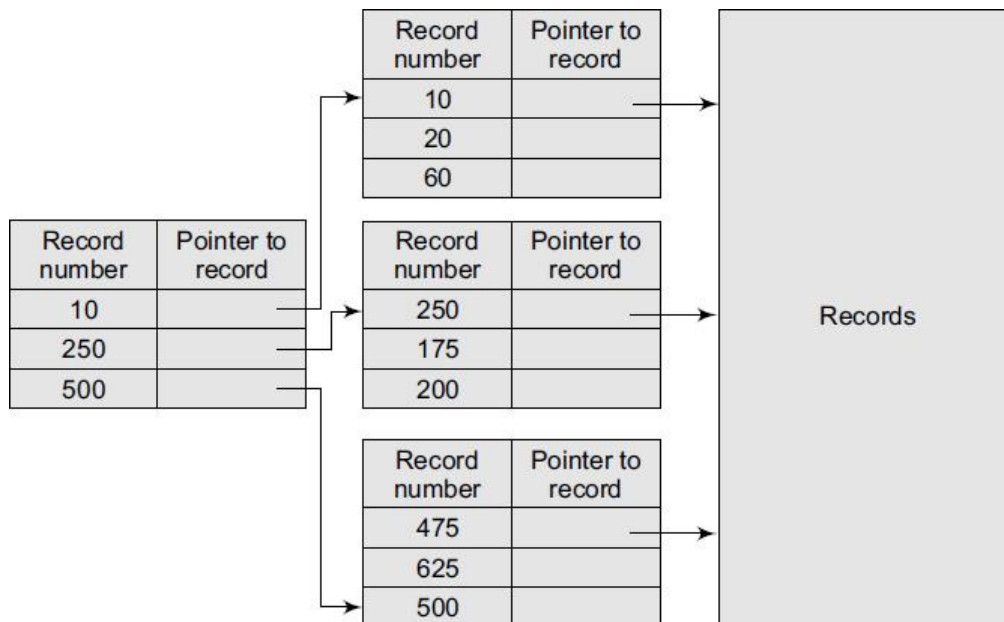

Figure Multi-level indices

5. Inverted Indices

- Inverted files are used in document retrieval systems for large textual databases.
- An inverted file reorganizes the structure of an existing data file in order to provide fast access to all records having one field falling within the set limits.
- When a term or keyword specified in the inverted file is identified, the record number is given and a set of records corresponding to the search criteria are created.
- For each keyword, an inverted file contains an inverted list that stores a list of pointers to all occurrences of that term in the main text. Therefore, given a keyword, the addresses of all the documents containing that keyword can easily be located.

There are two main variants of inverted indices:

- A record-level inverted index (inverted file index or inverted file) stores a list of references to documents for each word
- A word-level inverted index (full inverted index or inverted list) in addition to a list of references to documents for each word also contains the positions of each word within a document.

6. <u>B-Tree (Balanced Tree) Indices</u>

It is impractical to maintain the entire database in the memory, hence B-trees are used to index the data in order to provide fast access.
B-trees are used for its data retrieval speed, ease of maintenance, and simplicity.
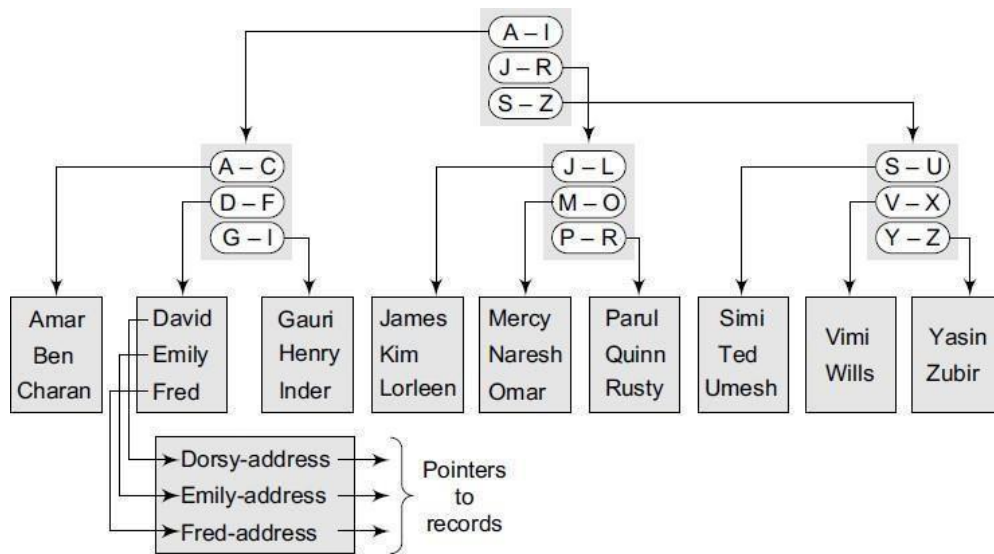

Figure B-tree index

- It forms a tree structure with the root at the top. The index consists of a B-tree (balanced tree) structure based on the values of the indexed column.
- In this example, the indexed column is name and the B-tree is created using all the existing names that are the values of the indexed column.
- The upper blocks of the tree contain index data pointing to the next lower block, thus forming a hierarchical structure. The lowest level blocks, also known as leaf blocks, contain pointers to the data rows stored in the table.

The B-tree structure has the following advantages:
- Since the leaf nodes of a B-tree are at the same depth, retrieval of any record from anywhere in the index takes approximately the same time.
- B-trees improve the performance of a wide range of queries that either searches a value having an exact match or for a value within specified range.
- B-trees provide fast and efficient algorithms to insert, update, and delete records that maintain the key order.
- B-trees perform well for small as well as large tables. Their performance does not degrade as the size of a table grows.
- B-trees optimize costly disk access.

7. Hashed Indices

Hashing is used to compute the address of a record by using a hash function on the search key value.

The hashed values map to the same address, then collision occurs and schemes to resolve these collisions are applied to generate a new address

Choosing a **good hash function** is critical to the success of this technique. By a good hash function, it mean two things.

1. First, a good hash function, irrespective of the number of search keys, gives an average-case lookup that is a small constant.
2. Second, the function distributes records uniformly and randomly among the buckets, where a bucket is defined as a unit of one or more records

The **worst hash function** is one that maps all the keys to the same bucket.

The drawback of using hashed indices includes:

- Though the number of buckets is fixed, the number of files may grow with time.
- If the number of buckets is too large, storage space is wasted.
- If the number of buckets is too small, there may be too many collisions.

The following operations are performed in a hashed file organization.

1. Insertion

To insert a record that has k_i as its search value, use the hash function $h(k_i)$ to compute the address of the bucket for that record.

If the bucket is free, store the record else use chaining to store the record.

2. Search

To search a record having the key value k_i, use $h(k_i)$ to compute the address of the bucket where the record is stored.

The bucket may contain one or several records, so check for every record in the bucket to retrieve the desired record with the given key value.

3. Deletion

To delete a record with key value k_i, use $h(k_i)$ to compute the address of the bucket where the record is stored. The bucket may contain one or several records so check for every record in the bucket, and then delete the record.