CANARA ENGINEERING COLLEGE

Benjanapadavu, Bantwal Taluk - 574219

Department of Computer Science & Engineering

VISION

To be recognized as a center of knowledge dissemination in Computer Science and Engineering by imparting value-added education to transform budding minds into competent computer professionals.

MISSION

M1. Provide a learning environment enriched with ethics that helps in enhancing problem solving skills of students and, cater to the needs of the society and industry.

M2. Expose the students to cutting-edge technologies and state-of-the-art tools in the many areas of Computer Science & Engineering.

M3. Create opportunities for all round development of students through co-curricular and extra-curricular activities.

M4. Promote research, innovation and development activities among staff and students.

PROGRAMME EDUCATIONAL OBJECTIVES

PE01: Graduates will work productively as computer science engineers exhibiting ethical qualities and leadership roles in multidisciplinary teams.

PEO2: Graduates will adapt to the changing technologies, tools and societal requirements.

PEO3: Graduates will design and deploy software that meets the needs of individuals and the industries

PEO4: Graduates will take up higher education and/or be associated with the field so that they can keep themselves abreast of Research & Development

PROGRAMME OUTCOMES

Engineering graduates in Computer Science and Engineering will be able to:

1. **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis:** Identify, formulate, review research literature and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences and engineering sciences.
3. **Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specific needs with appropriate consideration for the public health and safety, and the cultural, societal and environmental considerations.
4. **Conduct investigations of complex problems:** Use research-based knowledge and research methods, including design of experiments, analysis and interpretation of data and synthesis of the information to provide valid conclusions.
5. **Modern tool usage:** Select/Create and apply appropriate techniques, resources and modern engineering and IT tools, including prediction and modeling to complex engineering activities, taking comprehensive cognizance of their limitations.
6. **The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and Sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts and demonstrate the knowledge of and need for sustainable development.
8. **Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the relevant scientific and/or engineering practices.
9. **Individual and team work:** Function effectively as an individual and as a member or leader in diverse teams and in multidisciplinary settings.
10. **Communication:** Communicate effectively on complex engineering activities with the engineering community and with the society-at-large, such as being able to comprehend and write effective reports and design documentation, make effective presentations and give and receive clear instructions.
11. **Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one’s own work as a member and leader in a team to manage projects and in multidisciplinary environments.

12. **Life-long learning:** Recognize the need for and above have the preparation and ability to engage in independent and life-long learning in the broadcast context of technological changes.

PROGRAMME SPECIFIC OUTCOMES

1. **Computer System Components:** Apply the principles of computer system architecture and software to design, develop and deploy computer subsystem.
2. **Data Driven and Internet Applications:** Apply the knowledge of data storage, analytics and network architecture in designing Internet based applications.

DATA STRUCTURES AND APPLICATIONS **(Effective from the academic year 2018 -2019)** **SEMESTER – III**			
Course Code	**18CS32**	**CIE Marks**	40
Number of Contact Hours/Week	3:2:0	**SEE Marks**	60
Total Number of Contact Hours	50	**Exam Hours**	03
CREDITS –4			
Course Learning Objectives: This course (18CS32) will enable students to:			
• Explain fundamentals of data structures and their applications essential for programming/problem solving. • Illustrate linear representation of data structures: Stack, Queues, Lists, Trees and Graphs. • Demonstrate sorting and searching algorithms. • Find suitable data structure during application development/Problem Solving.			

Module 1	**Contact Hours**
Introduction: Data Structures, Classifications (Primitive & Non-Primitive), Data structure Operations, Review of Arrays, Structures, Self-Referential Structures, and Unions. Pointers and Dynamic Memory Allocation Functions. Representation of Linear Arrays in Memory, Dynamically allocated arrays. **Array Operations**: Traversing, inserting, deleting, searching, and sorting. Multidimensional Arrays, Polynomials and Sparse Matrices. **Strings:** Basic Terminology, Storing, Operations and Pattern Matching algorithms. Programming Examples. **Textbook 1: Chapter 1: 1.2, Chapter 2: 2.2 - 2.7 Text Textbook 2: Chapter 1: 1.1 - 1.4,** **Chapter 3: 3.1 - 3.3, 3.5, 3.7, Chapter 4: 4.1 - 4.9, 4.14 Reference 3: Chapter 1: 1.4 RBT: L1, L2, L3**	10

Module 2	
Stacks: Definition, Stack Operations, Array Representation of Stacks, Stacks using Dynamic Arrays, Stack Applications: Polish notation, Infix to postfix conversion, evaluation of postfix expression. **Recursion** - Factorial, GCD, Fibonacci Sequence, Tower of Hanoi, Ackerman's function. **Queues:** Definition, Array Representation, Queue Operations, Circular Queues, Circular queues using Dynamic arrays, Dequeues, Priority Queues, A Mazing Problem. Multiple Stacks and Queues. Programming Examples. **Textbook 1: Chapter 3: 3.1 -3.7 Textbook 2: Chapter 6: 6.1 -6.3, 6.5, 6.7-6.10, 6.12, 6.13** **RBT: L1, L2, L3**	10
Module 3	
Linked Lists: Definition, Representation of linked lists in Memory, Memory allocation; Garbage Collection. Linked list operations: Traversing, Searching, Insertion, and Deletion. Doubly Linked lists, Circular linked lists, and header linked lists. Linked Stacks and Queues. Applications of Linked lists – Polynomials, Sparse matrix representation. Programming Examples **Textbook 1: Chapter 4: 4.1 – 4.6, 4.8, Textbook 2: Chapter 5: 5.1 – 5.10,** **RBT: L1, L2, L3**	10
Module 4	
Trees: Terminology, Binary Trees, Properties of Binary trees, Array and linked Representation of Binary Trees, Binary Tree Traversals - Inorder, postorder, preorder; Additional Binary tree operations. Threaded binary trees, Binary Search Trees – Definition, Insertion, Deletion, Traversal, Searching, Application of Trees-Evaluation of Expression, Programming Examples **Textbook 1: Chapter 5: 5.1 –5.5, 5.7; Textbook 2: Chapter 7: 7.1 – 7.9** **RBT: L1, L2, L3**	10
Module 5	
Graphs: Definitions, Terminologies, Matrix and Adjacency List Representation of Graphs, Elementary Graph operations, Traversal methods: Breadth First Search and Depth First Search. **Sorting and Searching**: Insertion Sort, Radix sort, Address Calculation Sort. **Hashing:** Hash Table organizations, Hashing Functions, Static and Dynamic Hashing. **Files and Their Organization:** Data Hierarchy, File Attributes, Text Files and Binary Files, Basic File Operations, File Organizations and Indexing **Textbook 1: Chapter 6: 6.1 –6.2, Chapter 7:7.2, Chapter 8: 8.1-8.3** **Textbook 2: Chapter 8: 8.1 – 8.7, Chapter 9: 9.1-9.3, 9.7, 9.9** **Reference 2: Chapter 16: 16.1 - 16.7** **RBT: L1, L2, L3**	10
Course Outcomes: The student will be able to:	

- Use different types of data structures, operations and algorithms
- Apply searching and sorting operations on files
- Use stack, Queue, Lists, Trees and Graphs in problem solving
- Implement all data structures in a high-level language for problem solving.

Question Paper Pattern:

- The question paper will have ten questions.
- Each full Question consisting of 20 marks
- There will be 2 full questions (with a maximum of four sub questions) from each module.
- Each full question will have sub questions covering all the topics under a module.
- The students will have to answer 5 full questions, selecting one full question from each module.

Textbooks:

1. Ellis Horowitz and Sartaj Sahni, Fundamentals of Data Structures in C, 2nd Ed, Universities Press, 2014.
2. Seymour Lipschutz, Data Structures Schaum's Outlines, Revised 1st Ed, McGraw Hill, 2014.

Reference Books:

1. Gilberg & Forouzan, Data Structures: A Pseudo-code approach with C, 2nd Ed, Cengage Learning,2014.
2. Reema Thareja, Data Structures using C, 3rd Ed, Oxford press, 2012.
3. Jean-Paul Tremblay & Paul G. Sorenson, An Introduction to Data Structures with Applications, 2nd Ed, McGraw Hill, 2013
4. A M Tenenbaum, Data Structures using C, PHI, 1989
5. Robert Kruse, Data Structures and Program Design in C, 2nd Ed, PHI, 1996.

COURSE OBJECTIVES:

This course will enable students to	
1	Explain fundamentals of data structures and their applications essential for programming/problem solving.
2	Analyze Linear Data Structures: Stack, Queues, Lists
3	Analyze Non-Linear Data Structures: Trees, Graphs
4	Analyze and Evaluate the sorting & searching algorithms
5	Assess appropriate data structure during program development/Problem Solving

COURSE OUTCOMES (COs):

SL. NO	**DESCRIPTION**
	The students are able to:
CO:1	Explain various types of data structures, sorting and searching operations on arrays.
CO:2	Develop the programs on operations like searching, insertion, deletion, traversing mechanism on stack and queues.
CO:3	Apply the basic knowledge of linked list to solve real world problems.
CO:4	Develop the programs on operations like searching, insertion, deletion, traversing mechanism on trees.
CO:5	Explain the basic graph algorithms and their analyses. Employ graphs and Sorting and searching operations, to model engineering problems, when appropriate

MODULE-I

TABLE OF CONTENTS

MODULE-I

Introduction to Data Structures

Data Structure is a way of collecting and organizing data in such a way that we can perform operations on these data in an effective way. Data Structures is about rendering data elements in terms of some relationship, for better organization and storage. For example, we have data player's name "Virat" and age 26. Here "Virat" is of **String** data type and 26 is of **integer** data type.

We can organize this data as a record like **Player** record. Now we can collect and store player's records in a file or database as a data structure. For example: "Dhoni" 30, "Gambhir" 31, "Sehwag" 33.

- In simple language, Data Structures are structures programmed to store ordered data, so that various operations can be performed on it easily.
- It represents the knowledge of data to be organized in memory. It should be designed and implemented in such a way that it reduces the complexity and increases the efficiency.
- Data may be organized in many different ways. The logical or mathematical model of a particular organization of data is called a **data structure.**

Classification of data structures (Primitive & Non Primitive)

As we have discussed above, anything that can store data can be called as a data structure.

Data structures are generally categorized into two classes:

- Primitive data Structures
- Non-primitive data Structures

Primitive Data Structures

Primitive data structures are the fundamental data types which are supported by a programming language. Some basic data types are integer, real, character, and Boolean. The terms ‘data type’, ‘basic data type’, and ‘primitive data type’ are often used interchangeably.

Non-Primitive Data Structures

Non-primitive data structures are those data structures which are created using primitive data structures. Examples of such data structures include linked lists, stacks, trees, and graphs.

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

- Linear Data Structure

- Non-linear Data Structure

Linear data Structures

If the elements of a data structure are stored in a linear or sequential order, then it is a linear data structure. Examples include arrays, linked lists, stacks, and queues.

Linear data structures can be represented in memory in two different ways.

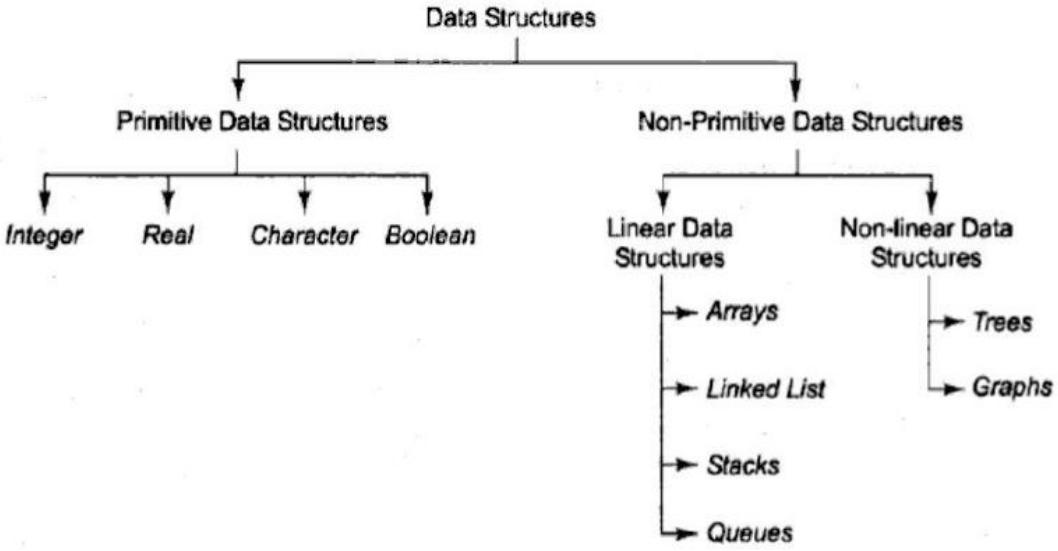
- One way is to have to a linear relationship between elements by means of sequential memory locations.
- The other way is to have a linear relationship between elements by means of links.

Non-linear data Structures

If the elements of a data structure are not stored in a sequential order, then it is a non-linear data structure.

- The relationship of adjacency is not maintained between elements of a non-linear data structure.
- This structure is mainly used to represent data containing a hierarchical relationship between elements.

  Examples include trees and graphs.



.

Figure (a). Classifications of Data Structures

Data structure Operations

Data are processed by means of certain operations which appearing in the data structure. In fact, particular data structure that one chooses for a given situation depends largely on the frequency with which specific operations are performed. This section introduces the reader to some of the most frequently used of these operations.

(1) ***Traversing:*** Accessing each record 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 record with a given key value, or finding the location of all records which satisfy one or more conditions.

(3) ***Inserting:*** Adding a new record to the structure.

(4) ***Deleting:*** Removing the record from the structure.

(5) ***Sorting:*** Arranging the data or record in some logical order (Ascending or descending order).

(6) ***Merging:*** Combining the record in two different sorted files into a single sorted file.

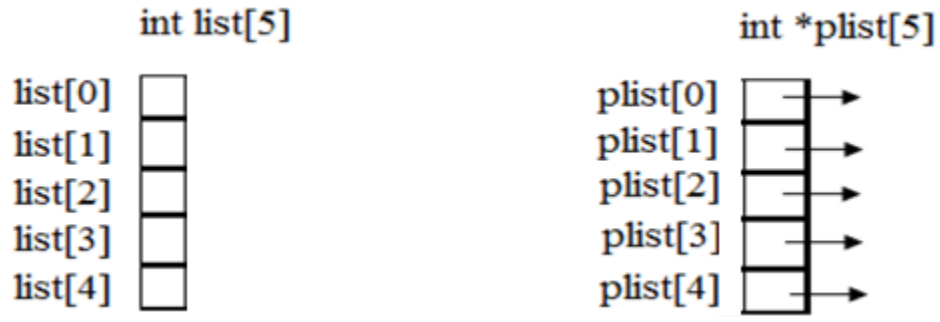
Review of Arrays

An **array data structure**, or simply an **array**, is a **data structure** consisting of a collection of (mainly of similar data types) elements (values or variables), each identified by at least one **array** index or key. An **array** is stored so that the position of each element can be computed from its index tuple by a mathematical formula.

ARRAYS IN C

- A one-dimensional array can be declared as follows:

```
int list[5];      //array of 5 integers
int *plist[5];    //array of 5 pointers to integers
```

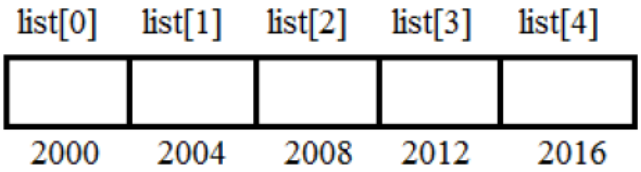


- Compiler allocates 5 consecutive memory-locations for each of the variables 'list' and 'plist'.
- Address of first element list [0] is called base-address.

- Memory-address of list[i] can be computed by compiler as

α + i * sizeof (int) where α =base address

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



Program to find sum of n numbers using array

```
#define MAX_SIZE 100
float sum (float [], int);

float input [MAX_SIZE], answer;
int i;

 main ()
{

        for (i = 0; i < MAX_SIZE; i++)
        input[i] = i;
        answer = sum (input, MAX_SIZE);
        printf("The sum is: %f\n", answer);

}

float sum (float list [], int n)

{
        int i;

        float tempsum = 0;
        for (i = 0; i < n; i++)

            tempsum += list[i];
            return tempsum;
}
```

Program to print both address of ith element of given array & the value found at that address.

```
void print1(int *ptr, int rows)
{

        /* print out a one-dimensional array using a pointer */
         int i;

        printf("Address Contents\n");
        for (i=0; i < rows; i++)

                printf("%8u%5d\n", ptr+i, *(ptr+i));
                printf("\n");

}

void main ()
{

    int one [] = {0, 1, 2, 3, 4};
    print1(&one [0], 5);

}
```

Output

Address	Contents
1228	0
1230	1
1232	2
1234	3
1236	4

Arrays are generally used when we want to store large amount of similar type of data. But they have the following limitations:

- Arrays are of fixed size.
- Data elements are stored in contiguous memory locations which may not be always available.
- Insertion and deletion of elements can be problematic because of shifting of elements from their positions.

STRUCTURES

In C,a way to group data that permits the data to vary in type. This mechanism is called the **structure**, for short **struct**.

A structure (a record) is a collection of data items, where each item is identified as to its type and name.

Syntax:

struct

{

data_type member 1;

data_type member 2;

……

……

data_type member n;

} variable_name;

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 character array

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;

} TypeName;

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 declared 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;
} human Being;
```

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;
}
```

2. Assignment operation on Structure variables:

person1 = person2

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

Valid Statements is given below:

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

Structure within a structure:

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

1. The structures are defined separately and a variable of structure type is declared inside the definition of another structure. The accessing of the variable of a structure type that are nested inside another structure in the same way as accessing other 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.
- These require dynamic storage management routines (malloc & free) to explicitly obtain and release memory.

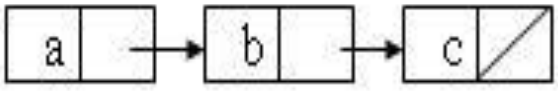
```
typedef struct
{
    char data;
    struct list *link;      //list is a pointer to a list structure
 } list;
```

- Consider three structures and values assigned to their respective fields:

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

- We can attach these structures together as follows

```
item1.link=&item2;
tem2.link=&item3;
```



INTERNAL IMPLEMENTATION OF STRUCTURES

- The size of an object of a struct or union type is the amount of storage necessary to represent the largest component, including any padding that may be required.
- Structures must begin and end on the same type of memory boundary. For ex, an even byte boundary (2, 4, 6 or 8).

Union

A **union** is a special data type available in C that allows us to store different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multiple-purpose.

```
#include <stdio.h>
#include <string.h>
 union Data
{
   int i;
   float f;
   char str [20];
};
 int main ()
{
   union Data data;
   printf (“Memory size occupied by data: %d\n", sizeof(data));
   return 0;
}
```

The memory occupied by a union will be large enough to hold the largest member of the union. For example, in the above example, Data type will occupy 20 bytes of memory space because this is the maximum space which can be occupied by a character string.

POINTERS

- In computer science, a pointer is a programming language data type whose value refers directly to (or "points to") another value stored elsewhere in the computer memory using its address.
- This is a memory-location which holds the address of another memory-location.
- The 2 most important operators used w.r.t pointer are:

  & (address operator)

  * (dereferencing/indirection operator)

```
#include<stdio.h>
void main ()
{
        int a=10, b=20;              //Declare a data variable
        int *p, *q;                  //Declare a pointer variable
        int p=&a, q=&b;              //Initialize a pointer variable
        int x=*p + *q;
        printf("%d+%d=%d",*p,*q, x);   //Access data using pointer variable
}
```

NULL POINTER

- The null pointer points to no object or function.

  i.e. it does not point to any part of the memory.

```
if(p==NULL)
    printf("p does not point to any memory");
else
    printf("access the value of p");
```

DYNAMIC MEMORY ALLOCATION

- This is process of allocating memory-space during execution-time (or run-time).
- This is used if there is an unpredictable storage requirement.
- Memory-allocation is done on a heap.
- Memory management functions include:
  - malloc (memory allocate)
  - calloc (contiguous memory allocate)
  - realloc (resize memory)
  - free (de-allocate memory)
- ***malloc*** function is used to allocate required amount of memory-space during run-time.
- If memory allocation succeeds, then address of first byte of allocated space is returned. If memory allocation fails, then NULL is returned.
- ***free ()*** function is used to de-allocate (or free) an area of memory previously allocated by malloc () or calloc().

```
#include<stdio.h>
void main ()
{
        int i, *pi;
        pi=(int*) malloc(sizeof(int));
        *pi=1024;
        printf("an integer =%d",pi);
        free(pi);
}
```

- If we frequently allocate the memory space, then it is better to define a macro as shown below:

```
#define MALLOC(p,s)          \
if (! ((p)==malloc(s))) \
{          \
          printf("insufficient memory");          \
          exit (0);          \
}
```

- Now memory can be initialized using following:

MALLOC (pi,sizeof(int));

MALLOC (pf,sizeof(float));

DANGLING REFERENCE

- Whenever all pointers to a dynamically allocated area of storage are lost, the storage is lost to the program. This is called a *dangling reference*.

POINTERS CAN BE DANGEROUS

1) Set all pointers to NULL when they are not actually pointing to an object. This makes sure that you will not attempt to access an area of memory that is either

→out of range of your program or

→that does not contain a pointer reference to a legitimate object.

2) Use explicit type casts when converting between pointer types.

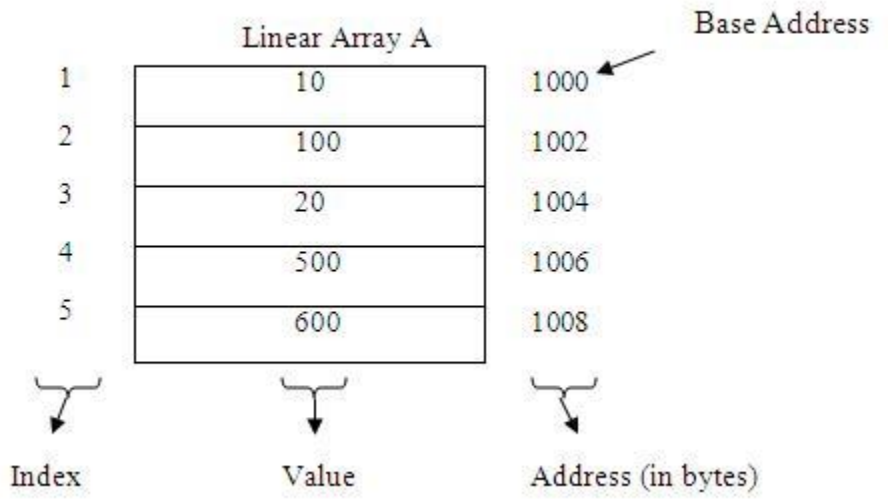
```
pi=malloc(sizeof(int));//assign to pi a pointer to int

pf=(float*) pi;          //casts an 'int' pointer to a 'float' pointer
```

3) Pointers have same size as data type 'int'. Since int is the default type specifier, some programmers omit return type when defining a function. The return type defaults to 'int' which can later be interpreted as a pointer. Therefore, programmer has to define explicit return types for functions.

```
void swap (int *p, int *q) //both parameters are pointers to ints
{
        int temp=*p; //declares temp as an int and assigns to it the contents of what p points to *p=*q;
        //stores what q points to into the location where p points
        *q=temp;     //places the contents temp in location pointed to by q

}
```

Representation of Linear Arrays in Memory

The elements of linear array are stored in consecutive memory locations. It is shown below:



DYNAMICALLY ALLOCATED ARRAYS

ONE-DIMENSIONAL ARRAYS

- When writing programs, sometimes we cannot reliably determine how large an array must be.
- A good solution to this problem is to
  - defer this decision to run-time &
  - Allocate the array when we have a good estimate of required array-size.

Dynamic memory allocation can be performed as follows:

```
int i,n,*list;
printf("enter the number of numbers to generate");
scanf("%d",&n);
if(n<1)
{
printf("improper value");
exit(0);
}
MALLOC (list, n*sizeof(int));
```

• The above code would allocate an array of exactly the required size and hence would not result in any wastage.

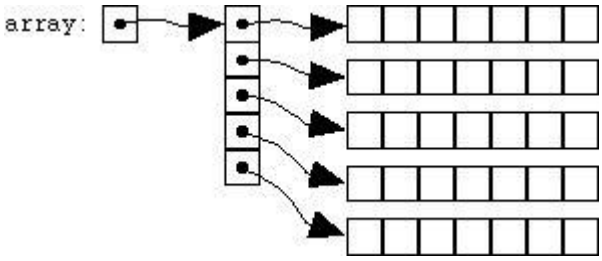
TWO DIMENSIONAL ARRAYS

• These are created by using the concept of array of arrays.

• A 2-dimensional array is represented as a 1-dimensional array in which each element has a pointer to a 1-dimensional array as shown below

```
int x [5][7];     //we create a 1-dimensional array x whose length is 5;
                  //each element of x is a 1-dimensional array whose length is 7.
```

• Address of x[i][j] = x[i]+j*sizeof(int)



Array-of-arrays representation

Dynamically create a two-dimensional array

```
Int ** make2darry (int rows, int cols)
{
    Int **x,i;
    /* get memory for new pointer*/
    MALLOC (x, rows * sizeof (*x));

    /* get memory for each row*/
    for (i=0; i<rows; i++)
    MALLOC(x[i], cols*sizeof (**x));
return x;
}
```

CALLOC

• These functions

→ allocate user-specified amount of memory &

→ Initialize the allocated memory to 0.

• On successful memory-allocation, it returns a pointer to the start of the new block.
On failure, it returns the value NULL.

• Memory can be allocated using calloc as shown below:

```
int *p;
p=(int*) calloc (n, sizeof(int)); //where n=array size
```

• To create clean and readable programs, a CALLOC macro can be created as shown below:

REALLOC

```
#define CALLOC(p,n,s)                \
if((p=calloc(n,s))==NULL)            \
{                                    \
printf("insufficient memory");       \
exit (0);                                  \
}
```

• These functions resize memory previously allocated by either malloc or calloc.
For example,

realloc(p,s); *//this changes the size of memory-block pointed at by p to s.*

• When s>oldSize, the additional s-oldSize have an unspecified value and
when s<oldSize, the rightmost oldSize-s bytes of old block are freed.

• On successful resizing, it returns a pointer to the start of the new block. On failure, it returns the value NULL.

• To create clean and readable programs, the REALLOC macro can be created as shown below

```
#define REALLOC(p,s)                 \
if((p=realloc(p,s))==NULL)           \
{                                    \
printf("insufficient memory");       \
exit (0);                                  \
}
```

ARRAY OPERATIONS

1. Traversing

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

Algorithm 1: (Traversing a Linear Array)

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

1. [Initialize Counter] set K:= LB

2. Repeat step 3 and 4 while K ≤ UB

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

4. [Increase counter] Set K:= K + 1

[End of step 2 loop]

5. Exit

Algorithm 2: (Traversing a Linear Array)

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

1. Repeat for K = LB to UB

    Apply PROCESS to LA [K]

[End of loop]

2. Exit.

Example:

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

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

1. [Initialization step.] Set NUM := 0

2. Repeat for K = 1932 to 1984:

    If AUTO [K] > 300, then: Set NUM: = NUM + 1.

[End of loop.]

3. Return.

2. Inserting

- Let A be a collection of data elements stored in the memory of the computer.

Inserting refers to the operation of adding another element to the collection A.

- Inserting an element at the "end" of the linear array can be easily done provided the memory space allocated for the array is large enough to accommodate the additional element.

- Inserting an element in the middle of the array, then on average, half of the elements must be moved downwards to new locations to accommodate the new element and keep the order of the other elements.

Algorithm:

INSERT (LA, N, K, ITEM)

Here LA is a linear array with N elements and K is a positive integer such that K ≤ 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 ≥ K
3. [Move Jth 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 ≤ 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]

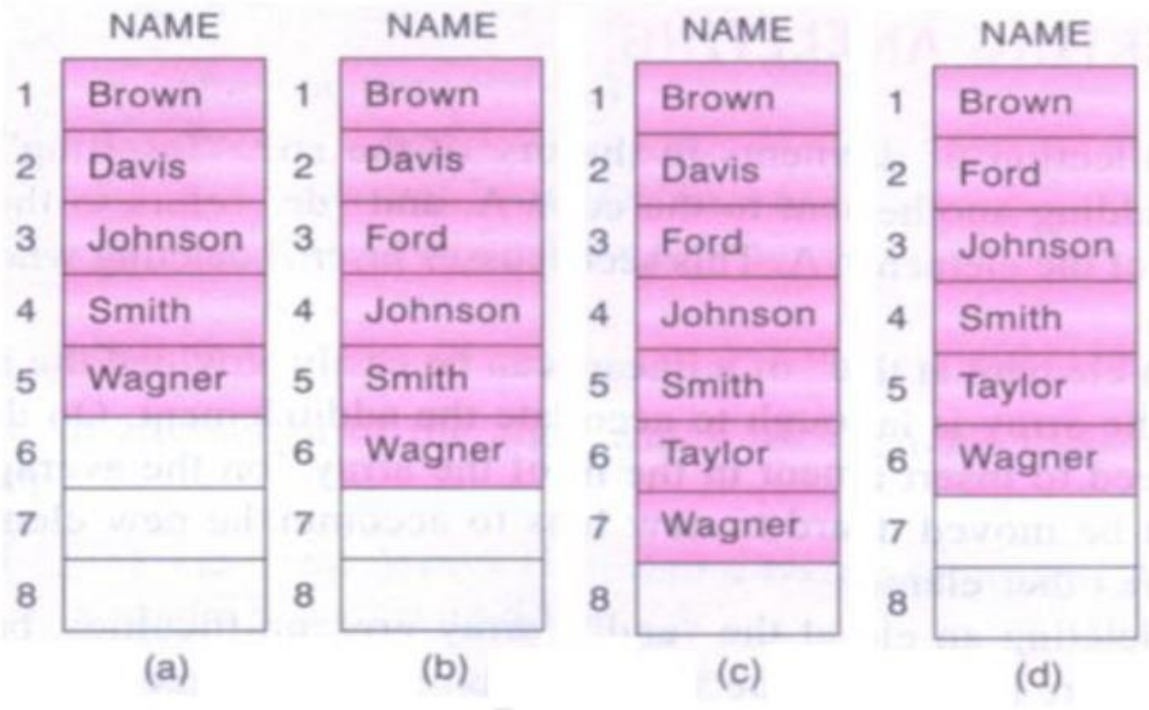
[End of loop]

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).



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

$$f(n) = (n - 1) + (n - 2) + \ldots + 2 + 1 = n(n-1)/2 = n^2/2 = O(n) = O(n^2)$$

5. Searching

- Let DATA be a collection of data elements in memory, and suppose a specific ITEM of information is given. ***Searching*** refers to the operation of finding the location LOC of ITEM in DATA, or printing some message that ITEM does not appear there.
- 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.

$$f(n) = (n+1)/2$$

Binary Search

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

Algorithm: (Binary Search) BINARY (DATA, LB, UB, ITEM, LOC)

Here DATA is a sorted array with lower bound LB and upper bound UB, and ITEM is a given item of information. The variables BEG, END and MID denote, the beginning, end and middle locations of a segment of elements of DATA.

This algorithm finds the location LOC of ITEM in DATA or sets LOC = NULL.

1. [Initialize segment variables.]

   Set BEG: = LB, END: = UB and MID = INT ((BEG + END)/2).

2. Repeat Steps 3 and 4 while BEG ≤ END and DATA [MID] ≠ ITEM.

3. If ITEM < DATA [MID], then:

   Set END: = MID - 1.

   Else:

   Set BEG: = MID + 1.

   [End of If structure.]

4. Set MID: = INT ((BEG + END)/2).

[End of Step 2 loop.]

5. If DATA[MID] = ITEM, then:

Set LOC: = MID.

Else:

Set LOC: = NULL.

[End of If structure.]

6. Exit.

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

$$f(n) = [\log_2 n] + 1$$

That is, the running time for the worst case is approximately equal to log2 *n.* One can also show that the running time for the average case is approximately equal to the running time for the worst case.

MULTIDIMENSIONAL ARRAY

Two-Dimensional Arrays

A two-dimensional m x n array A is a collection of *m x n* data elements such that each element is specified by a pair of integers (such as J, K), called subscripts, with the property that

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

The element of 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.*

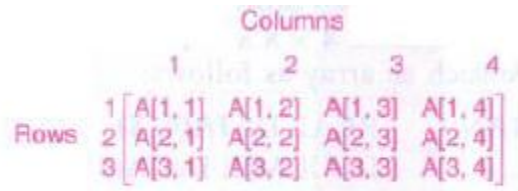
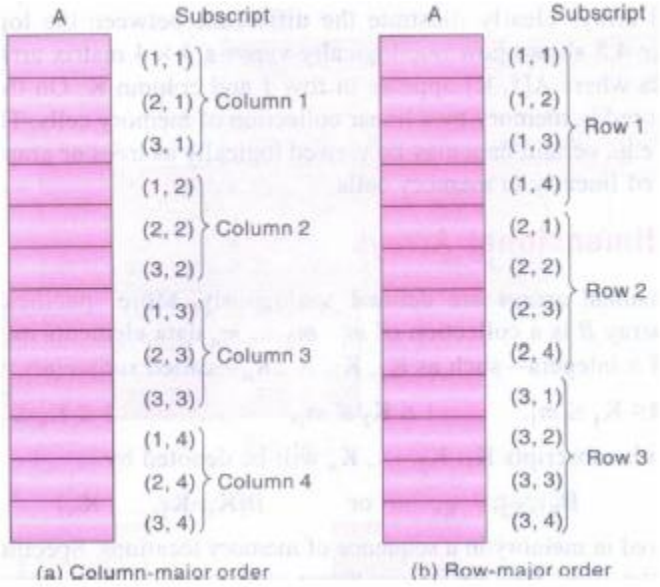


Fig. Two-Dimensional 3 × 4 Array A

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*.



Polynomials

- A polynomial is a sum of terms, where each term has a form ax^e,
  Where x=variable, a=coefficient and e=exponent.

  For ex,

  $A(x)=3x^{20}+2x^5+4$ and $B(x)=x^4+10x^3+3x^2+1$
- The largest (or leading) exponent of a polynomial is called its degree.
- Assume that we have 2 polynomials,

  $A(x)= \sum a_i x^i$ &

  $B(x)= \sum b_i x^i$ then $A(x)+B(x)= \sum (a_i + b_i) xi$

POLYNOMIAL REPRESENTATION: FIRST METHOD

```
#define MAX_DEGREE 100
        typedef struct
        {
                int degree;
                float coef [MAX_DEGREE];} polynomial;

polynomial a;
```

Initial version of padd function

```
/* d =a + b, where a, b, and d are polynomials */
 d = Zero ()
while (! IsZero(a) &&! IsZero(b)) do
{
        switch COMPARE (Lead_Exp(a), Lead_Exp(b))
        {
                case -1: d = Attach (d, Coef (b, Lead_Exp(b)), Lead_Exp(b));
                         b = Remove (b, Lead_Exp(b));
                         break;
                case 0: sum = Coef (a, Lead_Exp (a)) + Coef (b, Lead_Exp(b));
                         if (sum)
                         {
                                  Attach (d, sum, Lead_Exp(a));
                                 a = Remove(a , Lead_Exp(a));
                                 b = Remove(b , Lead_Exp(b));
                         }
                         break;
                 case 1: d = Attach (d, Coef (a, Lead_Exp(a)), Lead_Exp(a));
                         a = Remove (a, Lead_Exp(a));
         }
}
insert any remaining terms of a or b into d
```

• If a is of type 'polynomial' then $A(x)= \sum a_i x^i$ can be represented as:

a.degree=n a.coeff[i]=a_{n-i}

• In this representation, we store coefficients 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.coeff[i]=0

• Disadvantage: This representation wastes a lot of space.

For instance, if a.degree<<MAX_DEGREE and polynomial is sparse, then we will not need most of the positions in a.coef[MAX_DEGREE] (sparse means number of terms with non-zero coefficient is small relative to degree of the polynomial).

POLYNOMIAL REPRESENTATION: SECOND METHOD

```
#define MAX_TERMS 100

typedef struct

{       float coef;

        int expon;

} polynomial;

polynomial terms [MAX_TERMS];

int avail=0;
```

• $A(x)=2x^{1000}+1$ and $B(x)=x^4+10x^3+3x^2+1$ can be represented as shown below.

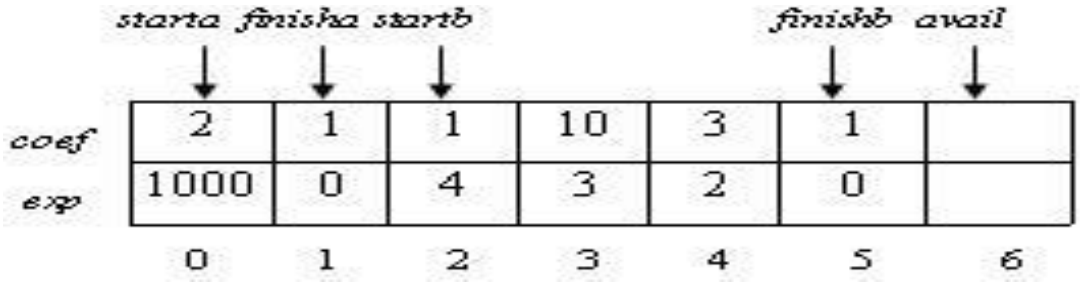


Fig A. Array representation of two polynomials

• startA & startB give the index of first term of A and B respectively.
• finishA & finishB give the index of the last term of A & B respectively.
avail gives the index of next free location in the array.
• Any polynomial A that has 'n' non-zero terms has startA & finishA such that finishA=startA+n-1
• Advantage: This representation solves the problem of many 0 terms since $A(x)-2x^{1000}+1$ uses only 6 units of storage (one for startA, one for finishA, 2 for the coefficients and 2 for the exponents)
• Disadvantage: However, when all the terms are non-zero, the current representation requires about twice as much space as the first one.

POLYNOMIAL ADDITION

Function to add two polynomials

```
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 */
```

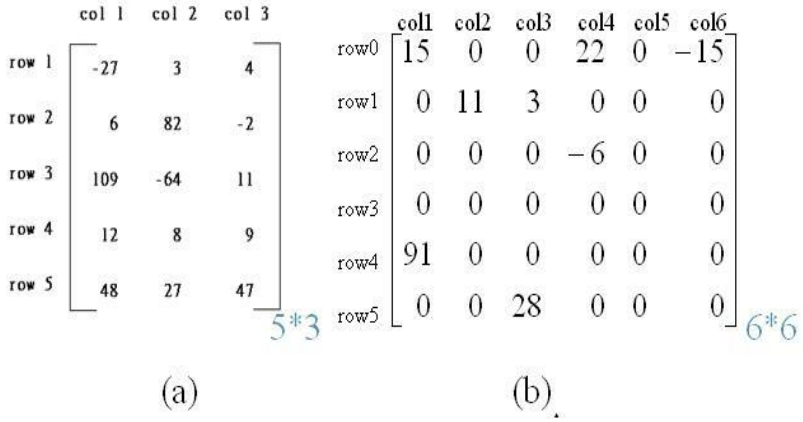
```
                attach(terms[starta].coef, terms[starta].expon); starta++;
            }
        }
            /* add in remaining terms of A(x) */
            for (; starta <= finisha; starta++)
            attach(terms[starta].coef, terms[starta].expon);
            /* add in remaining terms of B(x) */
            for (; startb <= finishb; startb++)
            attach(terms[startb].coef, terms[startb].expon);
            *finishd =avail -1;

}
```

Function to add a new term

```
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 (0);
    }
    terms[avail].coef = coefficient;
    terms[avail++].expon = exponent;
}
```

SPARSE MATRICES

- Sparse matrix contains many zero entries.
- When a sparse matrix is represented as a 2-dimensional array, we waste space.
- For ex, if 100*100 matrixes contain only 100 entries then we waste 9900 out of 10000 memory spaces.
- Solution: Store only the non-zero elements.



(a) (b)

SPARSE MATRIX REPRESENTATION

• We can classify uniquely any element within a matrix by using the triple <row,col,value>. Therefore, we can use an array of triples to represent a sparse matrix.

```
SpareMatrix Create(maxRow,maxCol) ::=

        #define MAX_TERMS 101

        typedef struct term

        {

                int col;

                int row;

                int value;

        } term;

        term    a[MAX_TERMS];
```

Sparse matrix and its transpose stored as triples

	row	col	value
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	−15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	−6
[7]	4	0	91
[8]	5	2	28

(a)

	row	col	value
b[0]	6	6	8
[1]	0	0	15
[2]	0	4	91
[3]	1	1	11
[4]	2	1	3
[5]	2	5	28
[6]	3	0	22
[7]	3	2	−6
[8]	5	0	−15

(b)

Figure 2.5: Sparse matrix and its transpose stored as triples

• a[0].row contains the number of rows; a[0].col contains number of columns and a[0].value contains the total number of nonzero entries.

TRANSPOSING A MATRIX

- To transpose a matrix, we must interchange the rows and columns.
- Each element a[i][j] in the original matrix becomes element b[j][i] in the transpose matrix.
- Algorithm To transpose a matrix:

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

TRANSPOSE OF A SPARSE MATRIX

```
void transpose (term a [], term b [])
{
        /* 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)
        {             /*non zero matrix */
                currentb = 1;
                for (i = 0; i < a [0]. col; i++)      /* transpose by columns in a */
                   for (j = 1; j <= n; j++)          /* find elements from the current column */
                        if (a[j]. col == i)
                        {                           /* element is in current column, add it to b */
                           b[currentb]. row = a[j]. col;
                           b[currentb]. col = a[j]. row;
                           b[currentb].value = a[j].value;
                           currentb++
                        }
         }
}
```

```
void fastTranspose(term a[], term b[])
{/* the transpose of a is placed in b */
   int rowTerms[MAX_COL], startingPos[MAX_COL];
   int i,j, numCols = a[0].col, numTerms = a[0].value;
   b[0].row = numCols;  b[0].col = a[0].row;
   b[0].value = numTerms;
   if (numTerms > 0) { /* nonzero matrix */
      for (i = 0; i < numCols; i++)
         rowTerms[i] = 0;
      for (i = 1; i <= numTerms; i++)
         rowTerms[a[i].col]++;
      startingPos[0] = 1;
      for (i = 1; i < numCols; i++)
         startingPos[i] =
                     startingPos[i-1] + rowTerms[i-1];
      for (i = 1; i <= numTerms; i++) {
         j = startingPos[a[i].col]++;
         b[j].row = a[i].col;   b[j].col = a[i].row;
         b[j].value = a[i].value;
      }
   }
}
```

Program 2.9: Fast transpose of a sparse matrix

Strings

- A string is a sequence of characters. In computer science, strings are more often used than numbers. We have all used text editors for editing programs and documents. Some of the Important Operations which are used on strings are: searching for a word, find -and -replace operations, etc.
- There are many functions which can be defined on strings. Some important functions are

  - **String length:** Determines length of a given string.
  - **String concatenation:** Concatenation of two or more strings.
  - **String copy:** Creating another string which is a copy of the original or a copy of a part of the original.
  - **String matching:** Searching for a query string in given string.

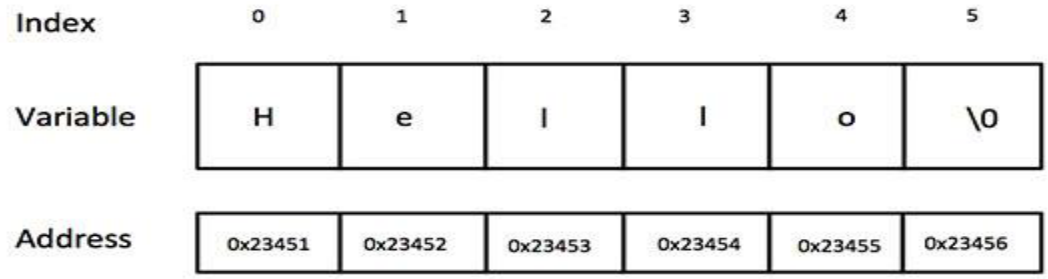
The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

If you follow the rule of array initialization then you can write the above statement as follows −

char greeting [] = "Hello";

Following is the memory presentation of the above defined string in C



Note: Actually, you do not place the null character at the end of a string constant. The C compiler automatically places the '\0' at the end of the string when it initializes the array.

Basic Terminology

Basic Terminology each programming language contains a character set that is used to communicate with the computer. This set usually includes the following:

Alphabets :	**A, B, C, ... X, Y, Z**
Digits :	**0, 1, 2, 3, 4, 5, 6, 7, 8, 9**
Special characters :	**+, -, *, (,), =, ^ etc..**

A finite sequence S of zero or more characters is called a string. The number of characters in a string is called its length. The string with zero characters is called the empty string or the null string. Specific strings will be denoted by enclosing their characters in single quotation marks. The quotation marks will also serve as string delimiters. Hence

'THE END' 'HELLO' 'WELCOME'

are strings with lengths 7, 5, 7 respectively. We emphasize that the blank space is a character and hence contributes to the length of the string.

Let S1 and S2 be strings. The string consisting of the characters of S1 followed by the characters of S2 is called **concatenation** of S1 and S2; it will be denoted by S1//S2. For example,

'HI'// 'ABC' = 'HIABC

Storing String

Strings are stored in three types of structures

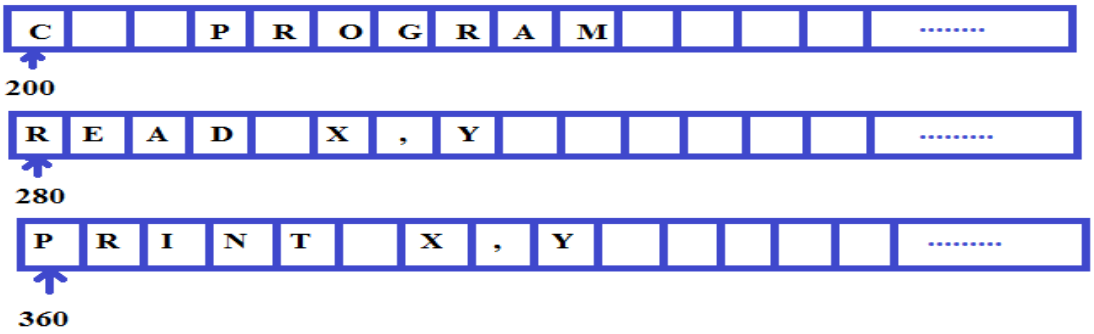
1) Fixed length structures
2) Variable length structures with fixed maximum
3) Linked structure

Fixed length structures

In fixed-length storage each line of print is viewed as a record, where all records have the same length, i.e. where each record accommodates the same number of characters. Since data are frequently input on terminals with 80-column images or using 80-columns cards, we will assume our records have length 80 unless otherwise stated or implied.

For example:
C PROGRAM
READ X, Y

PRINT X, Y



Records Stored Sequentially in Computer

The main disadvantages are:

- Time is wasted reading an entire record if most of the storage consists of inessential blank spaces.

- Certain records may require more space than available.
- When the correction consists of more or fewer characters than the original text, changing a misspelled word requires the entire record to be changed.

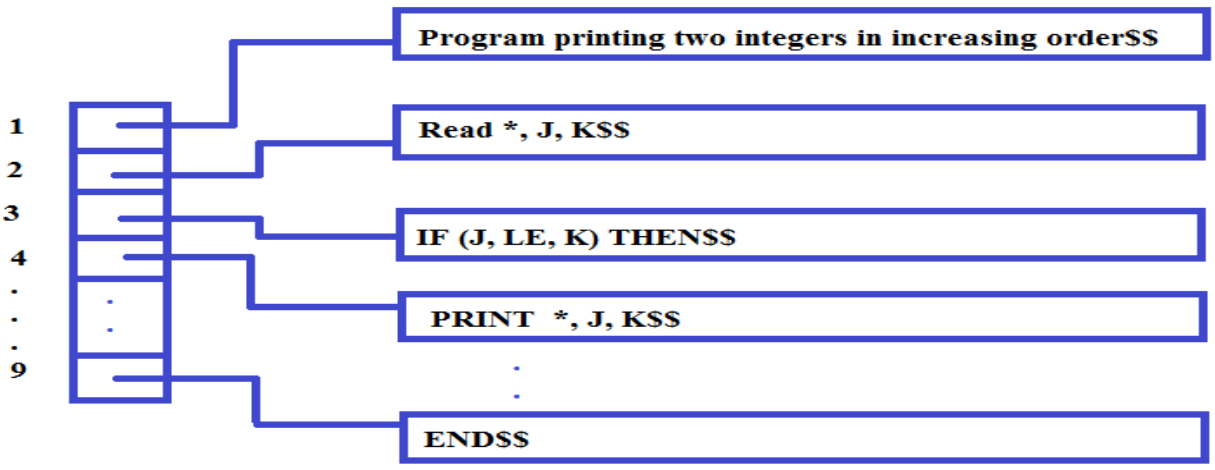
Variable length structures with fixed maximum

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

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

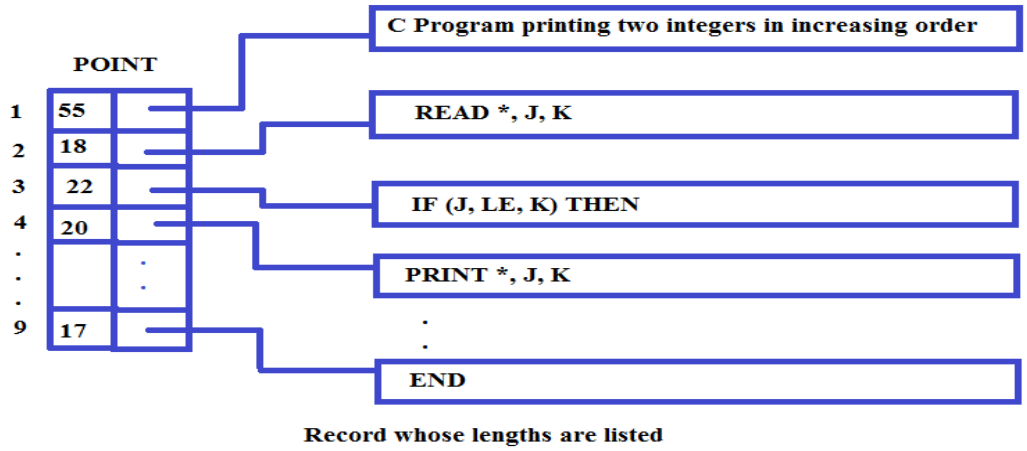
C Program printing two integers in increasing order

```
READ *, J, K
IF (J.LE.K) THEN
        PRINT *, J, K
ELSE
        PRINT *, K, J
ENDIF
STOP
END
```



Records with sentinels

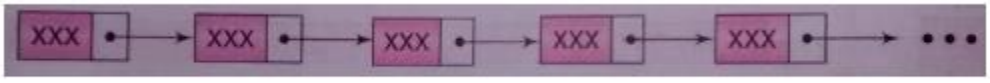
Second method



Remark: One might be tempted to store strings one after another by using some separation marker, such as the two dollar signs ($$) in first figure, or by using a pointer array giving the location of the strings, as shown in second figure. These ways of storing will obviously save and are sometimes used in secondary memory when records are relatively permanent and require little change. However, such methods of storage are usually inefficient when the strings and their lengths are frequently being changed.

Linked Storage

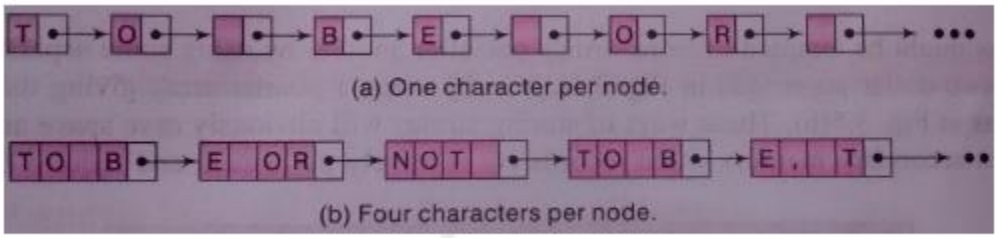
- Most extensive word processing applications, strings are stored by means of linked lists.
- In a one way linked list, a linearly ordered sequence of memory cells called nodes, where each node contains an item called a ***link,*** which points to the next node in the list, i.e., which consists the address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



Strings may be Stored in linked list as follows:

Each memory cell is assigned one character or a fixed number of characters and a link contained in the cell gives the address of the cell containing the next character or group of character in the string.

Ex: TO BE OR NOT TO BE



CHARACTER DATA TYPE

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

Constants

Many programming languages denote 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 given string

(3) The length of the substring or the position of the last character of the substring.

Syntax: SUBSTRING (string, initial, length)

The syntax denotes 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 S1 and S2 be string. The concatenation of S1 and S2 which is denoted by S1 // S2, is the string consisting of the characters of S1 followed by the character of S2.

Ex:

(a) Suppose S1 = 'MARK' and S2= 'TWAIN' then

S1 // S2 = '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*

Figure 2.8: C string functions

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.

$$W_K = \text{SUBSTRING (T, K, LENGTH (P))}$$

- Where, W_K denote the substring of T having the same length as P and beginning with the K^{th} character of T.
- First compare P, character by character, with the first substring, W_1. If all the characters are the same, then $P = W_1$ and so P appears in T and INDEX (T, P) = 1.
- Suppose it is found that some character of P is not the same as the corresponding character of W1. Then $P \neq W_1$

- Immediately move on to the next substring, W_2 That is, compare P with W_2. If $P \neq W_2$ then compare P with W3 and so on.
- The process stops, When P is matched with some substring WK and so P appears in T and INDEX (T, P) = K or When all the WK'S with no match and hence P does not appear in T.
- The maximum value MAX of the subscript K is equal to LENGTH(T) -LENGTH(P) +1.

Algorithm: (Pattern Matching)

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

1. [Initialize.] Set K: = 1 and MAX: = S - R + 1
2. Repeat Steps 3 to 5 while $K \leq MAX$
3. Repeat for L = 1 to R: [Tests each character of P]
   If $P[L] \neq T[K + L - l]$, 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 = 0
7. Exit

Observation of algorithms

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

Complexity

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

Second Pattern Matching Algorithm

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

For definiteness, suppose

P = aaba

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

The table is obtained as follows.

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

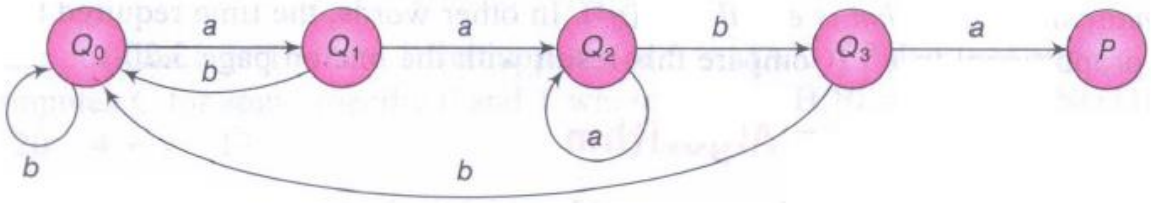
A is the largest Q that is a terminal substring of Q1*b* = *ab,* so *f (Q1, b)* = Q0

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 S_K = P, the desired pattern. In this case, P does appear in T and its index is
   K - LENGTH (P).
2. No state S_1, S_2, ..., S_{N+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 = T1 T2 T3 …… T_N. The algorithm finds the INDEX of P in T.

1. [Initialize] set K: =1 and S_1 = Q0
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.

Pattern matching by checking end indices first

```
int nfind(char *string, char *pat)
{/* match the last character of pattern first, and
    then match from the beginning */
   int i,j,start = 0;
   int lasts = strlen(string)-1;
   int lastp = strlen(pat)-1;
   int endmatch = lastp;

   for (i = 0; endmatch <= lasts; endmatch++, start++) {
      if (string[endmatch] == pat[lastp])
         for (j = 0, i = start; j < lastp &&
                        string[i] == pat[j]; i++,j++)
            ;
      if (j == lastp)
         return start; /* successful */
      }
      return -1;
}
```

Program 2.13: Pattern matching by checking end indices first

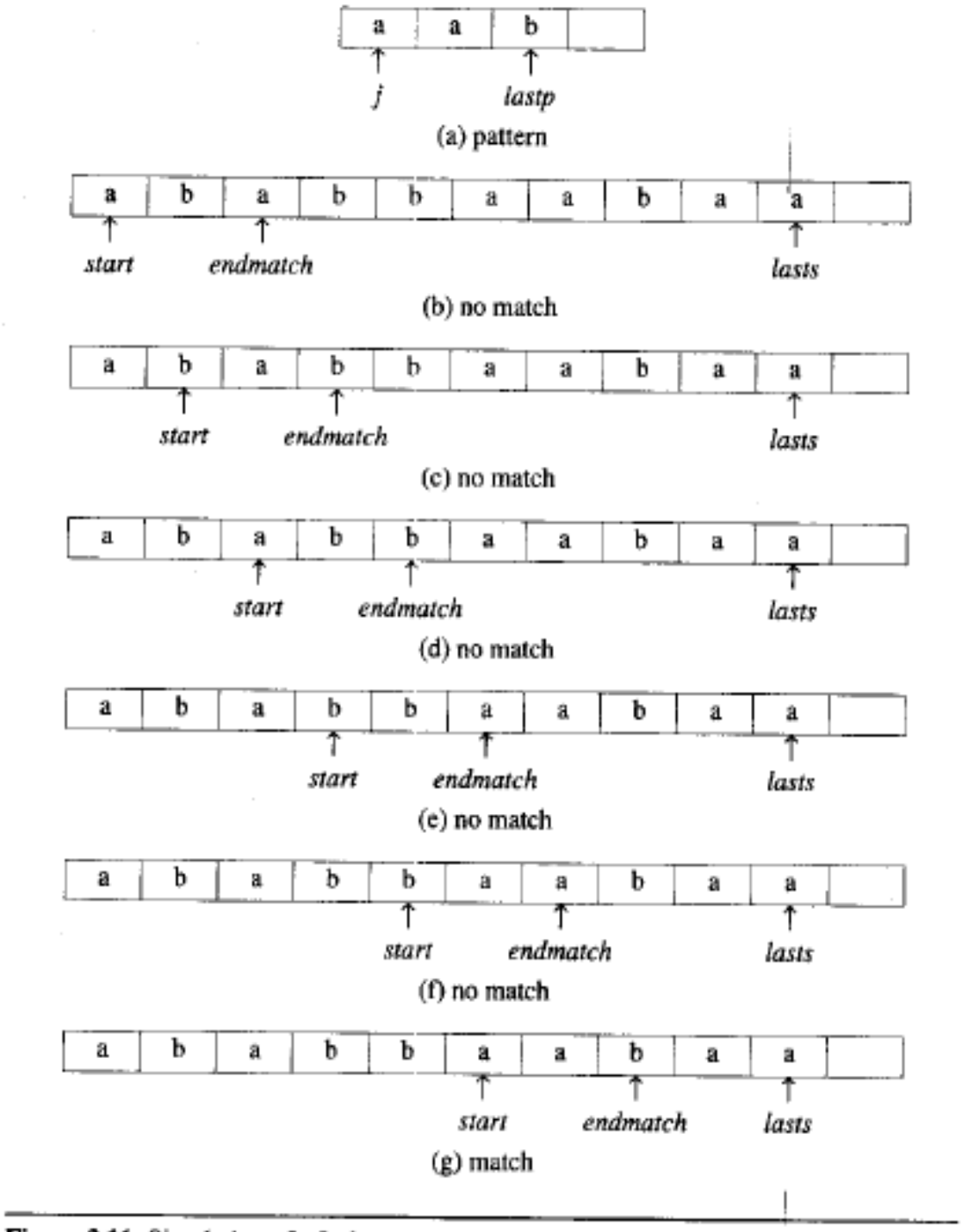


Figure 2.11: Simulation of *nfind*

Knuth, Morris, Pratt Pattern Matching algorithm.

Definition: If $p = p_0 p_1 \cdots p_{n-1}$ is a pattern, then its *failure function*, f, is defined as:

$$f(j) = \begin{cases} \text{largest } i < j \text{ such that } p_0 p_1 \cdots p_i = p_{j-i} p_{j-i+2} \cdots p_j \text{ if such an } i \geq 0 \text{ exists} \\ -1 \qquad \text{otherwise} \end{cases} \quad \square$$

For the example pattern, *pat* = *abcabcacab*, we have:

j	0	1	2	3	4	5	6	7	8	9
pat	*a*	*b*	*c*	*a*	*b*	*c*	*a*	*c*	*a*	*b*
f	−1	−1	−1	0	1	2	3	−1	0	1

```
int pmatch(char *string, char *pat)
{/* Knuth, Morris, Pratt string matching algorithm */
    int i = 0, j = 0;
    int lens = strlen(string);
    int lenp = strlen(pat);
    while ( i < lens && j < lenp ) {
        if (string[i] == pat[j]) {
            i++; j++; }
        else if (j == 0) i++;
            else j = failure[j-1]+1;
    }
    return ( (j == lenp) ? (i-lenp) : -1);
}
```

Program 2.14: Knuth, Morris, Pratt pattern matching algorithm

```
void fail(char *pat)
{/* compute the pattern's failure function */
    int n = strlen(pat);
    failure[0] = -1;
    for (j=1; j < n; j++) {
        i = failure[j-1];
        while ((pat[j] != pat[i+1]) && (i >= 0))
            i = failure[i];
        if (pat[j] == pat[i+1])
            failure[j] = i+1;
        else failure[j] = -1;
    }
}
```

Program 2.15: Computing the failure function

Online weblinks

1. **https://www.youtube.com/watch?v=sPTpClcwsbs&list=PLVDfFatHsysQGtvuaDbTTkle69C0wHaK_&index=5&t=50s**
2. **https://www.youtube.com/watch?v=M1ZRl4d7_XY&list=PLVDfFatHsysQGtvuaDbTTkle69C0wHaK_&index=6&t=28s**
3. **https://www.youtube.com/watch?v=3TBrP0nOwo4&list=PLVDfFatHsysQGtvuaDbTTkle69C0wHaK_&index=7&t=11s**
4. **https://www.youtube.com/watch?v=vcpUIcTAFks&list=PLVDfFatHsysQGtvuaDbTTkle69C0wHaK_&index=8&t=476s**
5. **https://www.youtube.com/watch?v=0nXxVtoLf6s&list=PLVDfFatHsysQGtvuaDbTTkle69C0wHaK_&index=9&t=23s**
6. **https://www.youtube.com/watch?v=f9vjC7FDNBA&list=PLVDfFatHsysQGtvuaDbTTkle69C0wHaK_&index=10&t=19s**
7. **https://www.youtube.com/watch?v=NDmS7W0MDLU&list=PLVDfFatHsysQGtvuaDbTTkle69C0wHaK_&index=11&t=2s**
8. **https://www.youtube.com/watch?v=4r1PFvtQbIY&list=PLVDfFatHsysQGtvuaDbTTkle69C0wHaK_&index=2&t=6s**
9. **https://www.youtube.com/watch?v=66ZRH2cfRR0&list=PLVDfFatHsysQGtvuaDbTTkle69C0wHaK_&index=3&t=4s**

Question Bank

1. Define Data Structures. Give its classifications. Explain the different operations that can be performed on data structures.
2. What is a pointer? How do you declare and initialize the pointer? How do you access the value pointed to by a pointer?
3. Define Structure with suitable example.
4. Write a C program with appropriate structure definition and variable declaration to read and display information about 5 Employees using nested structures. Consider the following fields like Ename, Empid, DOJ (day, month, year) and Salary (Basic, DA, HRA).
5. Differentiate Structures and Unions.
6. Develop a structure to represent planets in the solar system. Each planet has fields for the planets name, its distance from the sun in miles and the number of moons it has. Write a program to read the data for each planet and store. Also print the name of the planet that has the highest number moons.
7. What is Static and Dynamic memory allocation? Explain with examples, the dynamic memory allocation functions.
8. Write a C function to swap two numbers using pointers.
9. Define a polynomial. How would you represent two polynomials using array of structures? Write a function to add two polynomials.
10. Construct an algorithm to transpose a matrix, Express the given matrix as triplets and find its transpose.

$$a = \begin{bmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{bmatrix}$$

11. Write the different methods to represent polynomial, design an algorithm to add two polynomials.

12. Explain various operations that can be performed on structures with an example (function).
13. Consider two polynomials $A(x)=2x^{1000}+1$ and $B(x)=x^4+10x^3+3x^2+1$ with a diagram show how these two polynomials are stored using 1D array and also give its C representation.
14. A C program contains the following declaration

```
int X [8] = {10,20,30,40,50,60,70,80};
```

i) What is the meaning of X?

ii) What is the meaning of (X+2)?

iii) What is the meaning of *X?

iv) What is the meaning of (*X+2)?

v) What is the meaning of *(X+2)?

15. What is the output of the following code?

```
int num [5] = {3,4,6,2,1}
    int *p=num;
    int *q=num+2;
    int *r=&num [1];
    printf(%d%d", num[2],*(num+2));
    printf(%d%d", *p,*(p+1));
    printf(%d%d", *q,*(q+1));
    printf(%d%d", *r,*(r+1));
```

16. Design an algorithm to search a key element in an array using binary search.
17. Design transpose and fast transpose algorithm to transpose given sparse matrix.
18. What is dynamic memory allocation? Design an algorithm to create 1-D and 2-D arrays dynamically.
19. Construct an algorithm to sort n integers using bubble sort and estimate its efficiency.

20. Explain how pointers and be dangerous? Give advantages and disadvantages of pointers.
21. Write a program in C to read a sparse matrix of integer values and search this matrix for an element specified by the user.
22. What is polynomial? What is the degree of the polynomial? Write a function to add two polynomials.
23. Construct an algorithm to search a pattern string in the text string using automata and apply the same to search for the pattern **P=aaba** from the text **T=aabcaba**.
24. Write the Knuth Marries Pratt pattern matching algorithm and apply the same to search the pattern **P=abcdabcy** in the text **T=abcxabcdabxabcdabcdabcy.**
25. Write and explain different pattern matching algorithms with an example for each.
26. With a neat block diagram, explain with example the different methods of storing of strings.
27. What is an array? Design an algorithm to insert and delete an element from the specified passion of a 1-D array.

REFERENCES

1. Fundamentals of Data Structures in C - Ellis Horowitz and Sartaj Sahni, 2nd edition, Universities Press,2014
2. Data Structures - Seymour Lipschutz, Schaum's Outlines, Revised 1st edition, McGraw Hill, 2014