MODULE 1: INTRODUCTION

DATA STRUCTURES

A data structure is a specialized format for organizing and storing data. General data structure types include the array, the file, the record, the table, the tree, and so on. Any data structure is designed to organize data to suit a specific purpose so that it can be accessed and worked with in appropriate ways. In computer programming, a data structure may be selected or designed to store data for the purpose of working on it with various algorithms

Basic Terminology: Elementary Data Organization:

Data: Data are simply values or sets of values.

Information is organized or classified data, which has some meaningful values for the receiver. Information is the processed data on which decisions and actions are based.

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

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

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

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

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

Entities with similar attributes form an **entity set**. Each attribute of an entity set has a range of values, the set of all possible values that could be assigned to the particular attribute.

The term “information” is sometimes used for data with given attributes, of, in other words meaningful or processed data.

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

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

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

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

Records may also be classified according to length.

A file can have fixed-length records or variable-length records.

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

- In variable-length records file records may contain different lengths.

Example: Student records have variable lengths, since different students take differe nt 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.

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.

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:

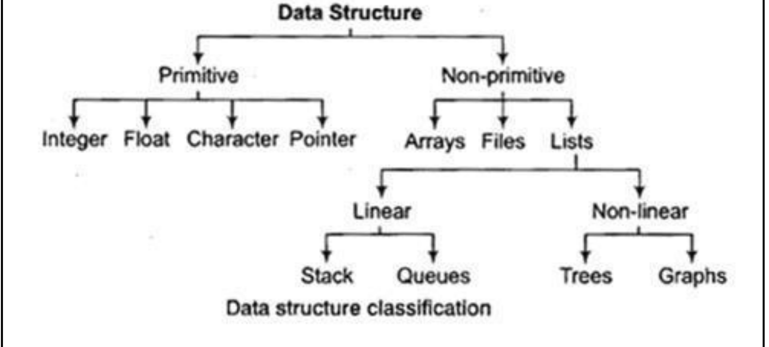
A data structure is said to be linear if its elements form a sequence or a linear list. There are basically two ways of representing such linear structure in memory.

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 data structure.



Data structure classification

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 a1, a2, a3….. an

by the bracket notation A [1], A [2], A [3] . . . . . . A [n]

Trees

Data frequently contain a hierarchical relationship between various elements. The data structure which reflects this relationship is called a rooted tree graph or a tree. Some of the basic properties of tree are explained by means of examples

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

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.

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. The data structure which reflects this type of relationship is called a graph.

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)

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

Traversing in Linear Array

Array is a container which can hold a fix number of items and these items should be of the same type. Most of the data structures make use of arrays to implement their algorithms.
Traverse − print all the array elements one by one.or process the each element one by one . Let A be a collection of data elements stored in the memory of the computer. Suppose we want to print the content of each element of A or suppose we want to count the number of elements of A with given property. This can be accomplished by traversing A, that is, by accessing and processing (frequently called visiting) each element of An exactly once.

Algorithm

```
Step 1 :    [Initialization]  Set I = LB

Step 2 :     Repeat Step 3 and Step 4 while I  < = UB

step 3 :      [ processing ] Process the A[I] element
```

Step 4 : [Increment the counter] I = I + 1
[End of the loop of step 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 K^{th} position in LA.

```
1. [Initialize counter]                  set J:= N
2. Repeat step 3 and 4                   while J ≥ 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
```

Searching
search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.

Algorithm

```
Linear Search ( Array A, Value x)

Step 1: Set i to 1
Step 2: if i > n then go to step 7
Step 3: if A[i] = x then go to step 6
Step 4: Set i to i + 1
Step 5: Go to Step 2
Step 6: Print Element x Found at index i and go to step 8
Step 7: Print element not found
Step 8: Exit
```

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 K^{th} element from LA

```
1. Set ITEM:= LA[K]
2. Repeat for J = K to N – 1
           [Move J + 1 element upward]        set LA[J]:= LA[J+1]
   [End of loop]

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

4. Exit
```

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,

$$A[I] < A[2] < A[3] < \ldots < A[N]$$

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.
```

Merge two arrays

1. Create an array arr3[] of size n1 + n2.
2. Simultaneously traverse arr1[] and arr2[].
   - Pick smaller of current elements in arr1[] and arr2[], copy this smaller element to next position in arr3[] and move ahead in arr3[] and the array whose element is picked.
3. If there are are remaining elements in arr1[] or arr2[], copy them also in arr3[].

Abstract Data Types

Abstract Data type (ADT) is a type (or class) for objects whose behavior is defined by a set of value and a set of operations.The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called "abstract" because it gives an implementation independent view. The process of providing only the essentials and hiding the details is known as abstraction.

The user of data type need not know that data type is implemented, for example, we have been using int, float, char data types only with the knowledge with values that can take and operations that can be performed on them without any idea of how these types are implemented. So a user only needs to know what a data type can do but not how it will do it. We can think of ADT as a black box which hides the inner structure and design of the data type. Now we'll define three ADTs namely List ADT, StackADT, Queue ADT.

List ADT

A list contains elements of same type arranged in sequential order and following operations can be performed on the list.
get() – Return an element from the list at any given position.
insert() – Insert an element at any position of the list.
remove() – Remove the first occurrence of any element from a non-empty list.
removeAt() – Remove the element at a specified location from a non-empty list.
replace() – Replace an element at any position by another element.
size() – Return the number of elements in the list.

isEmpty() – Return true if the list is empty, otherwise return false.
isFull() – Return true if the list is full, otherwise return false.

Stack ADT
A Stack contains elements of same type arranged in sequential order. All operations takes place at a single end that is top of the stack and following operations can be performed:
push() – Insert an element at one end of the stack called top.
pop() – Remove and return the element at the top of the stack, if it is not empty.
peek() – Return the element at the top of the stack without removing it, if the stack is not empty.
size() – Return the number of elements in the stack.
isEmpty() – Return true if the stack is empty, otherwise return false.
isFull() – Return true if the stack is full, otherwise return false

Algorithm INTRODUCTION

What is an Algorithm?

Informal Definition:

An Algorithm is any well-defined computational procedure that takes some value or set of values as input and produces a set of values or some value as output. Thus algorithm is a sequence of computational steps that transforms the input into the output.

Formal Definition:

An ***algorithm*** is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.
Properties of an algorithm

INPUT Zero or more quantities are externally supplied.
OUTPUT At least one quantity is produced.
DEFINITENESS each instruction is clear and unambiguous.
FINITENESS if we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
EFFECTIVENESS every instruction must very basic so that it can be carried out, in principle, by a person using only pencil & paper.

Performance of a program: time and space tradeoff

The performance of a program is the amount of computer memory and time needed to run a program. We use two approaches to determine the performance of a program. One is analytical, and the other experimental. In performance analysis we use analytical methods, while in performance measurement we conduct experiments.

Time Complexity:
The time needed by an algorithm expressed as a function of the size of a problem is called the time complexity of the algorithm. The time complexity of a program is the amount of computer time it needs to run to completion.
The limiting behavior of the complexity as size increases is called the asymptotic time complexity. It is the asymptotic complexity of an algorithm, which ultimately determines the size of problems that can be solved by the algorithm.

Space Complexity:
The space complexity of a program is the amount of memory it needs to run to completion. The space need by a program has the following components:
Instruction space: Instruction space is the space needed to store the compiled version of the program instructions.
Data space: Data space is the space needed to store all constant and variable values.
Data space has two components:
• Space needed by constants and simple variables in program.
• Space needed by dynamically allocated objects such as arrays and class instances.
Environment stack space: The environment stack is used to save information needed to resume execution of partially completed functions.
Instruction Space: The amount of instructions space that is needed depends on factors such as:
• The compiler used to complete the program into machine code.
• The compiler options in effect at the time of compilation
• the target computer.

Algorithm Design Goals
The three basic design goals that one should strive for in a program are:
1. Try to save Time
2. Try to save Space
3. Try to save Face

a program that runs faster is a better program, so saving time is an obvious goal. Likewise, a program that saves space over a competing program is considered desirable. We want to “save face” by preventing the program from locking up or generating reams of garbled data.

Algorithm Specification
Algorithm can be described in three ways.

1. Natural language like English: When this way is choosed care should be taken, we should ensure that each & every statement is definite.

2. Graphic representation called flowchart: This method will work well when the algorithm is small& simple.

3. Pseudo-code Method: In this method, we should typically describe algorithms as program, which resembles programming language constructs

Pseudo-Code Conventions:

1. Comments begin with // and continue until the end of line.

2. Bocks are indicated with matching braces {and}.

3. An identifier begins with a letter. The data types of variables are not explicitly declared.

4. Compound data types can be formed with records. Here is an example,

Node. Record { data type – 1 data-1; . . . data type – n data – n; node * link; }

Here link is a pointer to the record type node. Individual data items of a record can be accessed with → and period.

5. Assignment of values to variables is done using the assignment statement.

<Variable>:= <expression>;

6. There are two Boolean values TRUE and FALSE.
    - Logical Operators AND, OR, NOT

Relational Operators <, <=,>,>=, =, !=

7. The following looping statements are employed. For, while and repeat-until

While Loop:

While < condition > do

{ <statement-1> **. . .** <statement-n> }

For Loop:

For variable: = value-1 to value-2 step step do

{ <statement-1> **. . .** <statement-n> }

repeat-until:

repeat <statement-1> **. . .** <statement-n> until<condition>

8. A conditional statement has the following forms.
   - If <condition> then <statement>
   - If <condition> then <statement-1> Else <statement-1>
   - **Case statement:**
     Case

     { **:** <condition-1> **:** <statement-1>

     . . .

     : <condition-n> **:** <statement-n>

     : else **:** <statement-n+1>

     }

Input and output are done using the instructions read & write.

Orders Of Growth

♦A difference in running times on small inputs is not what really distinguishes efficient algorithms from inefficient ones.

♦When we have to compute, for example, the greatest common divisor of two small numbers, it is not immediately clear how much more efficient Euclid‘s algorithm is compared to the other two algorithms discussed in previous section or even why we should care which of them is faster and by how much. It is only when we have to find the greatest common divisor of two large numbers that the difference in algorithm efficiencies becomes both clear and important.

For large values of *n*, it is the function‘s order of growth that counts:

Complexity of Algorithms

The complexity of an algorithm M is the function f(n) which gives the running time and/or storage space requirement of the algorithm in terms of the size ‘n’ of the input data. Mostly, the storage space required by an algorithm is simply a multiple of the data size ‘n’. Complexity shall refer to the running time of the algorithm.

The function f(n), gives the running time of an algorithm, depends not only on the size ‘n’ of the input data but also on the particular data. The complexity function f(n) for certain cases are:

1. Best Case : The minimum possible value of f(n) is called the best case.

2. Average Case : The expected value of f(n).

3. Worst Case : The maximum value of f(n) for any key possible input.

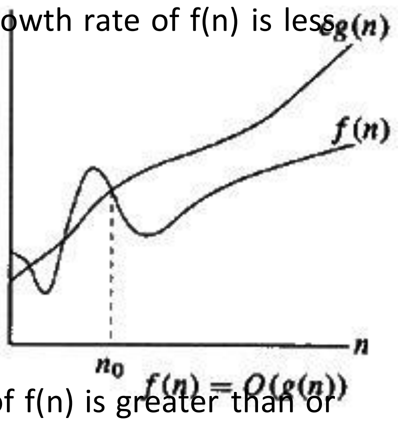
The field of computer science, which studies efficiency of algorithms, is known as analysis of algorithms.

Algorithms can be evaluated by a variety of criteria. Most often we shall be interested in the rate of growth of the time or space required to solve larger and larger instances of a problem. We will associate with the problem an integer, called the size of the problem, which is a measure of the quantity of input data.
Asymptotic notations
The following notations are commonly use notations in performance analysis and used to characterize the complexity of an algorithm:

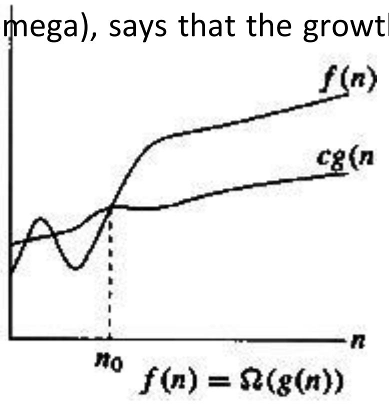
1. Big–OH (O)

2. Big–OMEGA (Ω),
3. Big–THETA (θ) and

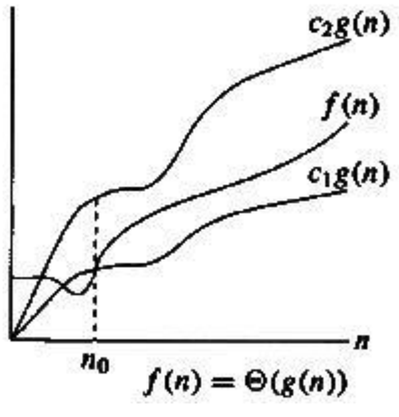
Big–OH O (Upper Bound)
f(n) <O(g(n)), (pronounced order of or big oh), says that the growth rate of f(n) is less than or equal (<) that of g(n).



Big–OMEGA ⍰ (Lower Bound)
f(n) > ⍰(g(n)) (pronounced omega), says that the growth rate of f(n) is greater than or equal to (>) that of g(n).



Big–THETA ⍰ (Same order)
g1(n) <f(n) <g2(n) (pronounced theta), says that the growth rate of f(n) equals (=) the growth rate of g(n) [if f(n) = O(g(n)) and T(n) = ⍰ (g(n)].



<u>STRING</u>

<u>BASIC TERMINOLOGY:</u>

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

Alphabet:	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Digits:	0 1 2 3 4 5 6 7 8 9
Special characters:	+ - / * ( ) , . $ = ' _ (Blank space)

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

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

Empty or Null String: The string with zero characters.

Concatenation: Let S1 and S2 be the strings. The string consisting of the characters of S1 followed by the character S2 is called Concatenation of S1 and S2. Ex: 'THE' // 'END' = 'THEEND'

'THE' // ' ' // 'END' = 'THE END'

Substring: A string Y is called substring of a string S if there exist string X and Z such that S = X // Y // Z

If X is an empty string, then Y is called an Initial substring of S, and Z is an empty string then Y is called a terminal substring of S.

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

STRINGS IN C

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

Declaration 1:

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

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.

The main advantages of this method are

1. The ease of accessing data from any given record
2. The ease of updating data in any given 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 than available
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

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.

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 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 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

C Program to Concat Two Strings without Using Library Function

```
#include<stdio.h>
#include<string.h>
void concat(char[], char[]);
int main() {
    char s1[50], s2[30];
    printf("\nEnter String 1 :");
    gets(s1);
    printf("\nEnter String 2 :");
    gets(s2);
    concat(s1, s2);
    printf("\nConcated string is :%s", s1);
    return (0);
}
void concat(char s1[], char s2[]) {
    int i, j;
    i = strlen(s1);
    for (j = 0; s2[j] != '\0'; i++, j++) {
        s1[i] = s2[j];
    }
    s1[i] = '\0';
        }
```

```
Enter String 1 : Ankit
Enter String 2 : Singh
Concated string is : AnkitSingh
```

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.

```
C program to find the length of a string without using the
 * built-in function
 */
#include <stdio.h>

void main()
{
   char string[50];
   int i, length = 0;

   printf("Enter a string \n");
   gets(string);
   /*  keep going through each character of the string till its end */
   for (i = 0; string[i] != '\0'; i++)
   {
      length++;
   }
   printf("The length of a string is the number of characters in it \n");
   printf("So, the length of %s = %d\n", string, length);
}
```

```
Enter a string
hello
The length of a string is the number of characters in it
So, the length of hello = 5
```

C strcmp()

The strcmp() function compares two strings and returns 0 if both strings are identical.

C strcmp() Prototype

```
int strcmp (const char* str1, const char* str2);
```

The strcmp() function takes two strings and return an integer.

The strcmp() compares two strings character by character. If the first character of two strings are equal, next character of two strings are compared. This continues until the corresponding characters of two strings are different or a null character '\0' is reached.

It is defined in string.h header file.

Return Value from strcmp()

Return Value	Remarks
0	if both strings are identical (equal)
Negative	if the ASCII value of first unmatched character is less than second.
positive integer	if the ASCII value of first unmatched character is greater than second.

C program to compare two strings without using string functions

```
#include<stdio.h>

int stringCompare(char[],char[]);
int main(){
    char str1[100],str2[100];
    int compare;

    printf("Enter first string: ");
    scanf("%s",str1);

    printf("Enter second string: ");
    scanf("%s",str2);

    compare = stringCompare(str1,str2);

    if(compare == 1)
        printf("Both strings are equal.");
    else
        printf("Both strings are not equal");

    return 0;
}

int stringCompare(char str1[],char str2[]){
    int i=0,flag=0;

    while(str1[i]!='\0' && str2[i]!='\0'){
        if(str1[i]!=str2[i]){
            flag=1;
            break;
        }
        i++;
    }

    if (flag==0 && str1[i]=='\0' && str2[i]=='\0')
        return 1;
    else
        return 0;

}
```

Sample output:

```
Enter first string: HELLO
Enter second string: HELLO
Both strings are equal.
```

C strcpy()

The strcpy() function copies the string to the another character array.

strcpy() Function prototype

```
char* strcpy(char* destination, const char* source);
```

The strcpy() function copies the string pointed by `source` (including the null character) to the character array `destination`.

This function returns character array `destination`.

The strcpy() function is defined in `string.h` header file.

String copy without using strcpy in c programming language

```
#include<stdio.h>

void stringCopy(char[],char[]);

int main(){

   char str1[100],str2[100];

   printf("Enter any string: ");
   scanf("%s",str1);

   stringCopy(str1,str2);

   printf("After copying: %s",str2);

   return 0;
}

void stringCopy(char str1[],char str2[]){
   int i=0;

   while(str1[i]!='\0'){
       str2[i] = str1[i];
       i++;
   }

   str2[i]='\0';
}
```

```
Sample output:
Enter any string:HELLO
After copying: HELLO
```

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.

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

- Where, WK 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, W1. If all the characters are the same, then P = W1 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 ≠ W1

- Immediately move on to the next substring, W2 That is, compare P with W2. If P ≠ W2 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)

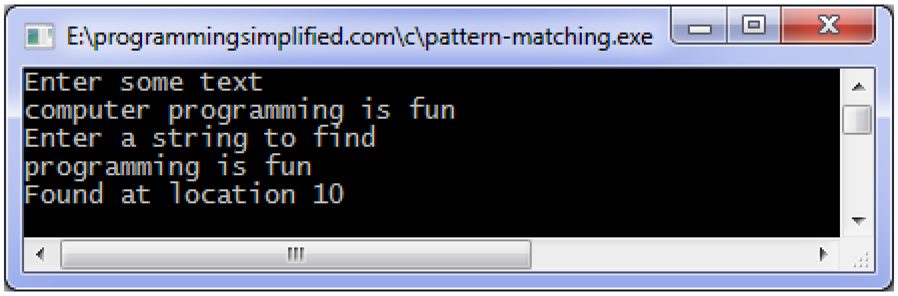
P and T are strings with lengths R and S, and are stored as arrays with one character per element. This algorithm finds the INDEX of P in T.

```
1. [Initialize.] Set K: = 1 and MAX: = S - R + 1

2. Repeat Steps 3 to 5 while K ≤ 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
```

PATTERN MATCHING PROGRAM

```
#include <stdio.h>
#include <string.h>

int match(char [], char []);
int main() {
  char a[100], b[100];
  int position;

  printf("Enter some text\n");
  gets(a);

  printf("Enter a string to find\n");
  gets(b);

  position = match(a, b);

  if (position != -1) {
    printf("Found at location: %d\n", position + 1);
  }
  else {
    printf("Not found.\n");
  }

  return 0;
}

int match(char text[], char pattern[]) {
  int c, d, e, text_length, pattern_length, position = -1;

  text_length    = strlen(text);
  pattern_length = strlen(pattern);

  if (pattern_length > text_length) {
    return -1;
  }

  for (c = 0; c <= text_length - pattern_length; c++) {
    position = e = c;

    for (d = 0; d < pattern_length; d++) {
      if (pattern[d] == text[e]) {
        e++;
      }
      else {
        break;
      }
    }
    if (d == pattern_length) {
      return position;
    }
  }

  return -1;
}
```

OUTPUT



```
Enter some text
computer programming is fun
Enter a string to find
programming is fun
Found at location 10
```