

Module1

1.1. Introduction to Data Structures:

Data structure is a representation of logical relationship existing between individual elements of data. In other words, a data structure defines a way of organizing all data items that considers not only the elements stored but also their relationship to each other. The term data structure is used to describe the way data is stored.

To develop a program of an algorithm we should select an appropriate data structure for that algorithm. Therefore, data structure is represented as: $\text{Algorithm} + \text{Data structure} = \text{Program}$

A data structure is said to be linear if its elements form a sequence or a linear list. The linear data structures like an array, stacks, queues and linked lists organize data in linear order. A data structure is said to be non linear if its elements form a hierarchical classification where, data items appear at various levels.

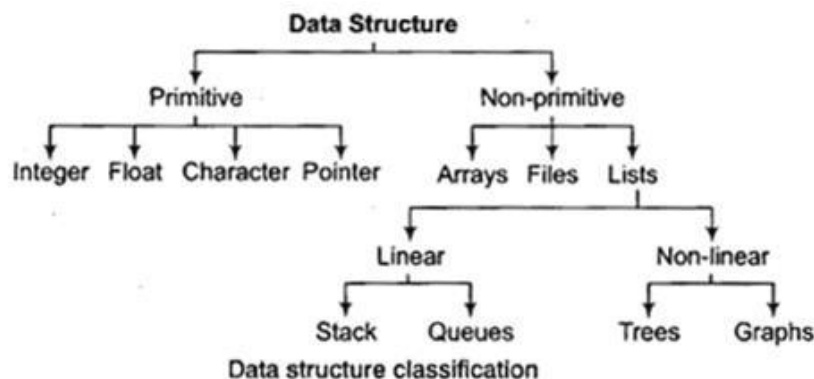
Trees and Graphs are widely used non-linear data structures. Tree and graph structures represent hierarchical relationship between individual data elements. Graphs are nothing but trees with certain restrictions removed.

Data structures are divided into two types:

- Primitive data structures.
- Non-primitive data structures.

Primitive Data Structures are the basic data structures that directly operate upon the machine instructions. They have different representations on different computers. Integers, floating point numbers, character constants, string constants and pointers come under this category.

Non-primitive data structures are more complicated data structures and are derived from primitive data structures. They emphasize on grouping same or different data items with relationship between each data item. Arrays, lists and files come under this category. Figure 1.1 shows the classification of data structures.



Data Structures Operations:

1. Traversing
2. Searching
3. Inserting
4. Deleting
5. Sorting
6. Merging

1. **Traversing**- It is used to access each data item exactly once so that it can be processed.
2. **Searching**- It is used to find out the location of the data item if it exists in the given collection of data items.
3. **Inserting**- It is used to add a new data item in the given collection of data items.
4. **Deleting**- It is used to delete an existing data item from the given collection of data items.
5. **Sorting**- It is used to arrange the data items in some order i.e. in ascending or descending order in case of numerical data and in dictionary order in case of alphanumeric data.
6. **Merging**- It is used to combine the data items of two sorted files into single file in the sorted form.

Review of Arrays

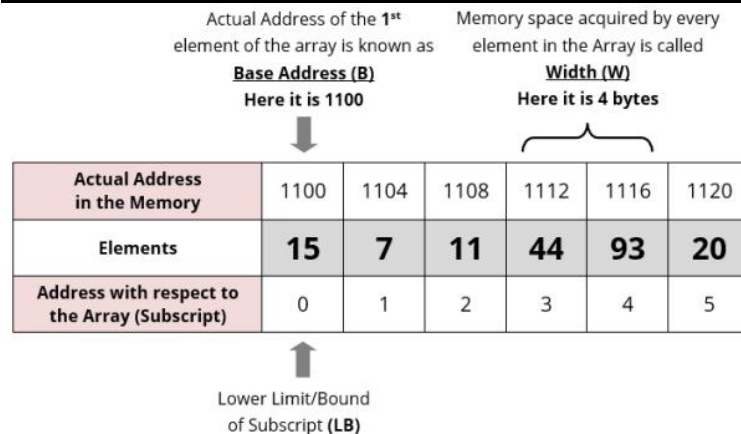
The simplest type of data structure is a linear array. This is also called one dimensional array.

Definition:

Array is a data structure that can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type. An array holds several values of the same kind. Accessing the elements is very fast. It may not be possible to add more values than defined at the start, without copying all values into a new array. An array is stored so that the position of each element can be computed from its index.

For example, an array of 10 integer variables, with indices 0 through 9, may be stored as 10 words at memory addresses 2000, 2004, 2008, 2036, so that the element with index i has the address $2000 + 4 \times i$.

Address Calculation in single (one) Dimension Array:



Address of an element of an array say “A[I]” is calculated using the following formula:

$$\text{Address of A [I]} = \text{B} + \text{W} * (\text{I} - \text{LB})$$

Where,

B = Base address

W = Storage Size of one element stored in the array (in byte)

I = Subscript of element whose address is to be found

LB = Lower limit / Lower Bound of subscript, if not specified assume 0 (zero)

Example:

Given the base address of an array **B[1300.....1900]** as 1020 and size of each element is 2 bytes in the memory. Find the address of **B[1700]**.

Solution:

The given values are: **B** = 1020, **LB** = 1300, **W** = 2, **I** = 1700

$$\text{Address of A [I]} = \text{B} + \text{W} * (\text{I} - \text{LB})$$

$$= 1020 + 2 * (1700 - 1300)$$

$$= 1020 + 2 * 400$$

$$= 1020 + 800$$

$$= 1820 \text{ [Ans]}$$

Address Calculation in Double (Two) Dimensional Array:

While storing the elements of a 2-D array in memory, these are allocated contiguous memory locations. Therefore, a 2-D array must be linearized so as to enable their storage. There are two alternatives to achieve linearization: Row-Major and Column-Major.

		Column Index			
		0	1	2	3
Row Index	0	8	6	5	4
	1	2	1	9	7
	2	3	6	4	2

Two-Dimensional Array

Row-Major (Row Wise Arrangement)

8	6	5	4	2	1	9	7	3	6	4	2
Row 0				Row 1				Row 2			

Column-Major (Column Wise Arrangement)

8	2	3	6	1	6	5	9	4	4	7	2
Column 0			Column 1			Column 2			Column 3		

C allows for arrays of two or more dimensions. A two-dimensional (2D) array is an array of arrays. A three-dimensional (3D) array is an array of arrays of arrays.

In C programming an array can have two, three, or even ten or more dimensions. The maximum dimensions a C program can have depends on which compiler is being used.

More dimensions in an array means more data be held, but also means greater difficulty in managing and understanding arrays.

How to Declare a Multidimensional Array in C

A multidimensional array is declared using the following syntax:

```
type array_name[d1][d2][d3][d4].....[dn];
```

Where each **d** is a dimension, and **dn** is the size of final dimension.

Examples:

1. **int table[5][5][20];**
2. **float arr[5][6][5][6][5];**

In Example 1:

- **int** designates the array type integer.
- **table** is the name of our 3D array.
- Our array can hold 500 integer-type elements. This number is reached by multiplying the value of each dimension. In this case: $5 \times 5 \times 20 = 500$.

In Example 2:

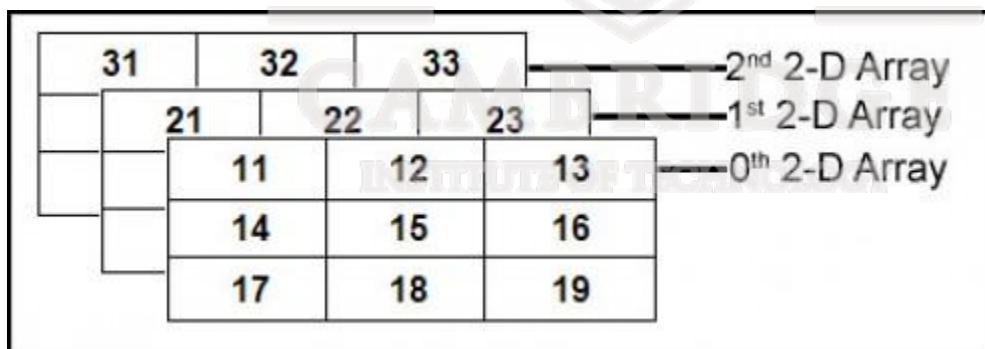
- Array **arr** is a five-dimensional array.
- It can hold 4500 floating-point elements ($5 \times 6 \times 5 \times 6 \times 5 = 4500$).

When it comes to holding multiple values, we would need to declare several variables. But a single array can hold thousands of values.

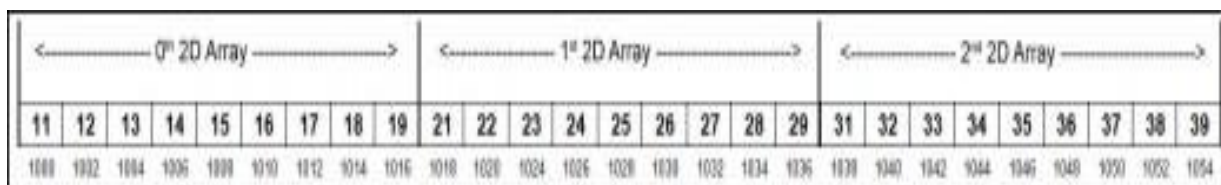
Explanation of a 3D Array

A 3D array is essentially an array of arrays of arrays: it's an array or collection of 2D arrays, and a 2D array is an array of 1D array.

The diagram below shows a 3D array representation:



3D Array Conceptual View



3D array memory map.

Example to show reading and printing a 3D array

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

void main()
{
    int i, j, k;
    int arr[3][3][3]=
    {
        {
            {11, 12, 13},
            {14, 15, 16},
            {17, 18, 19}
        },
        {
            {21, 22, 23},
            {24, 25, 26},
            {27, 28, 29}
        },
        {
            {31, 32, 33},
            {34, 35, 36},
            {37, 38, 39}
        },
    };
    clrscr();
    printf(":::3D Array Elements:::\n\n");
    for(i=0;i<3;i++)
    {
        for(j=0;j<3;j++)
        {
            for(k=0;k<3;k++)
            {
                printf("%d\t",arr[i][j][k]);
            }
            printf("\n");
        }
        printf("\n");
    }
}
```

Note: refer notes for dynamic 1D & 2D arrays and also for pointers.

Structures

Structure is a user defined data type that can hold data items of different data types.

Structure Declaration

```
struct tag { member 1; member 2;
```

```
-----
```

```
-----
```

```
member m; }
```

In this declaration, struct is a required keyword ,tag is a name that identifies structures of this type.

The individual members can be ordinary variables, pointers, arrays or other structures. The member names within a particular structure must be distinct from one another, though a member name can be same as the name of a variable defined outside of the structure and individual members cannot be initialized within a structure-type declaration. For example:

```
struct student
```

```
{ char name [80];
```

```
int roll_no;
```

```
float marks;
```

```
}s1,s2;
```

we can now declare the structure variable s1 and s2 as follows: struct student s1, s2; where

s1 and s2 are structure type variables whose composition is identified by the tag student.

Nested Structure

It is possible to combine the declaration of the structure composition with that of the structure variable .It is then called a nested structure.

```
struct dob
```

```
{ int month;
```

```
int day;
```

```
int year;
```

```
};
```

```
struct student
```



```
{ char name [80];  
  
int roll_no;  
  
float marks;  
  
struct dob d1;  
  
}st;
```

The member of the nested structure can be accessed by using the dot operator twice.(ie)st.d1.year

Array of structures

It is also possible to define an array of structures that is an array in which each element is a structure. The procedure is shown in the following example:

```
struct student{  
  
char name [80];  
  
int roll_no ;  
  
float marks ;  
  
} st [100];
```

In this declaration st is a 100- element array of structures.

It means each element of st represents an individual student record.

Accessing members of a structure using the dot operator

The members of a structure are usually processed individually, as separate entities. Therefore, we must be able to access the individual structure members. A structure member can be accessed by writing

structurevariable. member name.

This period (.) is an operator, it is a member of the highest precedence group, and its associativity is left-to-right.

e.g. if we want to print the detail of a member of a structure then we can write as

printf(“%s”,st.name); or printf(“%d”, st.roll_no) and so on. More complex expressions involving the repeated use of the period operator may also be written. For example, if a structure member is itself a structure, then a member of the embedded structure can be accessed by writing.

variable.member.submember.

Thus in the case of student and dob structure, to access the month of date of birth of a student, we would write

```
st.d1.month // accessing member of nested structure
```

The use of the period operator can be extended to arrays of structure, by writing

```
array [expression]. member
```

Structures members can be processed in the same manner as ordinary variables of the same data type. Single-valued structure members can appear in expressions. They can be passed to functions and they can be returned from functions, as though they were ordinary single-valued variables.

e.g. suppose that s1 and s2 are structure variables having the same composition as described earlier. It is possible to **copy the values of s1 to s2 simply by writing**

```
s2=s1; //copying one structure to another which are of the same type
```

USER-DEFINED DATA TYPES (typedef)

The typedef feature allows users to define new data types that are equivalent to existing data types. Once a user-defined data type has been established, then new variables, arrays, structure and so on, can be declared in terms of this new data type. In general terms, a new data type is defined as

```
typedef type new- type;
```

Where type refers to an existing data type and new-type refers to the new user-defined data type.

e.g. typedef int age;

In this declaration, age is user- defined data type equivalent to type int. Hence, the variable declaration

```
age male, female;
```

is equivalent to writing

```
int age, male, female;
```

The typedef feature is particularly convenient when defining structures, since it eliminates the need to repeatedly write struct tag whenever a structure is referenced. As a result, the structure can be referenced more concisely.

In general terms, a user-defined structure type can be written as

```
typedef struct { member 1; member 2: - - - - - member m; }new-type;
```

The typedef feature can be used repeatedly, to define one data type in terms of other user-defined data types.

STRUCTURES AND POINTERS

The beginning address of a structure can be accessed in the same manner as any other address, through the use of the address (&) operator.

Thus, if variable represents a structure type variable, then & variable represents the starting address of that variable. A pointer to a structure can be defined as follows:

```
struct student *ptr;
```

ptr represents the name of the pointer variable of type student. We can then assign the beginning address of a structure variable to this pointer by writing

```
ptr= &variable; //pointer initialisation
```

Let us take the following example:

```
typedef struct {  
    char name [ 40];  
    int roll_no;  
    float marks;  
}student;  
student s1,*ps;
```

In this example, s1 is a structure variable of type student, and ps is a pointer variable whose object is a structure variable of type student. Thus, the beginning address of s1 can be assigned to ps by writing.

```
ps = &s1;
```

An individual structure member can be accessed in terms of its corresponding pointer variable by using the -> operator (arrow operator)

```
ptr →member
```

Where ptr refers to a structure- type pointer variable and the operator → is comparable to the period (.) operator. The associativity of this operator is also left-to-right.

The operator → can be combined with the period operator (.) to access a submember within a structure. Hence, a submember can be accessed by writing

```
ptr → member.submember
```

PASSING STRUCTURES TO A FUNCTION

There are several different ways to pass structure-type information to or from a function. Structure member can be transferred individually, or entire structure can be transferred. The individual structures members can be passed to a function as arguments in the function call; and a single structure member can be returned via the return statement. To do so, each structure member is treated the same way as an ordinary, single-valued variable.

A complete structure can be transferred to a function by passing a structure type pointer as an argument. It should be understood that a structure passed in this manner will be passed by reference rather than by value. So, if any of the structure members are altered within the function, the alterations will be recognized outside the function. Let us consider the following example:

```
#include <stdio.h>

typedef struct{
    char name[10];
    int roll_no;
    float marks ;
} record student={"Param", 2,99.9};

void adj(record*ptr)
{
    ptr → name="Tanishq";
    ptr → roll_no=3;
    ptr → marks=98.0;
    return;
}

main ( )
{printf("%s%d%f\n", student.name, student.roll_no,student.marks);
  adj(&student);
  printf("%s%d%f\n", student.name, student.roll_no,student.marks);
}
```

Let us consider an example of transferring a complete structure, rather than a structure-type pointer, to the function.

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

```

typedef struct{
char name[10];
int roll_no;
float marks;
}record student={"Param," 2,99.9};
void adj(record stud) /*function definition */
{
stud.name="Tanishq";
stud.roll_no=3;
stud.marks=98.0;
return;
}
main()
{
printf("%s%d%f\n", student.name,student.roll_no,student.marks);
adj(student);
printf("%s%d%f\n", student.name,student.roll_no,student.marks);
}

```

Union is a derived datatype , like structure, i.e. collection of elements of different data types which are grouped together. Each element in a union is called member.

- Union and structure in C are same in concepts, except allocating memory for their members.
- Structure allocates storage space for all its members separately.
- Whereas, Union allocates one common storage space for all its members, or memory space is shared between its members.
- Only one member of union can be accessed at a time. All member values cannot be accessed at the same time in union. But, structure can access all member values at the same time. This is because, Union allocates one common storage space for all its members, where as Structure allocates storage space for all its members separately.
- Many union variables can be created in a program and memory will be allocated for each union variable separately.
- The table below will help you how to form a C union, declare a union, initializing and accessing the members of the union.

Type	Using normal variable	Using pointer variable
Syntax	union tag_name { data type var_name1; data type var_name2; data type var_name3; };	union tag_name { data type var_name1; data type var_name2; data type var_name3; };
Example	union student { int mark; char name[10]; float average; };	union student { int mark; char name[10]; float average; };
Declaring union variable	union student report;	union student *report, rep;
Initializing union variable	union student report = { 100, "Mani", 99.5};	union student rep = {100, "Mani", 99.5}; report = &rep;
Accessing union members	report.mark report.name report.average	report -> mark report -> name report -> average

Example program for C union:

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

union student
{
    char name[20];
    char subject[20];
    float percentage;
};

int main()
{
    union student record1;
    union student record2;

    // assigning values to record1 union variable
    strcpy(record1.name, "Raju");
    strcpy(record1.subject, "Maths");
    record1.percentage = 86.50;

    printf("Union record1 values example\n");
    printf(" Name      : %s \n", record1.name);
    printf(" Subject   : %s \n", record1.subject);
```

```

printf(" Percentage : %f \n\n", record1.percentage);

// assigning values to record2 union variable
printf("Union record2 values example\n");
strcpy(record2.name, "Mani");
printf(" Name      : %s \n", record2.name);

strcpy(record2.subject, "Physics");
printf(" Subject   : %s \n", record2.subject);

record2.percentage = 99.50;
printf(" Percentage : %f \n", record2.percentage);
return 0;
}

```

Output:

```

Union record1 values example
Name :
Subject :
Percentage : 86.500000;
Union record2 values example
Name : Mani
Subject : Physics
Percentage : 99.500000

```

Explanation for above C union program:

There are 2 union variables declared in this program to understand the difference in accessing values of union members.

Record1 union variable:

- “Raju” is assigned to union member “record1.name”. The memory location name is “record1.name” and the value stored in this location is “Raju”.
- Then, “Maths” is assigned to union member “record1.subject”. Now, memory location name is changed to “record1.subject” with the value “Maths” (Union can hold only one member at a time).
- Then, “86.50” is assigned to union member “record1.percentage”. Now, memory location name is changed to “record1.percentage” with value “86.50”.
- Like this, name and value of union member is replaced every time on the common storage space.
- So, we can always access only one union member for which value is assigned at last. We can’t access other member values.
- So, only “record1.percentage” value is displayed in output. “record1.name” and “record1.subject” are empty.

Record2 union variable:

- If we want to access all member values using union, we have to access the member before assigning values to other members as shown in record2 union variable in this program.
- Each union members are accessed in record2 example immediately after assigning values to them.
- If we don't access them before assigning values to other member, member name and value will be over written by other member as all members are using same memory.
- We can't access all members in union at same time but structure can do that.

Example program – Another way of declaring C union:

In this program, union variable “record” is declared while declaring union itself as shown in the below program.

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

union student
{
    char name[20];
    char subject[20];
    float percentage;
}record;

int main()
{
    strcpy(record.name, "Raju");
    strcpy(record.subject, "Maths");
    record.percentage = 86.50;

    printf(" Name      : %s \n", record.name);
    printf(" Subject   : %s \n", record.subject);
    printf(" Percentage : %f \n", record.percentage);
    return 0;
}
```

Output:

Name :
Subject :
Percentage : 86.500000

We can access only one member of union at a time. We can't access all member values at the same time in union. But, structure can access all member values at the same time. This is because, Union allocates one common storage space for all its members. Whereas Structure allocates storage space for all its members separately.

Difference between structure and union in C:

S.no	C Structure	C Union
1	Structure allocates storage space for all its members separately.	Union allocates one common storage space for all its members. Union finds that which of its member needs high storage space over other members and allocates that much space
2	Structure occupies larger memory space.	Union occupies lower memory space over structure.
3	We can access all members of structure at a time.	We can access only one member of union at a time.
4	Structure example: struct student { int mark; double average; };	Union example: union student { int mark; double average; };
5	For above structure, memory allocation will be like below. int mark – 2B double average – 8B Total memory allocation = 2+8 = 10 Bytes	For above union, only 8 bytes of memory will be allocated since double data type will occupy maximum space of memory over other data types. Total memory allocation = 8 Bytes

POINTERS & DYNAMIC MEMORY ALLOCATION FUNCTIONS:

When a variable is defined the compiler (linker/loader actually) allocates a real memory address for the variable.

- int x; will allocate 4 bytes in the main memory, which will be used to store an integer value.

When a value is assigned to a variable, the value is actually placed to the memory that was allocated.

- x=3; will store integer 3 in the 4 bytes of memory.

The process of allocating memory during program execution is called dynamic memory allocation.

C language offers 4 dynamic memory allocation functions. They are,

1. `malloc()` : `malloc (number * sizeof(int));`
2. `calloc()` : `calloc (number, sizeof(int));`
3. `realloc()` : `realloc (pointer_name, number * sizeof(int));`
4. `free()` : `free (pointer_name);`

1. MALLOC():

- is used to allocate space in memory during the execution of the program.
- does not initialize the memory allocated during execution. It carries garbage value.
- returns null pointer if it couldn't able to allocate requested amount of memory.

2. CALLOC():

- `calloc ()` function is also like `malloc ()` function. But `calloc ()` initializes the allocated memory to zero. But, `malloc()` doesn't.

3. REALLOC():

- `Realloc ()` function modifies the allocated memory size by `malloc ()` and `calloc ()` functions to new size.
- If enough space doesn't exist in memory of current block to extend, new block is allocated for the full size of reallocation, then copies the existing data to new block and then frees the old block.

4. FREE():

- `free ()` function frees the allocated memory by `malloc ()`, `calloc ()`, `realloc ()` functions and returns the memory to the system.

DIFFERENCE BETWEEN STATIC MEMORY ALLOCATION AND DYNAMIC MEMORY ALLOCATION IN C:

Static memory allocation	Dynamic memory allocation
In static memory allocation, memory is allocated while writing the C program. Actually, user requested memory will be allocated at compile time.	In dynamic memory allocation, memory is allocated while executing the program. That means at run time.
Memory size can't be modified while execution. Example: array	Memory size can be modified while execution. Example: Linked list

DIFFERENCE BETWEEN MALLOC() AND CALLOC() FUNCTIONS IN C:

malloc()	calloc()
It allocates only single block of requested memory	It allocates multiple blocks of requested memory
<code>int *ptr; ptr = malloc(20 * sizeof(int));</code> For the above, 20*4 bytes of memory only allocated in one block. Total = 80 bytes	<code>int *ptr; Ptr = calloc(20, 20 * sizeof(int));</code> For the above, 20 blocks of memory will be created and each contains 20*4 bytes of memory. Total = 1600 bytes
malloc () doesn't initializes the allocated memory. It contains garbage values	calloc () initializes the allocated memory to zero
type cast must be done since this function returns void pointer <code>int *ptr; ptr = (int*)malloc(sizeof(int)*20);</code>	Same as malloc () function <code>int *ptr; ptr = (int*)calloc(20, 20 * sizeof(int));</code>

Array Operations: All operations remain same as mentioned above for data structures operations.

Note: → Refer 1st lab program for the operations.

SORTING:

Sorting takes an unordered collection and makes it an ordered one.

In bubble sort method the list is divided into two sub-lists sorted and unsorted. The smallest element is bubbled from unsorted sub-list. After moving the smallest element the imaginary wall moves one element ahead. The bubble sort was originally written to bubble up the highest element in the list. But there is no difference whether highest / lowest element is bubbled. This method is easy to understand but time consuming. In this type, two successive elements are compared and swapping is done. Thus, step-by-step entire array elements are checked. Given a list of 'n' elements the bubble sort requires up to n-1 passes to sort the data.

Algorithm for Bubble sort: Bubble_Sort(A[], N)

```
Step1 : Repeat for p = 1 to N-1
        Begin
Step2 :   Repeat for j = 1 to N-p
            Begin
Step3 :       if (A[j] < A[j-1])
                Swap ( A[j], A[j-1]);
            End for
        End for
Exit
```

Example:

Ex:- A list of unsorted elements are: 10 47 12 54 19 23 (Bubble up for highest value shown here)

10	54	54	54	54	54
47	10	47	47	47	47
12	47	10	23	23	23
54	12	23	10	19	19
19	23	12	19	10	12
23	19	19	12	12	10
Original List	After Pass 1	After Pass 2	After Pass 3	After Pass 4	After Pass 5

/* bubble sort implementation */

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int i,n,temp,j,arr[25];
    printf("Enter the number of elements in the Array: ");
    scanf("%d",&n);
    printf("\nEnter the elements:\n\n");
    for(i=0 ; i<n ; i++)
        scanf("%d",&arr[i]);

    for(i=0 ; i<n ; i++)
    {
        for(j=0 ; j<n-i-1 ; j++)
        {
            if(arr[j]>arr[j+1]) //Swapping Condition is Checked
            {
                temp=arr[j];
                arr[j]=arr[j+1];
                arr[j+1]=temp;
            }
        }
    }
    printf("\nThe Sorted Array is:\n\n");
```

```
for(i=0 ; i<n ; i++)  
    printf(" %4d",arr[i]);  
}
```

SEARCHING TECHNIQUE

1. LINEAR SEARCH

Linear search or sequential search is a method for finding a target value within a list. It sequentially checks each element of the list for the target value until a match is found or until all the elements have been searched.

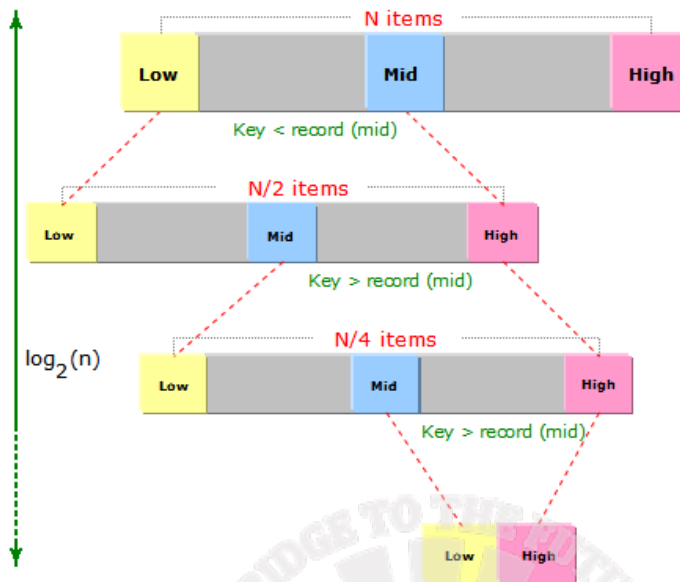
Linear search runs in at worst linear time and makes at most n comparisons, where n is the length of the list. If each element is equally likely to be searched, then linear search has an average case of $n/2$ comparisons, but the average case can be affected if the search probabilities for each element vary. Linear search is rarely practical because other search algorithms and schemes, such as the binary search algorithm and hash tables, allow significantly faster searching for all but short lists.

```
int linear_search(int *list, int size, int key, int* rec )  
{  
    // Basic Linear search  
    int found = 0;  
    int i;  
    for ( i = 0; i < size; i++ )  
    {  
        if ( key == list[i] )  
            found = 1;  
        return found;  
    }  
    Return found;  
}
```

2. BINARY SEARCH

We always get a sorted list before doing the binary search. Now suppose we have an ascending order record. At the time of search it takes the middle record/element, if the searching element is greater than middle element then the element must be located in the second part else it is in the first half. In this way this search algorithm divides the records in the two parts in each iteration and thus called binary search.

In binary search, we first compare the key with the item in the middle position of the array. If there's a match, we can return immediately. If the key is less than the middle key, then the item must lie in the lower half of the array; if it's greater then the item must lie in the upper half of the array. So we repeat the procedure on the lower or upper half of the array depending on the comparison.



```
int BinarySearch(int *array, int N, int key)
```

```
{
```

```
    int low = 0, high = N-1, mid;
```

```
    while(low <= high)
```

```
    {
```

```
        mid = (low + high)/2;
```

```
        if(array[mid] < key)
```

```
            low = mid + 1;
```

```
        else if(array[mid] == key)
```

```
            return mid;
```

```
        else if(array[mid] > key)
```

```
            high = mid-1;
```

```
    }
```

```
    return -1;
```

```
}
```

SPARSE MATRICES

A **sparse matrix** is a matrix in which most of the elements are zero. By contrast, if most of the elements are nonzero, then the matrix is considered **dense**. When storing and manipulating sparse matrices on a computer, it is beneficial and often necessary to use specialized algorithms and data structures that take advantage of the sparse structure of the matrix. Operations using standard dense-matrix structures and algorithms are slow and inefficient when applied to large sparse matrices as processing and memory are wasted on the zeroes. Sparse data is by nature more easily compressed and thus require significantly less storage. Some very large sparse matrices are infeasible to manipulate using standard dense-matrix algorithms.

Storing a sparse matrix

A matrix is typically stored as a two-dimensional array. Each entry in the array represents an element $a_{i,j}$ of the matrix and is accessed by the two indices i and j . Conventionally, i is the row index, numbered from top to bottom, and j is the column index, numbered from left to right. For an $m \times n$ matrix, the amount of memory required to store the matrix in this format is proportional to $m \times n$.

In the case of a sparse matrix, substantial memory requirement reductions can be realized by storing only the non-zero entries.

Sparse Matrix Representations

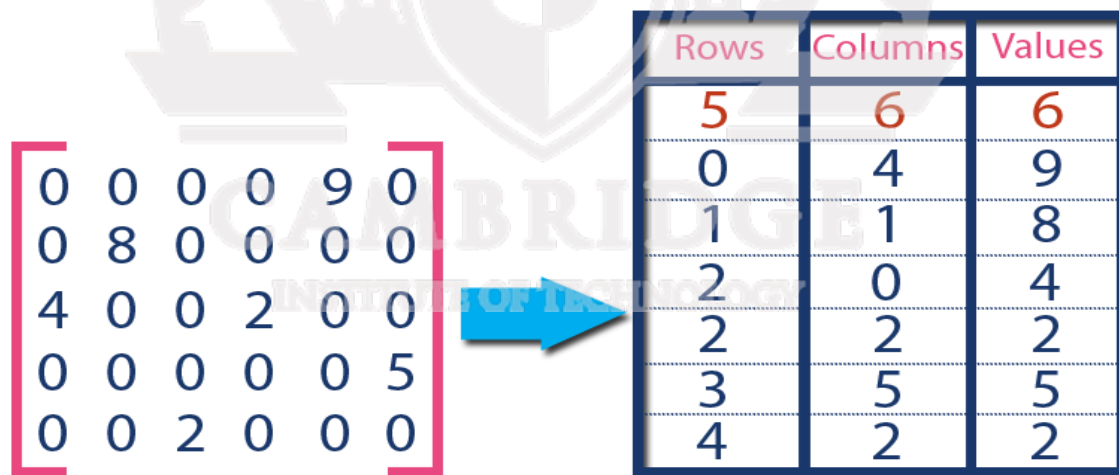
A sparse matrix can be represented by using TWO representations...

1. Triplet Representation
2. Linked Representation

1. Triplet Representation

In this representation, we consider only non-zero values along with their row and column index values. Each non zero value is a triplet of the form $\langle R, C, \text{Value} \rangle$ where R represents the row in which the value appears, C represents the column in which the value appears and Value represents the non-zero value itself. In this representation, the 0th row stores total rows, total columns and total non-zero values in the matrix.

For example, consider a matrix of size 5 X 6 containing 6 number of non-zero values. This matrix can be represented as shown in the image...



Rows	Columns	Values
5	6	6
0	4	9
1	1	8
2	0	4
2	2	2
3	5	5
4	2	2

In above example matrix, there are only 6 non-zero elements (those are 9, 8, 4, 2, 5 & 2) and matrix size is 5 X 6. We represent this matrix as shown in the above image. Here the first row in the right side table is filled with values 5, 6 & 6 which indicates that it is a sparse matrix with 5 rows, 6 columns & 6 non-zero values. Second row is filled with 0, 4, & 9 which indicates the value in the matrix at 0th row, 4th column is 9. In the same way the remaining non-zero values also follows the similar pattern.

2. Linked Representation

In linked representation, we use linked list data structure to represent a sparse matrix. In this linked list, we use two different nodes namely **header node** and **element node**. Header node consists of three fields and element node consists of five fields as shown in the image...

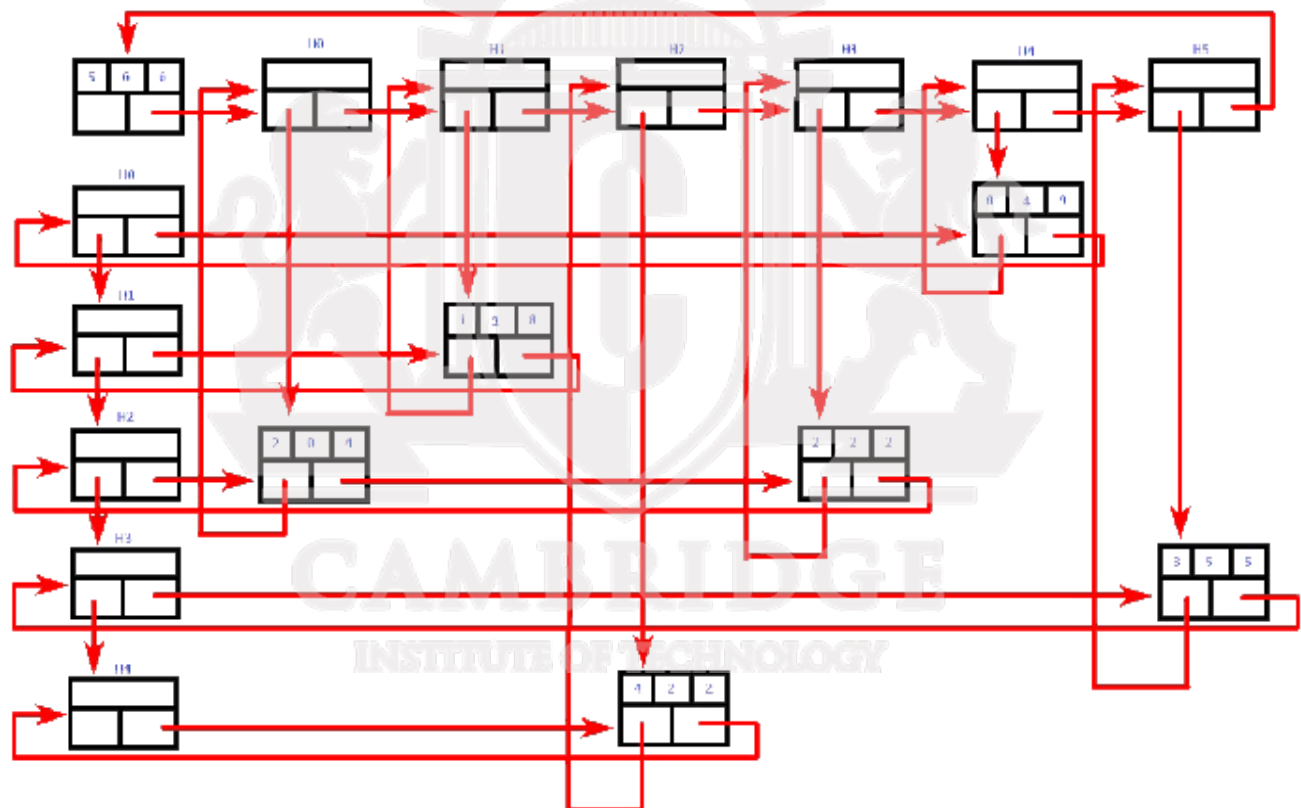
Header Node



Element Node



Consider the above same sparse matrix used in the Triplet representation. This sparse matrix can be represented using linked representation as shown in the below image...



In above representation, H0, H1,...,H5 indicates the header nodes which are used to represent indexes. Remaining nodes are used to represent non-zero elements in the matrix, except the very first node which is used to represent abstract information of the sparse matrix (i.e., It is a matrix of 5 X 6 with 6 non-zero elements).

In this representation, in each row and column, the last node right field points to it's respective header node.

Basic operations on Sparse Matrix

- Reading a sparse matrix
- Displaying a sparse matrix
- Searching for a non zero element in a sparse matrix

Note :Refer class notes for implementation of basic operations of sparse matrices

POLYNOMIALS

A *polynomial* object is a homogeneous ordered list of pairs $\langle \text{exponent}, \text{coefficient} \rangle$, where each coefficient is unique.

Operations include returning the degree, extracting the coefficient for a given exponent, addition, multiplication, evaluation for a given input

Polynomial operations

- Representation
- Addition
- Multiplication

Representation of a Polynomial: A polynomial is an expression that contains more than two terms. A term is made up of coefficient and exponent. An example of polynomial is

$$P(x) = 4x^3 + 6x^2 + 7x + 9$$

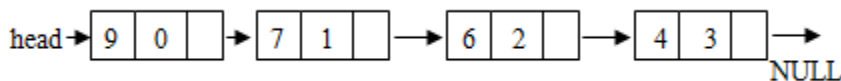
A polynomial thus may be represented using arrays or linked lists. Array representation assumes that the exponents of the given expression are arranged from 0 to the highest value (degree), which is represented by the subscript of the array beginning with 0. The coefficients of the respective exponent are placed at an appropriate index in the array. The array representation for the above polynomial expression is given below:

arr	9	7	6	4	(coefficients)
	0	1	2	3	(exponents)

A polynomial may also be represented using a linked list. A structure may be defined such that it contains two parts- one is the coefficient and second is the corresponding exponent. The structure definition may be given as shown below:

```
struct polynomial
{
int coefficient;
int exponent;
struct polynomial *next;
};
```

Thus the above polynomial may be represented using linked list as shown below:



Addition of two Polynomials:

For adding two polynomials using arrays is straightforward method, since both the arrays may be added up element wise beginning from 0 to n-1, resulting in addition of two polynomials. Addition of two polynomials using linked list requires comparing the exponents, and wherever the exponents are found to be same, the coefficients are added up. For terms with different exponents, the complete term is simply added to the result thereby making it a part of addition result. The complete program to add two polynomials is given in subsequent section

Refer notes for memory representation of polynomial

STRINGS

❖ Strings are Character Arrays

Strings in C are simply array of characters.

- Example: `char s[10];` This is a ten (10) element array that can hold a character string consisting of ≤ 9 characters. This is because C does not know where the end of an array is at run time. By convention, C uses a NULL character `'\0'` to terminate all strings in its library functions
- For example: `char str[10] = {'u', 'n', 'i', 'x', '\0'};`
It's the string terminator (not the size of the array) that determines the length of the string.

❖ Accessing Individual Characters

The first element of any array in C is at index 0. The second is at index 1, and so on...

`char s[10];`

`s[0] = 'h';`

`s[1] = 'i';`

`s[2] = '!';`

`s[3] = '\0';`

s [0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
H	i	!	\0						

This notation can be used in all kinds of statements and expressions in C:

For example:

`c = s[1];`

`if (s[0] == '-') ...`

`switch (s[1]) ...`

❖ String Literals

String literals are given as a string quoted by double quotes.

- `printf("Long long ago.");`

Initializing char array ...

- `char s[10]="unix"; /* s[4] is '\0'; */`
- `char s[]="unix"; /* s has five elements */`
- Printing with `printf()`

Example:

```
Char    str[    ]    =    "A    message    to    display";  
printf ("%s\n", str);
```

`printf` expects to receive a string as an additional parameter when it sees `%s` in the format string

- Can be from a character array.
- Can be another literal string.
- Can be from a character pointer (more on this later).
- `printf` knows how much to print out because of the NULL character at the end of all strings.
- When it finds a `\0`, it knows to stop.

❖ Printing with `puts()`

The `puts` function is a much simpler output function than `printf` for string printing.

- Prototype of `puts` is defined in `stdio.h`
`int puts(const char * str)`. This is more efficient than `printf` because your program doesn't need to analyze the format string at run-time.

For example:

```
char sentence[] = "The quick brown fox\n";  
puts(sentence);
```

Prints out: The quick brown fox

▪ Inputting Strings with `gets()`

`gets()` gets a line from the standard input.

The prototype is defined in `stdio.h`

```
char *gets(char *str)
```

- `str` is a pointer to the space where `gets` will store the line to, or a character array.
- Returns NULL upon failure. Otherwise, it returns `str`.

```
char your_line[100];  
printf("Enter a line:\n");  
gets(your_line);  
puts("Your input follows:\n");  
puts(your_line);
```

You can overflow your string buffer, so be careful!

▪ **Inputting Strings with scanf ()**

To read a string include:

- %s scans up to but not including the “next” white space character
- %ns scans the next n characters or up to the next white space character, whichever comes first

Example:

```
scanf ("%s%s%s", s1, s2, s3);
```

```
scanf ("%2s%2s%2s", s1, s2, s3);
```

- Note: No ampersand(&) when inputting strings into character arrays! (We’ll explain why later ...)

Difference between gets

- gets() read a line
- scanf("%s",...) read up to the next space

Example:

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

```
int main () {
```

```
    char lname[81], fname[81];
```

```
    int count, id_num;
```

```
    puts ("Enter the last name, firstname, ID number separated");
```

```
    puts ("by spaces, then press Enter \n");
```

```
    count = scanf ("%s%s%d", lname, fname,&id_num);
```

```
    printf ("%d items entered: %s %s %d\n",  
           count,fname,lname,id_num);
```

```
    return 0;
```

```
}
```

▪ **The C String Library**

String functions are provided in an ANSI standard string library.

- Access this through the include file:

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

- Includes functions such as:
 - Computing length of string
 - Copying strings
 - Concatenating strings

This library is guaranteed to be there in any ANSI standard implementation of C.

- strlen() returns the length of a NULL terminated character string:

```
strlen (char * str) ; Defined in string.h
```

A type defined in string.h that is equivalent to an unsigned int

char *str

Points to a series of characters or is a character array ending with '\0'

- strcpy() copying a string comes in the form:

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

A copy of source is made at destination. Source should be NULL terminated and destination should have enough room (its length should be at least the size of source). The return value also points at the destination.

- strcat() comes in the form:

char * strcat (char * str1, char * str2);

Appends a copy of str2 to the end of str1 & a pointer equal to str1 is returned.

Ensure that str1 has sufficient space for the concatenated string!

Array index out of range will be the most popular bug in your C programming career.

Example:

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

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

```
int main() {
```

```
    char str1[27] = "abc";
```

```
    char str2[100];
```

```
    printf("%d\n",strlen(str1));
```

```
    strcpy(str2,str1);
```

```
    puts(str2);
```

```
    puts("\n");
```

```
    strcat(str2,str1);
```

```
    puts(str2);
```

```
}
```

- strcmp() can be compared for equality or inequality

If they are equal - they are ASCII identical

If they are unequal the comparison function will return an int that is interpreted as:

< 0 : str1 is less than str2

0 : str1 is equal to str2

> 0 : str1 is greater than str2

Four basic comparison functions:

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

- ❖ Does an ASCII comparison one char at a time until a difference is found between two chars

- Return value is as stated before

❖ If both strings reach a '\0' at the same time, they are considered equal.

```
int strncmp (char *str1, char * str2, size_t n);
```

- ❖ Compares n chars of str1 and str2
 - Continues until n chars are compared or
 - The end of str1 or str2 is encountered

Example:

```
int main() {  
    char str1[] = "The first string."  
    char str2[] = "The second string."  
    size_t n, x;  
    printf("%d\n", strncmp(str1, str2, 4) );  
    printf("%d\n", strncmp(str1, str2, 5) );  
}
```

Note: Refer lab program 2 for implementation

