USN | | | | | | | | | | |

Internal Assessment Test 1 – Sept. 2017

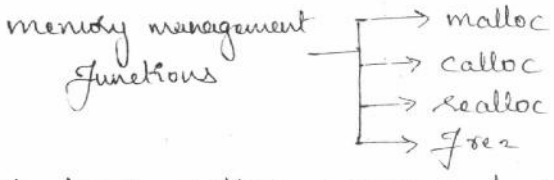
Sub:	DATA STRUCTURES AND APPLICATIONS					Sub Code:	15CS33	Branch:	CSE	
Date:	20/09/2017	Duration:	90 min's	Max Marks:	50	Sem / Sec:	3(C)		OBE	

Answer any FIVE FULL Questions

		MARKS	CO	RBT
1 (a)	Define Dynamic Memory Allocation? Explain the dynamic allocation functions with syntax and example?	[10]	CO2	L1
2 (b)	Write KMP Pattern Matching Algorithm. Apply the Same to search pattern 'abcdabcy' in text 'abcxabcdabxabcdabcdabcy'	[10]	CO6	L3
3 (a)	Write a function to sort integers using Bubble sort algorithm.	[05]	CO5	L2
(b)	Define Data Structure? Explain the Classification and operations.	[05]	CO1	L1
4 (a)	Write an algorithm to implement stack using dynamic array whose initial capacity is 1 and array doubling is used to increase the stack capacity (reallocation) whenever an element is added to a full stack.	[10]	CO3	L3
5 (a)	Define String? Explain the following a) strtok b) strstr	[05]	CO1	L1
(b)	Write a recursive function for Tower of Hanoi and Ackerman's function	[05]	CO3	L2
6 (a)	Write function to convert infix to postfix. Apply the same for the following expression (A+(B*C)/D^E)	[10]	CO5	L3
7 (a)	Define Sparse Matrix? Explain triplets and transpose of sparse Matrix with an example	[05]	CO5	L3
(b)	Differentiate between Structure and union	[05]	CO1	L2

Define Dynamic memory allocation? Explain the dynamic memory allocation functions?

* The memory management functions that are used to allocate & dealloc memory are shown below



* Using the functions malloc, calloc and realloc memory space can be allocated where as using the function free() space can be deleted.

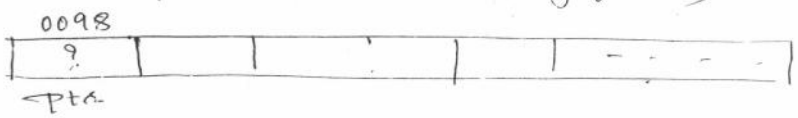
① malloc()

* Function allows the program to allocate memory explicitly as & when required and the exact amount needed during execution.
* This function allocates a block of memory. The size of the block is number of bytes specified in the parameter.
* It is defined in stdlib.h header file. Syntax is

```
#include<stdlib.h>
        :
ptr = (data type *) malloc(size);
```

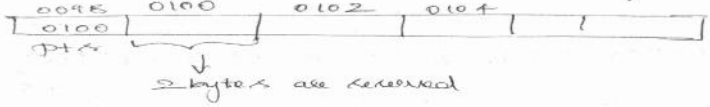
Where ptr is a pointer variable of type datatype
datatype can be any of the basic data type
Size is the number of bytes required.

For example: int *ptr;
ptr = (int *) malloc (sizeof(int));



ptr = (int *) malloc (sizeof(one));

Since 2 bytes of free memory space is available, the function malloc allocates a block of 2 bytes of memory and returns the address of the first byte.
This returned address is stored in the pointer



- Calloc

* Calloc stands for contiguous allocation of multiple blocks and it mainly used to allocate memory for arrays.
* The number of blocks determined by n. The size of each block is equal to the number of bytes specified in the parameter i.e size.
* The total number of bytes allocated is n * size and all bytes will be initialized to 0.
* The syntax:

```
#include<stdlib.h>
.....
ptr = (data type *) calloc (n, size);
```

where

* ptr is pointer variable of type data-type
* data-type can be any of the basic data type or user defined data type.
* n is the number of blocks to be allocated
* size is the number of bytes in each block.

* On successful allocation, the function return the address of first byte of allocated memory. Since the address is returned, the return type is a void pointer, By type casting appropriately we can use it to store integer float etc.
* If specified size of memory is not available, the condition is called overflow of memory. In such case, the function returns NULL.

For example: `int *ptr;`

`ptr = (int *) calloc( 5, sizeof(int));`

free

* This function is used to de-allocate the allocated blocks of memory which is allocated by using the functions calloc(), malloc() & realloc().
* It is responsibility of a programmer to de-allocate memory whenever it is not required by the program and initialize ptr to NULL

realloc

* Sometimes, the allocated memory may not be sufficient and we may require additional memory space. In another situation, where allocated memory may be much larger and we want to reduce.
* In both the situations, the size of allocated memory can be changed using realloc() and process is called reallocation of memory.
* realloc() changes the size of the block by extending or deleting the memory at the end of the block
* If the existing memory can be extended, ptr value will not be changed.
* If the memory can not be changed extended, this function allocates completely new block.

Syntax: `#include<stdlib.h>`

`ptr = (datatype *) realloc(ptr, size);`

where

ptr is a pointer to block of memory which is allocated previously

size is new size of block

* On successful allocation, the function returns the address of first byte of allocated memory.
* If memory can not be allocated, function returns NULL

2. KMP algorithm

* This algorithm was conceived by Donald Knuth and Vaugh Pratt and James Morris in 1977.
* In the naive or Brute force approach, wasted time in comparision on mismatch. To overcome KMP algorithm use longest prefix in a Suffix concept.
* This LPS will reduce the number of comparision

$O(m \cdot n)$ to $O(m+n)$

where $n \Rightarrow$ length of String

$m \Rightarrow$ length of Pattern String.

How to find LPS

* While finding LPS, ~~if~~ failure function preprocesses the pattern to find matches of prefixed of the pattern with itself.
* It is defined as size of the largest prefix $P[0 \ldots j-1]$ that is also a suffix of $P[1 \ldots j]$
* It also indicates how much of the last comparision can be reused if it fails.

Algorithm computeLPS (pat, M, lps)

Input: pat is a pattern String
M is length of pattern String
lps is an array which we need to compute

output: lps is an array

```
lps[0]=0;  // initialize lps[0] to zero
// The loop calculates for i=1 to M-1
int i=1; int j=0;
while (i<M)
{
    if (pat[i]==pat[j]) // if the characters are matching increment j and assign lps with the value of j
    {
        j++;
        lps[i]=j;
        i++;
    }
    else // if the characters are not matching go back and compare with j and i index
    {
        if (j!=0)
        {
            j=lps[j-1];
        }
        else
        {
            lps[i]=0;
            i++;
        }
    }
}
```

```
Algorithm KMPSearch(char Sts, Pts, M, N)
{
    Sts ⇒ main string
    Pts ⇒ Pattern string
    M ⇒ Length of the pattern
    N ⇒ Length of the string
int LPS[];
ComputeLPS(Pts, LPS, M);

int i=0;
int j=0;
while (i<N)
{
        if (pat[j]==Sts[i])
        {
            j++;
            i++;
        }
```

```
    if(j==M)
    {
      printf("pattern found at index %d", i-
      j= LPS[j-1];
    }
    else if (i<N && pat[j]!=sts[i])
    {
        if(j!=0)
            j=lps[j-1];
        else
            i=i+1
    }
}
}
```

String:

→ abcxabcdabxabcdabcdabcy

Pattern abcdabcy

a	b	c	d	a	b	c	y
0	0	0	0	1	2	3	0

① abcxabcdabxabcdabcdabcy
abcdabcy

② abcxabcdabxabcdabcdabcy
no match
abcdabcy

③ abcxabcdabxabcdabcdabcy
abcdabcy
prefix Suffix

Since we have prefix of same Suffix compare

Compare x and c ⇒ no match. Start com from the begining of the pattern ⇒ i.e a to x

abcxabcdabxabcdabcdabcy
no match
abcdabcy

abcxabcdabxabcdabcdabcy
abcdabcy
prefix Suffix
compare from d

abcxabcdabxabcdabcdabcy
abcdabcy
prefix

Since Suffix & same prefix no need of compasing abc. Comparision starts from d.

3a) BUBBLE sort

```
for(i=0; i<n; i++)
{
    for(j=0; j<n-i-1; j++)
    {
        if (a[j] > a[j+1])
        {
            temp = a[j];
            a[j] = a[j+1];
            a[j+1] = temp;
        }
    }
}
printf("The sorted array using Bubble
        Sort is :\n");
for(i=0; i<n; i++)
{
    printf("%d\t", a[i]);
}
```

3b) data structure and its classifications

Data Structure is way of Organizing data in a memory so that it can be used efficiently

or

Data Structure is specialized format for Organizing, Storing and retrieving the data.

Types of Data Structure

* Data Structure can be classified as follows
  1) Primitive Data Structure
  2) Non-Primitive Data Structure
  3) Linear Data Structure
  4) Non-Linear Data Structure

(a) Primitive Data-type or Data Structure

* Basic data types that are available in most of the Programming Language. Data structures that are directly operated upon by machine-level instructions are known as primitive data types
* for example
  1) Integer (int): used to represent a number without decimal point Eg: 12, 90
  2) Float or double (float): used to represent a number with decimal point Eg: 12.50
  3) Character (char): used to represent single character Eg: 'c', 'a'
  4) Boolean: used to represent logical values i.e true or false

Non-Primitive Data Structure

* The data structures are derived from the primitive data structures. They stress on formation of sets of homogeneous and heterogeneous data elements.

Example: Arrays, Stack, Queue, Linked List, Tree, Graph

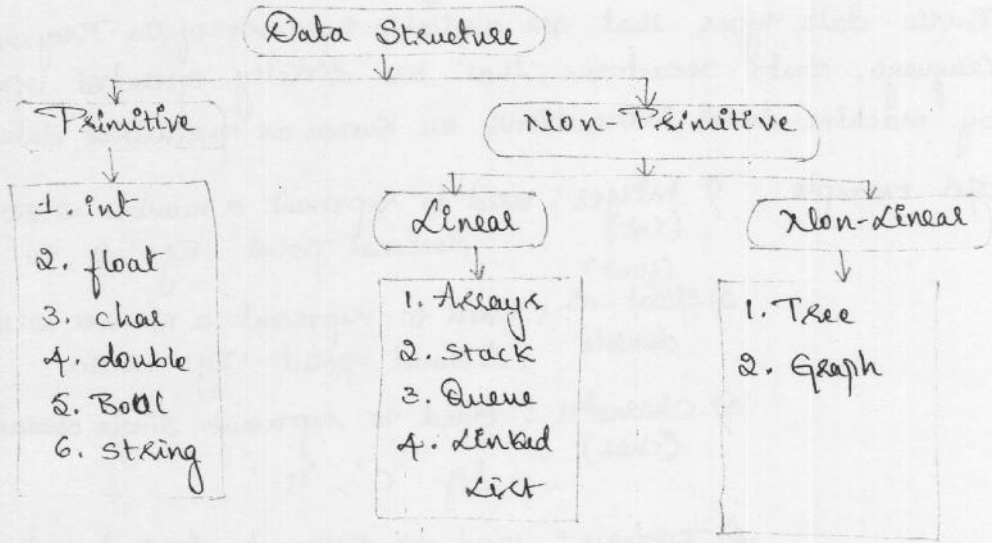
Linear-Data Structure

* In Linear Data Structure, data is arranged in linear fashion. Eg: Arrays, Stack Queue, Linked List

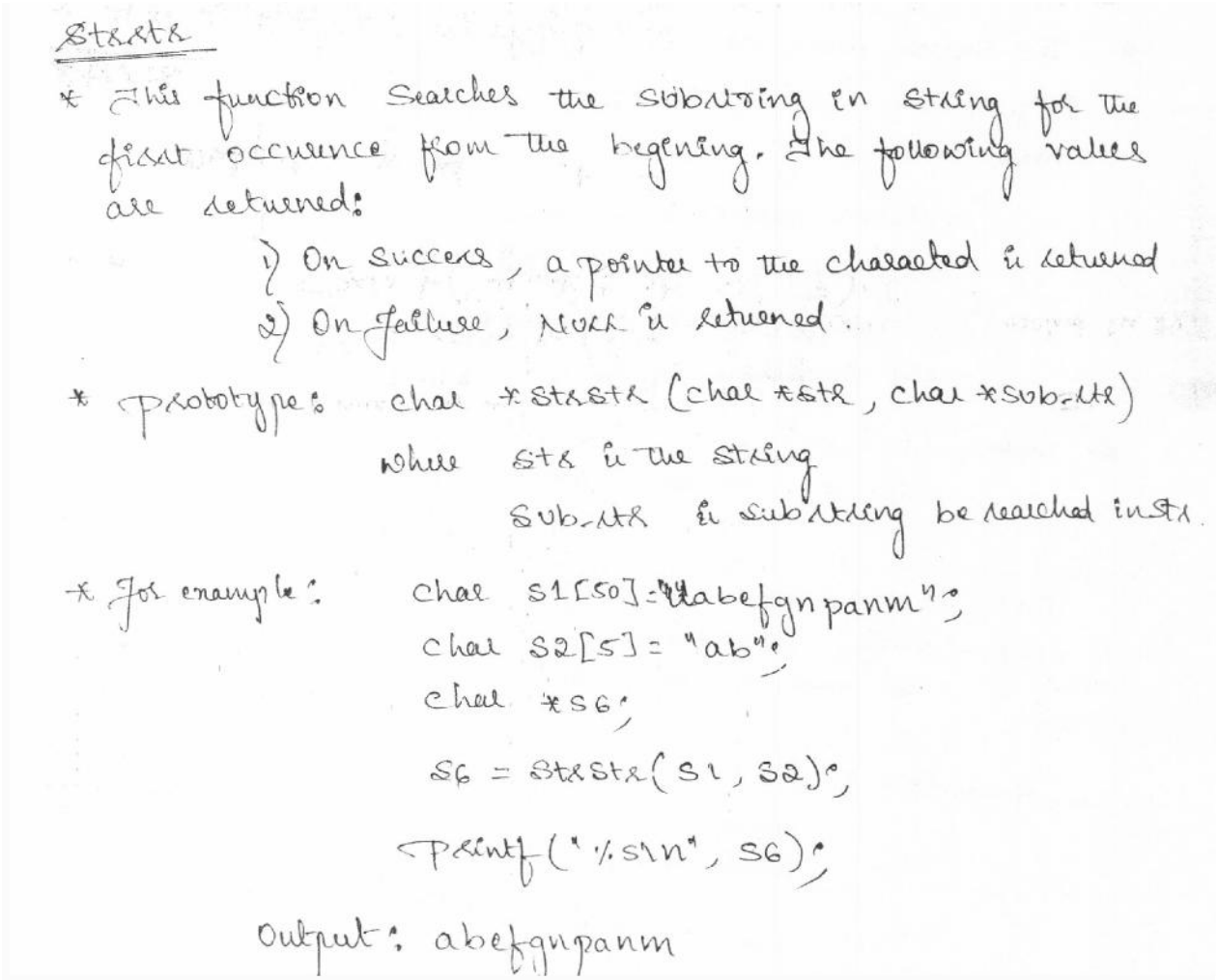
Non-Linear Data Structure

* In Non Linear Data Structure, data is not arranged in order or linear fashion

Eg: Trees, Graph, Table etc.



5a) define string and explain the following a)strtok and b)stsstr



strtok() function

strtok( ) function in C tokenizes/parses the given string using delimiter. Syntax for strtok( ) function is given below.

char * strtok (char * str, const char * delimiters);

EXAMPLE PROGRAM FOR STRTOK() FUNCTION IN C:

In this program, input string “Test,string1,Test,string2:Test:string3” is parsed using strtok() function. Delimiter comma (,) is used to separate each sub strings from input string.

```
#include <stdio.h>
#include <string.h>
int main ()
{
        char string[50] ="Test,string1,Test,string2:Test:string3";
```

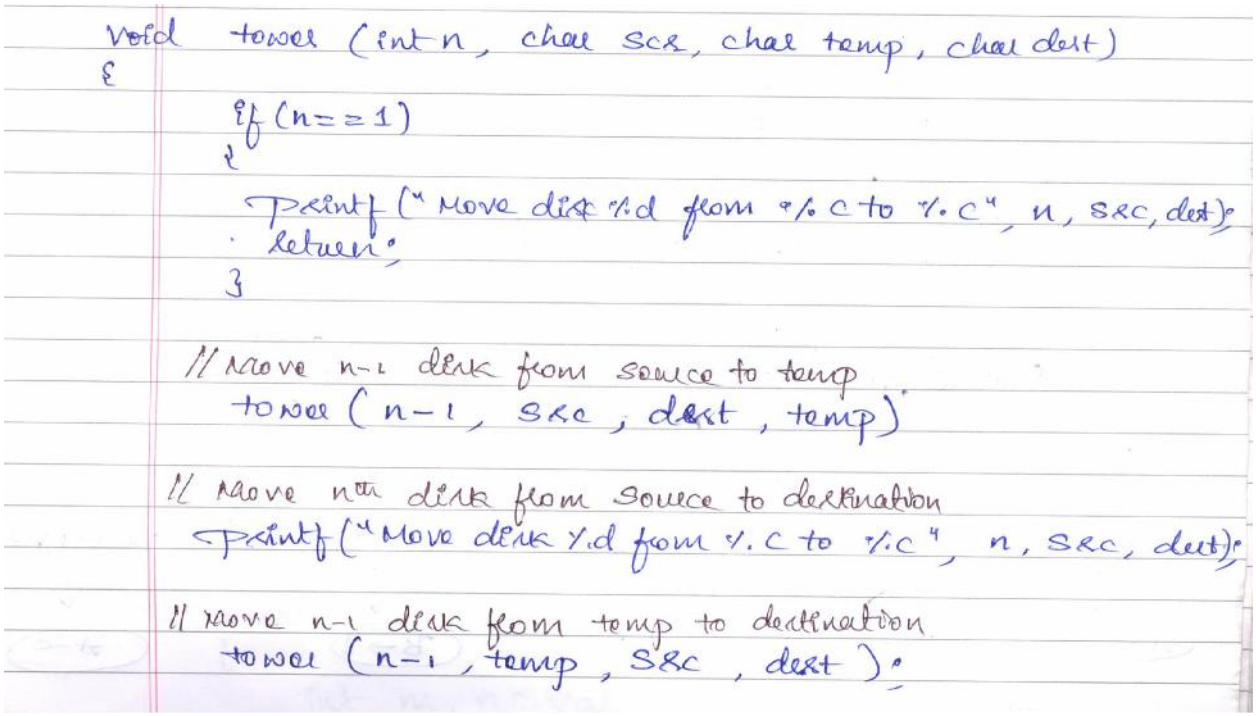
```
        char *p;
         printf ("String  \"%s\" is split into tokens:\n",string);
        p = strtok (string,",:");

        while (p!= NULL)
        {
                printf ("%s\n",p);
                p = strtok (NULL, ",:");
        }

        return 0;
}
```

String “Test,string1,Test,string2:Test:string3” is split into tokens:
Test
string1
Test
string2
Test
string3

5b) Tower of Hanoi and Ackerman function



```
void tower (int n, char src, char temp, char dest)
{
    if (n==1)
    {
     printf ("Move disk %d from %c to %c", n, src, dest);
     return;
    }

    // Move n-1 disk from source to temp
     tower (n-1, src, dest, temp);

    // Move nth disk from source to destination
     printf ("Move disk %d from %c to %c", n, src, dest);

    // Move n-1 disk from temp to destination
     tower (n-1, temp, src, dest);
```

```
int Ack(int m, int n)
{
    if (m==0)
        return (n+1);
    else if ((m>0) && (n==0))
        return (Ack(m-1, 1));
    else
        return Ack(m-1, ack(m, n-1));
}
```

6a) infix to postfix conversion

```
int priority (char c)
{
    if (c == '#')
        return 0;
    else if (c == '(')
        return 1;
    else if (c == '+' || c == '-')
        return 2;
    else if (c == '*' || c == '/' || c == '%')
        return 3;
    else
        return 4;
}

int main()
{
    char infix[MAX], postfix[MAX];
    int i, j = 0;

    printf ("enter the infix expression \n");
    scanf ("%s", infix);

    push ('#');

    for (i=0; infix[i] != '\0'; i++)
    {   // if its alphanumeric place it in ~~numeric~~ postfix
        if (isalnum (infix[i]))
            postfix[j++] = infix[i];
        // check if the character is '(', then push into stack
        else if (infix[i] == '(')
            push (infix[i]);
        // if the character is ')', then pop the characters from
        else if (infix[i] == ')')   stack until, top of stack
                                    is '('
```

```
        while (stack[top] != '(')
        {
            postfix[j++] = pop();
        }
        top--;
    }
    else
    { // check the priority of character
        while(priority(stack[top]) >= priority(infix[i]))
        {
            postfix[j++] = pop();
        }
        push(infix[i]);
    }
}

while(stack[top] != '#')
    postfix[j++] = pop();

postfix[j++] = '\0';

printf("\n The postfix expression is %s", postfix);

}
```

infix expression: ((A+(B-C)*D)^E+F)

Stack	top of the stack	Symbol	Postfix	Operation
#	#	(		Push into stack
# (	(	(		Push into stack
# ((	(	A	A	Place it in postfix
# ((	(	+	A	+ has higher precedence. Push into stack
# ((+	+	(	A	Push into stack
# ((+ (	(	B	A B	Place it in postfix
# ((+ (	(	-	A B	- has higher precedence push into stack
# ((+ (-	-	C	A B C	Push into stack
# ((+ (-)	-	)	A B C -	Pop out until we get matching '('
# ((+	+	*	A B C -	* has higher precedence than + push
# ((+ *	*	D	A B C - D	Place it in postfix
# ((+ *)	*	)	A B C - D * +	Pop until we get matching '('
# (	(	^	A B C - D * +	Push into stack
# (^	^	E	A B C - D * + E	Place it in postfix
# (^	^	+	A B C - D * + E ^	+ has less precedence than ^, pop out ^ from stack placed it in postfix

7 b) Differentiate between union and structure

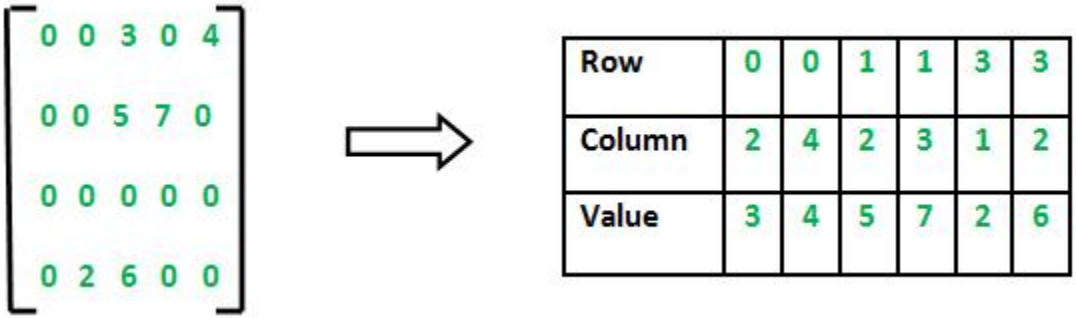
Difference Between Structure and Union

STRUCTURE	UNION
1) Keyword struct is used to define structure	1) Keyword union is used to define a union
2) When a variable is associated with a structure the compiler allocates memory to each of the variable.	2) When a variable associated with a union the compiler allocates memory by considering the size of the largest memory.
3) The size of structure will be greater than or equal to the sum of sizes of its members	3) The size of union will be equal to size of largest member.
4) Each member with in a structure is assigned unique storage area of locations	4) Memory allocated is shared by individual members.
5) The address of each member will be in ascending order. This indicates that memory for each member will start at different offset values.	5) The address is same for all the members of a union. every member begins at different offset values.
6) Individual member can be accessed at a time	6) Only one member can be accessed

A matrix is a two-dimensional data object made of m rows and n columns, therefore having total m x n values. If most of the elements of the matrix have **0 value**, then it is called a sparse matrix.

Why to use Sparse Matrix instead of simple matrix ?

- **Storage:** There are lesser non-zero elements than zeros and thus lesser memory can be used to store only those elements.
- **Computing time:** Computing time can be saved by logically designing a data structure traversing only non-zero elements..



Representing a sparse matrix by a 2D array leads to wastage of lots of memory as zeroes in the matrix are of no use in most of the cases. So, instead of storing zeroes with non-zero elements, we only store non-zero elements. This means storing non-zero elements with **triples- (Row, Column, value).**

Sparse Matrix Representations can be done in many ways following are two common representations:

1. Array representation
2. Linked list representation

Method 1: Using Arrays

2D array is used to represent a sparse matrix in which there are three rows named as

- **Row:** Index of row, where non-zero element is located
- **Column:** Index of column, where non-zero element is located
- **Value:** Value of the non zero element located at index – (row,column)