SECTION 1 DATA AND FILE STRUCTURES LAB MANUAL

1.0 INTRODUCTION

A data structure is one of the most important courses that is to be mastered by any software professional. They are the building blocks of a program. The life of a house depends on the strength of its pillars. Similarly, the life of a program depends on the strength of the data structures used in the program. It is very important to write an algorithm before writing a program. This will enable you to focus more on the semantics of the program rather than syntax. Also, it will lead you to write efficient programs.

In this section, a brief introduction to the data structures that are discussed in the corresponding course (MCS-021) is given. Each introduction is followed by a list of programming problems. Wherever, a particular program for a given programming problem was already given in the course material, it is suggested that you should not copy it. Rather, you should develop your own logic to write the program.

To successfully complete this section, the learner must have already:

- Thoroughly studied the course material of MCS-011,
- Thoroughly studied the Section-1(C Programming Manual) of MCSL-017, and
- Executed most of the programming problems of the Lab assignments.

Also, to successfully complete this section, the learner should adhere to the following:

- Before attending the lab session, the learner must have already written algorithms and programs in his/her lab record. This activity should be treated as home work that is to be done before attending the lab session.
- The learner must have already thoroughly studied the corresponding units of the course material (MCS-021) before attempting to write algorithms and programs for the programming problems given in a particular lab session.
- Ensure that you include comments in your program. This is a practice, which will enable others to understand your program and enable you to understand the program written by you after a long time.
- Ensure that the lab record includes **algorithms, programs, I/O and complexity (both time and space)** of each program.

1.1 OBJECTIVES

After following the instructions in this section, you should be able to

- Write programs using basic data structures such as Arrays etc. as well as advanced data structures such as trees etc.

1.2 ARRAYS

An Array is a collection of element(s) of the same data type. They are referred through an index along with a name.

Consider the following examples:

int a[5];

The above is a declaration of an array which is a collection of 5 elements of integer data type. Since the collection consists of 5 elements, the index starts from 0(zero) and ends at 4(four). Of course, it is very important to remember that, in 'C' language, the index always starts at 0 and ends at (length of array-1). The *length of array* is nothing but the number of maximum elements that can reside in an array. In our example, the length is 5. The first element is referred as a[0], the second element is referred to as a[1] . The third, fourth and fifth elements are referred to as a[2], a[3] and a[4].

The example given above is a single dimensional array. We can declare arrays of any dimension.

Consider the following array:

int b[5][6];

This is a two-dimensional array. It can hold a maximum of 30(5X6) elements. The first element is referred to as b[0][0], the second as b[0][1] and so on. The last element is referred to as b[4][5].

It is always possible to declare arrays of elements of any type. It is possible to declare an array of characters, structures, etc.

For more information on arrays, Refer to Unit-6 of MCS-011, Block-2.

Session 1: Arrays (3 hours)

1. Write a program in `C' language that accepts two matrices as input and prints their product.

   Input: Two matrices A and B in two-dimensional arrays
   Output: Matrix `C' which is result of A X B in the form of two dimensional array

2. Write a program in `C' Language to accept 10 strings as input and print them in lexicographic order

   Input: 10 strings (use array of strings)
   Output: 10 input strings in lexicographic order

3. Write a program in `C' Language that accepts two strings S1 and S2 as input. The program should check if S2 is a substring of S1 or not. If S2 is a substring of S1, then the program should output the starting location and ending location

of S2 in S1. If S2 appears more than once in S1, then the locations of all instances have to be given.

4. Write a program to concatenate two strings S1 and S2.

1.3 STRUCTURES

A Structure is a collection of element(s) of one or more data types. The value of each element of the structure is referred through its corresponding identifier coupled with the variable name of the structure.

Consider the following example:

```
struct  student   {
char name[25];
char  *address;
int   telephone[8];
};
```

In the above example, the name of the structure is student. There are three elements in the structure (of course, you can declare any number of elements in the structure). They are name, address and telephone number. The first element is an array of characters, the second is address which is a pointer to a string or sequence of characters and the third is an array of integers. It is also permissible to have nested structures. That is, a structure inside a structure.

Consider the following declaration:

```
struct student  s;
```

s is a variable of type **student**.

The values of the elements in the above given example can be referred as s.name, s.address and s.telephone.

For more information on Structures, refer to unit-9 of MCS-011, Block, 3

Session 2: Structures (3 hours)

1. Write a program in 'C' language, which accepts Enrolment number, Name Aggregate marks secured in a Program by a student. Assign ranks to students according to the marks secured. Rank-1 should be awarded to the students who secured the highest marks and so on. The program should print the enrolment number, name of the student and the rank secured in ascending order.
2. Write a program in 'C' language to multiply two sparse matrices.
3. Write a program in 'C' language to accept a paragraph of text as input. Make a list of words and the number of occurrences of each word in the paragraph as output. As part of the processing, an array and structure should be created wherein each structure consists of two fields, namely, one for storing the word and the other for storing the number of occurrences of that word.

1.4 LINKED LISTS

A linked list is a sequence of zero or more number of elements that are connected directly or indirectly through pointers.

A linked list can be a singly linked list or a doubly linked list. Again, a singly linked

list or a doubly linked list can also be a circularly linked list.
For more information on linked lists, refer to unit-3 of MCS-021.
For more information on pointers, refer to unit-10 of MCS-011, Block-3.

Session 3: Linked Lists (3 hours)

1. Write a program in ‘C’ language for the creation of a list. Also, write a procedure for deletion of an element from the list. Use pointers.

   You have to write the above program separately for Singly linked list, Doubly linked list and Circularly linked lists (both singly linked and doubly linked). Make necessary assumptions.

2. Write a program in ‘C’ language that accepts two singly linked lists A and B as input. Now, print a singly linked list that consists of only those elements, which are common to both A and B.

3. Write a program in ‘C’ language to accept a singly linked list of integers as input. Now, sort the elements of the list in ascending order. Then, accept an integer as input. Insert this integer into the singly linked list at the appropriate position.

1.5 STACKS

A Stack is a LIFO (Last In First Out) data structure. A stack can be implemented using arrays or pointers. But, the disadvantages of using arrays is that the maximum number of elements that can be stored are limited. This disadvantage can be overcome by using pointers.

There are two important operations associated with a stack. They are **push** and **pop**. A **push** will add an element to the stack and a **pop** will delete (either really by freeing the memory location or by reducing the counter indicating the number of elements in the stack by one).

For more information on stacks, refer to unit-4 of MCS-021.

Session 4: Stacks (3 hours)

1. Write a program in ‘C’ language to convert a prefix expression to a postfix expression using pointers.

2. Write a program in ‘C’ language to reverse an input string.

3. Write a program in ‘C’ language to implement multiple stacks in a single array.

1.6 QUEUES

A Queue is a FIFO (First In First Out) data structure. A Queue can be implemented using arrays or pointers. The same disadvantages that are associated with Stacks hold true for Queues. In addition, in the case of Queues, whenever the element at front is deleted, all the elements are to be moved one position forward which leaded to time overhead. Else, there will be a waste of memory. With the help of pointers, these disadvantages can be overcome.

There are two important operations associated with a Queue. They are **Add** and **Delete**. An Add operation will add an element to the end (or rear) of the queue. A Delete operation will delete an element from the front of the queue. One simple way of understanding these operations is to remember that the order of deletion of elements from the queue will be exactly the same as the order of addition of elements to the Queue. You can draw an analogy of Queue data structure to the people standing in a queue for entering a bus etc. Whoever is at the front will get the opportunity to enter the bus first. Any person who is not in the queue and intends to enter the bus should stand in the queue at the rear/last.

A Dequeue is a special form of queue in which addition/deletion of elements is possible to be done at the front as well as at the rear. So, the concept of FIFO does not hold here.

For more information on Queues, refer to unit-5 of MCS-021.

Session 5: Queues (3 hours)

1. Write a program in 'C' language to implement a Dequeue using Arrays. All operations associated with a Dequeue are to be implemented.

2. Write a program in 'C' language to implement a Dequeue using pointers. All operations associated with a Dequeue are to be implemented.

3. Write a program in 'C' language to reverse the elements of a queue.

4. Write a program in 'C' language to implement a queue using two stacks.

5. Write a program in 'C' language to implement a stack using two queues.

1.7 TREES

Often, there is confusion about the differences between a tree and a binary tree. The major difference is that a Tree is always non-empty. It means that there is at least one element in it. It obviously leads to the conclusion that a Tree always had a root. The existence of the remaining elements is optional. It is possible for a Binary tree to be empty. For a binary tree, the number of children of a node cannot be more than 2. There is no restriction on the number of children to the nodes of a tree.

For more information on Trees and Binary trees, refer to unit-6 of MCS-21.

Session 6: Trees and Binary Trees (3 hours)

1. Write a program in 'C' language for the creation of a binary tree. Also, provide for insertion and deletion operations.

2. Write a program in 'C' language for pre-order, post-order and in-order tranversals of a Binary tree. Don't use Recursion.

3. Write a program in 'C' language to accept a tree as input and convert it into a binary tree. Print the resultant binary tree.

4. Write a program in 'C' language to accept a binary tree as input and check if the input binary tree is a full binary tree or not.

5. Write a program in 'C' language to accept two trees as input and check if both of them are the same. Give the appropriate message.

6. Write a program in ‘C’ language to count the number of leaves of a Binary tree.

1.8 ADVANCED TREES

The trees that were discussed in this section are Binary search trees and AVL trees.

A binary search tree (BST) is a binary tree in which all the elements of the left subtree of a node are less than the element stored in the node and the elements stored in the right subtree of the node. So, whenever there is a need to insert an element into a BST, the element is compared with the root. If the element is less than the root, then continue the comparison with the left node of the root recursively. If the element is larger than the root, then the same process has to be carried to the nodes at the right of the root recursively. BST is one of the major applications of the Binary Tree.

An AVL tree is a balanced Binary search tree.

For more information on Binary search trees and AVL trees , refer to unit-7 of MCS-21.

Session 7: Advanced trees (3 hours)

1. Write a program in ‘C’ language to create a binary search tree. Also, accept a key value and search for it in BST. Print the appropriate message as output.

2. Write a program in ‘C’ language to insert 15, 25, 2, 4, 3, 1, 50 into an initially empty AVL tree. Make assumptions, if necessary.

1.9 GRAPHS

A Graph is a collection of Vertices and Edges. The set of Vertices is always non-empty. Edges represent ordered or unordered pairs of vertices which are directly connected to each other. If the pairs of vertices are unordered, it means that an edge (v1,v2) indicates that v1 is directly connected to v2 and vice-versa. If the pairs of vertices are ordered, it means that an edge (v1,v2) indicates that v1 is directly connected to v2. V2 is directly connected to v1 only if the edge (v2,v1) is present in the list of edges.

For more information on Graphs, refer to unit-8 of MCS-21.

Session 8: Graphs (3 hours)

1. Write a program in ‘C’ language to implement Dijkstra’s algorithm.

2. Write a program in ‘C’ language to implement Kruskal’s algorithm.

3. Write a program in ‘C’ language to accept an undirected graph as input and print the list of all vertices in the Graph which are articulation points.

4. Write a program in ‘C’ language which accepts a directed graph as input and prints all the strongly connected components of the Graph.

1.10 SEARCHING AND SORTING

For information on Searching, refer to unit-9 of MCS-21.

For information on Sorting, refer to unit-10 of MCS-21.

Session 9: Searching and Sorting (3 hours)

1. Write a program in 'C' language to implement linear search using pointers.
2. Write a program in 'C' language to implement binary search using pointers.
3. Write a program in 'C' language to implement Quick sort using pointers.
4. Write a program in 'C' language to implement Heap sort using pointers.
5. Write a program in 'C' language to implement 2-way Merge sort using pointers.
6. Write a program in 'C' language to implement Bubble sort using pointers.
7. Write a program in 'C' language to implement Topological sort using pointers.

1.11 ADVANCED DATA STRUCTURES

The advanced data structures that are dealt in MCS-021 are Splay trees, Red-Black trees and AA-trees.

For more information on the above mentioned data structures, refer to Unit-12 of MCS-021.

Session 10: Advanced Data Structures (3 hours)

1. Write a program in 'C' language to insert 15, 25, 2, 4, 3, 1, 50 into an initially empty Splay tree.
2. Write a program in 'C' language for the creation of a Red Black tree. Also, implement insertion and deletion operations.
3. Write a program in 'C' language for the creation of a AA-tree. Also, implement insertion and deletion operations.

1.12 SUMMARY

In this section, we discussed the data structures that are covered in the course material of MCS-021 briefly. Each discussion is followed by a lab session. The session includes programming problems for the learners. More stress has been made on the programming using pointers as it is regarded as a very special skill. It is very important to attend the lab sessions with the necessary homework done. This enables better utilization of lab time. The learner can execute more programs in a given amount of time if s/he had come with preparation. Else, the learner may not be able to execute programs successfully. If the learner had executed program successfully in lab without sufficient preparation, then, it is very important to assess the efficiency of the program. In most cases, the programs lack efficiency. That is, the space and time complexities may not be optimal. In simple terms, it is possible to work out a better logic for the same program.

1.13 FURTHER READINGS

- The C programming language by Brian W.Kernighan and Dennis M. Ritchie;

Prentice Hall

- Data Structures and Algorithm analysis in C++ by Mark Allen Weiss; Pearson Education Asia

Web References

- http://www.isi.edu/~iko/pl/hw3 c.html
- http://www.programmersheaven.com