MA'DIN POLYTECHNIC COLLEGE

3071

For Semester IV

Data Structure LAB (3071) For CT

Department of Computer Engineering

INTRODUCTION:-

A data structure is one of the most important course 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 (Data Structures (3071)) is given. Each introduction is followed by a list of programming problems. It is suggested to you that **don't take copy of the program from anywhere**. You should **develop your own logic to write the program.**

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

• Thoroughly studied the Data structures (3071)

• Thoroughly studied the Advanced C Lab.

• Executed most of the programming problems of the Lab Questions.

• 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 (Data Structures-3071) 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.

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.

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.

Solution:

```
#include<stdio.h>
main()
 {
 int i,j,k,r1,c1,r2,c2,a[10][10],b[10][10],c[10][10];
 printf("Enter the no. of rows and columns of matrix 1 : ");
  scanf("%d%d",&r1,&c1);
 printf("Enter the no. of rows and columns of matrix 2 : ");
  scanf("%d%d",&r2,&c2);
 if(c1==r2)
  {
  printf("Enter the matrix 1\n");
  for(i=0;i<r1;i++)
   {
    for(j=0;j<c1;j++)
    {
     scanf("%d",&a[i][j]);
    }
   }
  printf("Enter the matrix 2\n");
  for(i=0;i<r2;i++)
   {
    for(j=0;j<c2;j++)
    {
     scanf("%d",&b[i][j]);
    }  }
  for(i=0;i<r1;i++)
   {
    for(j=0;j<c2;j++)
    {
    c[i][j]=0;
     for(k=0;k<r1;k++)
     {    c[i][j]=c[i][j]+a[i][k]*b[k][j];
     }  }  }
  printf("Product of matrices\n");
  for(i=0;i<r1;i++)
   {
    for(j=0;j<c2;j++)
    {    printf("%d\t",c[i][j]);    }
   printf("\n");
   } }
 else
  printf("Matrix multiplication not possible");
 printf("\n");
 }
```

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

1.3 STRUCTURES

s

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.

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.

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.

Solution (creation):

```
#include<stdio.h>
#include<stdlib.h>
#define NULL 0

struct linked_list
{
   int number;
   struct linked_list *next;
};
typedef struct linked_list node;  /* node type defined */

main()
{
   node *head;
   void create(node *p);
   int count(node *p);
   void print(node *p);
   head = (node *)malloc(sizeof(node));
   create(head);
   printf(“\n”);
   printf(head);
   printf(“\n”);
   printf(“\nNumber of items = %d \n”, count(head));
}
void create(node *list)
{
   printf(“Input a number\n”);
   printf(“(type -999 at end): “);
   scanf(“%d”, &list -> number); /* create current node */

   if(list->number == -999)
   {
      list->next = NULL;
   }
   else    /*create next node */
   {
      list->next = (node *)malloc(sizeof(node));
      create(list->next);
   }
   return;
}
void print(node *list)
{
      if(list->next != NULL)
      {
        printf(“%d—>”,list ->number);  /* print current
item */

         if(list->next->next == NULL)
          printf(“%d”, list->next->number);

        printf(list->next);        /* move to next item */
      }
      return;
   }

   int count(node *list)
   {
      if(list->next == NULL)
           return (0);
      else
           return(1+ count(list->next));
   }
```

2. Create a linked list and do the following operations-insertion, deletion, search and traverse.

3. Implement an algorithm to store a polynomial using linked list and perform addition.

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 are that the maximum numbers 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).

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

Postfix expression using pointers.

2. Write a program in C to implement stack using array

Solution:

```
#include<stdio.h>
#include<stdlib.h>
int st_arr[20];
int t=-1;
void push_ele(int ele);
int pop_ele();
void display_ele();
int main(void)
{
  char choice,num1=0,num2=0;
  while(1)
 {
clrscr();
printf("=======================================");
printf("\n\t\t MENU ");
printf("\n=======================================");
printf("\n[1] Using Push Function");
printf("\n[2] Using Pop Function");
printf("\n[3] Elements present in Stack");
printf("\n[4] Exit\n");
printf("\n\tEnter your choice: ");
fflush(stdin);
scanf("%c",&choice);
switch(choice-'0')
{
case 1:
     {
printf("\n\tElement to be pushed: ");
scanf("%d",&num1);
push_ele(num1);
break;
}
case 2:
     {
num2=pop_ele();
printf("\n\tElement to be popped: %d\n\t",num2);
getch();
break;
}
case 3:
     {
display_ele();
return 0;
break;
     }
     case 4:
exit(1);
break;
     default:
printf("\nYour choice is invalid.\n");
break;
   }
  }
}
/*Implementing the push() function. */
void push_ele(int ele)
{
 if(t==99)
 {
   printf("STACK is Full.\n");
   getch();
exit(1);
 }
  st_arr[++t]=ele;
}
/*Implementing the pop() function. */
int pop_ele()
{

 if(t==-1)
 {
printf("\n\tSTACK is Empty.\n");
getch();
exit(1);
 }
  return(st_arr[t—]);
}
/*Implementing display() function. */
void display_ele()
{
  int k;
  printf("\n\tElements present in the stack are:\n\t");
  for(k=0;k<=t;k++)
  printf("%d\t",st_arr[k]);
}
```

3. Write a program in 'C' language to reverse an input string.

4. 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.

1. Write a program to implement linear queue.

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

4. Write a program to implement linked queue.

5. Implement Circular queue using C program.

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.

1. Implement an algorithm to create a binary search tree and perform following operations- insertion, deletion, search, pre-order, in-order and post-order traversals.
2. Implement an algorithm to find the height of a tree.
3. Implement an algorithm for determining number of nodes in a tree.

1.8 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.

1. Implement BFS algorithm for traversing a graph.
2. Implement DFS algorithm for traversing a graph.
3. Implement Warshall's algorithm to find shortest path.

1.9 SEARCHING AND SORTING

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.

Solution:

```
#include <stdio.h>

int split ( int*, int, int ) ;

void main( )
{
        int arr[10] = { 11, 2, 9, 13, 57, 25, 17, 1, 90, 3 } ;
        int i ;

        void quicksort ( int *, int, int ) ;

        quicksort ( arr, 0, 9 ) ;

        printf ( "\nArray after sorting:\n") ;

        for ( i = 0 ; i <= 9 ; i++ )
                printf ( "%d\t", arr[i] ) ;

}

void quicksort ( int a[ ], int lower, int upper )
{
        int i ;
        if ( upper > lower )
        {
                i = split ( a, lower, upper ) ;
                quicksort ( a, lower, i - 1 ) ;
                quicksort ( a, i + 1, upper ) ;
        }
}

int split ( int a[ ], int lower, int upper )
{
        int i, p, q, t ;

        p = lower + 1 ;
        q = upper ;
        i = a[lower] ;

        while ( q >= p )
        {
                while ( a[p] < i )
                        p++ ;

                while ( a[q] > i )
                        q— ;

                if ( q > p )
                {
                        t = a[p] ;
                        a[p] = a[q] ;
                        a[q] = t ;
                }
        }

        t = a[lower] ;
        a[lower] = a[q] ;
        a[q] = t ;

        return q ;
}
```

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.

1.10 SUMMARY

In this section, we discussed the data structures that are covered in the course Data Structures, 3071 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.
