B.Tech.

Third Semester Examination, 2010-2011

Data Structure Using 'C' (CSE-201-F)

Note : Attempt any **five** questions.

Q. 1. (a) Write an algorithm that determines whether a given number is prime or not.

Ans. Step 1 : Take input from user.

Step 2 : Check whether the No. has any divisor other than/and the No. itself.

Step 3 : If there is any divisor other than 1 or that No. itself. Consider the No. as NOT PRIME.

Step 4 : Else consider the No. as a PRIME No.

Program to Find Prime No.

```
# include <stdio.h>
# include <conio.h>
void main( )
{
    int number, 1, flag = 1;
    printf("Enter the no. to check whether it is prime or not.");
    scanf("%d", & number);
    for (i=2; i <=number/2; i++)
{
    if ((number 1.1) = = 0);
{
    flag = 0;
    break;
}}
    if (flag = = 0)
    printf("This is not a prime no.");
        else
    printf("This is a prime no.");
    getch( );
    }
```

Q. 1. (b) What is an array ? Discuss the various operations that are allowed on array data structure. Describe the formula for calculating the address of any element of a two dimensional array.

Ans. An array is a finite collection of similar elements stored in adjacent memory locations. For example, an array cantains all integers or all characters but not both.

Array Operations : There are several operations that can be perform on an array. These operations are listed below :

(a) Traversal (b) Search (c) Insertion (d) Deletion
(e) Sorting (f) Merging (g) Reversing.

(a) Traversal : Processing each element in the array.

(b) Search : Finding the location of an element with a given value.

(c) Insertion : Adding a new elements to an array.

(d) Deletion : Removing an element from an array.

(e) Sorting : Organizing the elements in some order.

(f) Merging : Combining two arrays into a single array.

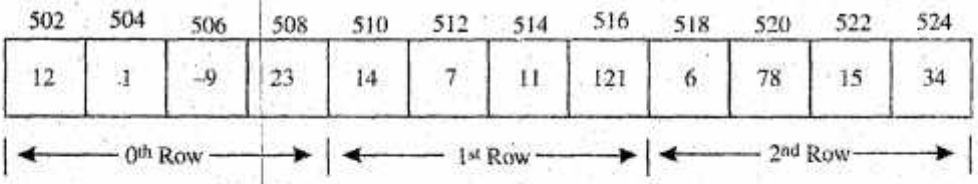
(g) Reversing : Reversing the elements of an array.

Two-Dimensional Arrays : A 2D array is a collection of elements placed in **m** rows and **n** columns. For e.g., arr[3][4] is a 2-D array containing 3 rows & 4 columns.

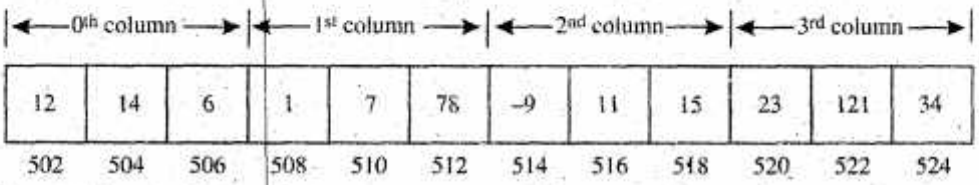
Representation of a 2-D array :

		0	1	2	3	Column
Row	0	12	1	−9	23	
	1	14	7	11	121	
	2	6	78	15	34	

Row Major Arrangement :



Column Major arrangement :



If base address is 502 and we want to refer the element 121 than the calculation :

Row Major Arrangement :

Element 121 is present at a[1][3]. So location of 121

$= 502 + 1 * 4 + 3$
$= 502 + 7$
$= 516$

$\left[\because \begin{matrix} a[i][j] \\ \text{Base add.} + i * n + j \end{matrix}\right]$

Column Major Arrangement :

Element 121 is present at a[i][j]. So

$= 502 + 3*3 + 1$

$= 502 + 10$

$= 522$

$\left[\because \begin{matrix} a[i][j] \\ \text{Base add.} + j * m + i \end{matrix}\right]$

Q. 2. (a) Write a non recursive algorithm for reversal of a string of characters.

Ans.

```
# include <stdio.h>
# include < conio.h>
# include <string.h>
char *_string_reverse (const char * src_string, char * dest_string)
{
      if (* src_string ! = '\0')
{
      dest_string- - = * src_string ++;
      _string_reverse (src_string, dest_string);
}
      return dest_string
}
      char * string_reverse (const char * my_string)
{
      int length = strlen (my_string);
      char * reyersed = (char *) malloc (length + 1);
      reversed [length] = '\0';
      char * tmp = &(reversed ([0]);
      reversed = _ string_reverse (my_string, reversed);
      return tmp;
      int main (void)
{
      const char * my_string = "Hello All !!";
      char * reversed = string_reverse (my_string);
      printf("\n\n original string : %s and Reversed String : (%s),
            my string; reversed);
return 0; }
```

Q. 2. (b) Write a program that takes a list pointed by LIST and traverses it in such a manner that after travel the links of the visited nodes become reversed.

Ans.

```
# include < stdio.h>
type def struct node
{
      int value;
```

```
        struct node*next;
        1 mynode;
        mynode*head, *tail, *temp.;
        void add (int value);
        void iterative_reverse( );
        void print_list( );
        int main( )
{
        head = (mynode*) 0;
        add(1);
        add(2);
        add(3);
        print_list( );
        iterative_reverse( );
        print_list( );
        return(0);
}
        void iterative_reverse( )
{
        mynode *p, *q, *r;
        if (head = = (mynode*) 0)
{
        return;
}
        p = head;
        q = p-> next;
        p → next = (mynode*) 0;
        while(q! = (mynode*) 0)
{
        r = q -> next
        q -> next = p;
        p = q;
        q = r;
}
        head = p;
}
        void add(int value)
{
```

```
        temp = (mynode*) malloc(sizeof(struct node));
        temp –> next = (mynode*)0;
        temp –> value = value;
        if head = = (mynode*)0)
{
        head = temp;
        tail = temp;
}
        else
{
        tail = temp;
}}
        void print_list( )
{
        printf("\n\n");
        for(temp = head; temp ! = (mynode*) 0; temp = temp –> next)
{
        printf("e%d) –>", (temp -> value));
}
        print("NULL} \n\n");
}
```

Q. 3. (a) Describe insertion sort algorithm and trace the steps of insertion sort for sorting the list – 12, 19, 33, 26, 29, 35, 22. Find the total number of comparison made.

Ans.

0^{th}	1^{st}	2^{nd}	3^{rd}	4^{th}	5^{th}	6^{th}
– 12	19	33	26	29	35	22

(a) In the first step the I^{st} element 19 is compared with the 0^{th} element – 12. Since – 12 is already smaller than 19. So No change is required.

– 12	19	33	26	29	35	22

(b) In second step the 2^{nd} element 33 and 0^{th} element –12 is compared. –12 is already small that 33 so no change. Then 2^{nd} element 33 is compared with first element 19. 19 is already small than 33 so no change required.

(c)

– 12	19	33	26	29	35	22

Now third element 26 is compared with 0^{th} element. No change. 3^{rd} comapred with 1^{st} no change. 3^{rd} compared with 2^{nd} element. 2^{nd} element is larger than 3^{rd} so change is required.

(d)

– 12	19	26	33	29	35	22

Now 4^{th} element compare with 0^{th} element, no change required 4^{th} element compare with I^{st} element, no change required. 4^{th} element compare with 2^{nd} element, no change required. 4^{th} element compare with 3^{rd}, there 3^{rd} element is larger than 4^{th} so change is required.

(e)

– 12	19	26	29	33	35	22

Now 5^{th} compare with 0^{st}, 1^{st}, 2^{nd}, 3^{rd} & 4^{th} no change required.

(f)

– 12	19	26	29	33	35	22

Now 6^{th} compare with 0^{th} element, no change required 6^{th} element compare with 1^{st}, no change. 6^{th} element compare with 2^{nd} change required, because 6^{th} is smaller than 2^{nd}. Than 6^{th} compare with 3^{rd} element, change is required because 6^{th} is smaller than 3^{rd}. Then 6^{th} compare with 4^{th} change is required. Then 6^{th} compare with 5^{th}, change is required.

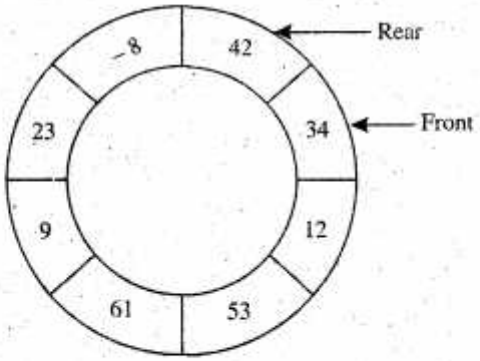
6 step is used to sort the algorithm.

– 12	19	22	26	29	33	35

Ans.

Q. 3. (b) What is circular queue ? Write a program that implements a circular queue.

Ans. Circular Queue : The queue that we implemented using an array suffers from one limitation. In that implementation there is a possibility that the queue is reported as full, even though in actually there might be empty slots at the beginning of the queue. To over come this limitation we can implement the queue as a circular queue.



Implement a Circular Queue :

```
# include <stdio.h>
# include <conio.h>
# define MAX 10
```

```
void addq (int*, int, int*, int*);
void display (int*);
void main( )
{
        int arr[MAX]
        int i, front, rear;
        clrscr( );
        front = rear = – 1;
        for (i = 0; i < MAX; i ++)
                arr[i] = 0;
        addq (arr, 14 &front, &rear);
        addq (arr, 22, &front, &rear);
        addq (arr, 13, &front, &rear);
        addq (arr, – 6, &front, &rear);
        addq (arr, 25, &front, &rear);
        printf ("\n Elements in the circular queue");
        display(arr);
                getch( );
        }
```

Q. 4. (a) Define heap and heap sort with suitable example.

Ans. A sorting algorithm that works by first organizing the data to be sorted into a special type of binary tree called heap. The heap sort algorithm must also reverse the order. The following steps are use :

(i) Remove the topmost item (the largest) and replace it with the rightmost leaf. The topmost item is sorted in an array.

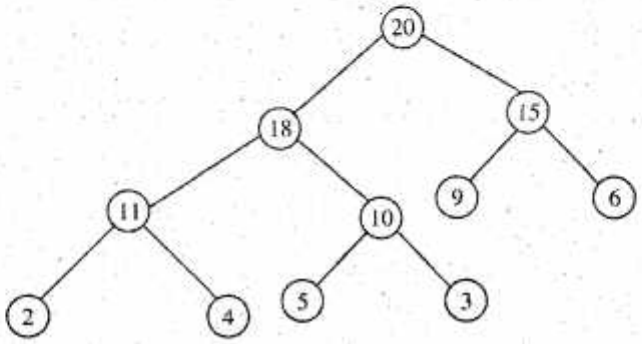
(ii) Re-establish the heap.

(iii) Repeat step (i) and (ii) until there are no more items left in the heap.

The sorted elements are now in an array.

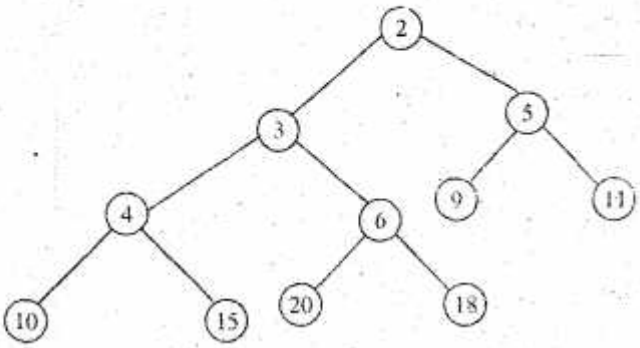
Max. Heap :

20	18	15	11	10	9	6	2	4	5	3



Min. Heap :

2	3	5	4	6	9	11	10	15	20	18



Q. 5. (b) Write a program which counts the number of times a given alphabet appears in a text file.

Ans.

```
# include <iostream.h>
# include <cctype.h>
using namespace std;
void read And count(int & numwords, intletter count [ ]);
void output letter counts (int letterCount[ ]);
int main( )
{
int numwords;
int letter count[26];
count <<endl;
cout <<"Enter the line of text" <<endl<<endl;
read And count(numwords, letterCounts);
cout <<endl;
cout <<numwords <<"words" <<endl;
output letterCounts (letterCount);
return 0;
}
void output letterCounts (int letterCount [ ])
{
for (int i=e; i<26; i++)
{
if letterCount [i] > 0)
{
```

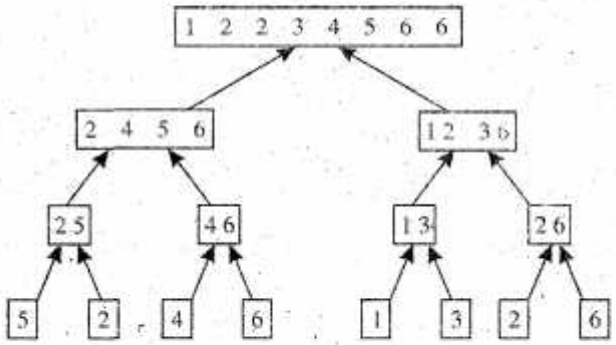
```
        cout <<leiterCount [i] <<"<< char ('a' + i) <<endl;
        }
        }
}
```

Q. 6. (a) Write merge sort algorithm and derive the expression for its run time complexity in best, average and worst case.

Ans. Merge sort is based on the divide and conquer paradigm. Its worst case running time has a lower order of growth than insertion sort.

Algorithm : To sort the entire sequence A[1..n], make the initial call to the procedure MERGE SORT

(A, 1, n)
MERGE_SORT (A, p, r)
(i) IF P < r
(ii) Then q = Floor[(p + r) / 2]
(iii) Merge (A, p, q)
(iv) Merge (A, q + 1, r)
(v) Merge (A, p, q, r)



Worst Case : The worst case scenario for merge sort is when, during every merge step exactly one value remain in the opposing list; in other words no comparison were skipped.

Best Case : Merge sort's best case is when the largest element of one sorted sub-list is smaller than first element of its opposing sub-list, for every merge step that occurs.

Time Complexity (Merge Sort) : The time complexity of merge sort in all three cases (best, average & worst) is

$$T(n) = \Theta(n \log_2 n)$$

Q. 6. (b) What is threaded binary tree ? Discuss with the help of examples.

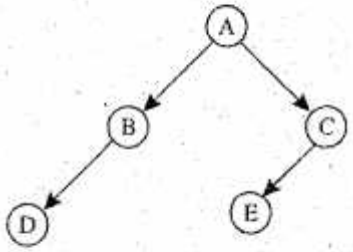
Ans. The threaded binary tree in which every node that does not have a right child has a threaded (in actual sense, a link) to its INORDER successor. By doing this threading we avoid the recursive method of traversing a tree, which makes use of stack and consumes a lot of memory and time.

The node structure for a threaded binary tree varies a bit and like this :

```
struct NODE
{
```

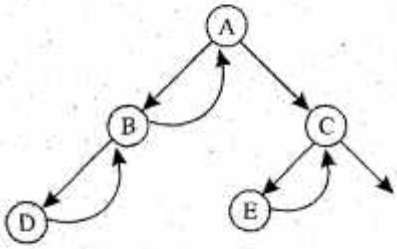
```
struct NODE * leftchild;
int node_value;
struct NODE * right child;
struct NODE * thread;
}
```

Make the threaded binary trees out of a normal binary tree.



Inorder traversal for the above tree is :

DBAEC, so the respective threaded binary tree will be :



Algorithm : (i) For the current node check whether it has a left child when which is not there in the visited list. If it has go to step (ii) or else step (iii).

(ii) Put the left child in the list visited node and make it your current node in consideration. Go to step (vi).

(iii) For the current node check whether it has a right child. If it has go to step (iv) else go to step (v).

(iv) Make the right child as your current node in consideration. Go to step (vi).

(v) Check for the threaded node and if its there make it your current node.

(vi) Go to step (i) if all nodes are not over otherwise quit.

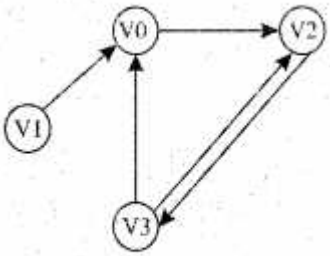
Q. 7. (a) Draw the directed graph that corresponds to the following adjacency matrix :

	V0	V1	V2	V3
V0	1	0	1	0
V1	1	0	0	0
V2	0	0	0	1
V3	1	0	1	0

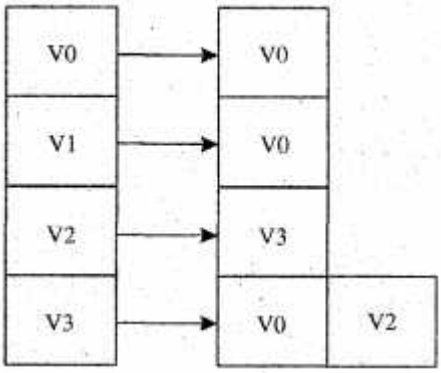
Also write down the adjacency list corresponding to the graph.

Ans.

	V0	V1	V2	V3
V0	1	0	1	0
V1	1	0	0	0
V2	0	0	0	1
V3	1	0	1	0



Adjacency List Representation



Q. 7. (b) Write an algorithm that inserts an edge into an undirected graph represented using an adjacency matrix.

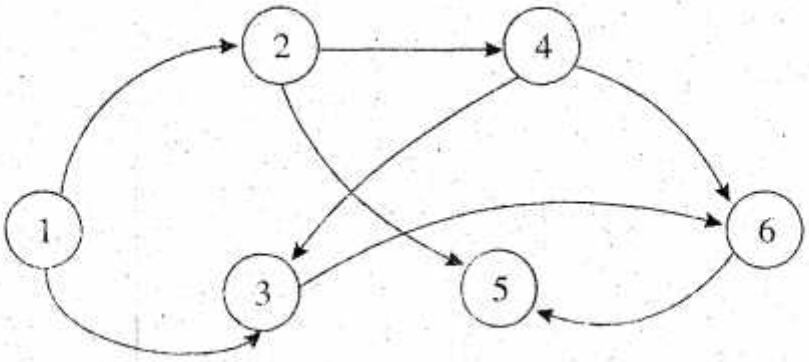
Ans. A graph G = (V, E) is an order pair of sets. The first set V_1 is the set of vertices. The second set, E is the set of edges. An edge in E is a pair of vertices from V_1 so an edge connects two vertices.

e.g., Consider the following graph :

G = (V, E), V = {1, 2, 3, 4, 5, 6} and

E = {(1, 2); (1, 3) (2, 4) (2, 5) (3, 1) (3, 6) (4, 6) (4, 3) (6, 5)}

Represented graphically, the graph looks like this



Now add edge (G, a, b) – add the edge (a, b) to G.

```
Void add_edge (graph*G; int a, int b)
{
    edgenode *p;
    P = malloc (size of (edgenode));
    P -> to = b;
    P -> from = a;
    P -> next = * G;
    * G = P;
```

Q. 8. Write short notes on any two of the following :

(i) B Tree

(ii) AVL Tree

(iii) Skip Lists

Ans. (i) B Tree : B Tree is a data structure that keeps data stored and allow searches, sequential access, insertions and deletions in logarithmic amortized time. The B Tree is a generalization of a binary tree in that a node can have more than two children. In B Tree internal node can have a variable no. of child nodes with some pre-define range. When data is inserted and removed from a node, its no. of child nodes changes. In order to maintain predefine range, internal nodes may be joined or split.

The best case height of a B tree is

$$\log_m n.$$

The worst case height of a B tree is

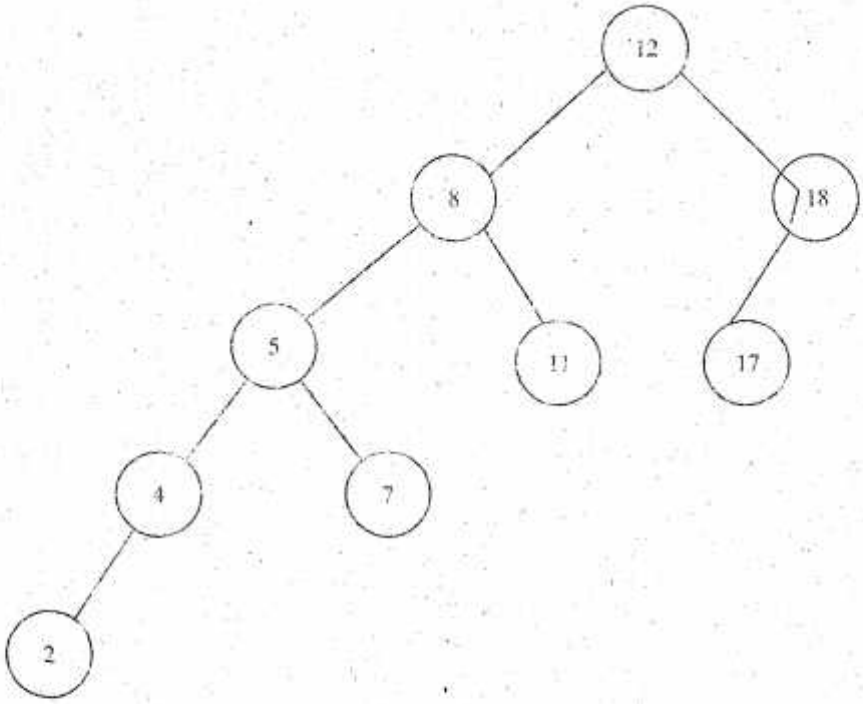
$$\log_{m/2} n$$

(ii) AVL Tree : AVL Tree is another balanced binary search tree. Named after the inventors. Adelson Velskii and Landis, they were first dynamically balanced tree to be proposed.

An AVL tree is a binary search tree which has the following properties :

(i) The sub-tree of every node differ in height by the most one.

(ii) Every sub-tree is an AVL tree.



(iii) Skip Lists : A skip list is a data structure for storing list of items, using a hierarchy of linked list that connects increasingly sparse subsequences of the items. The auxiliary list allow item looking with efficiency comparable to balanced binary search trees.