INDEX

Table of Contents

❑❑❑

Syllabus

Module 1 : Introduction to Data Structure

Types of Data Structure, Arrays, Strings, Recursion, ADT (Abstract Data type),Concept of Files, Operations with files, types of files.

Linear Data Structure

Module 2 : Linked List

Linked List as an ADT, Linked List Vs. Arrays, Memory Allocation & De-allocation for a Linked List, Linked List operations, Types of Linked List, Implementation of Link.

Module 3 : Stack

The Stack as an ADT, Stack operation, Array Representation of Stack, Link Representation of Stack, Application of stack – Recursion, Polish Notation.

Module 4 : Queues

The Queue as an ADT, Queue operation, Array Representation of Queue, Linked Representation of Queue, Circular Queue, Priority Queue, & Dequeue, Application of Queues – Johnsons Algorithm, Simulation.

Non-linear Data Structure

Module 5 : Trees

Basic trees concept, Binary tree representation, Binary tree operation, Binary tree traversal, Binary search tree implementation, Thread Binary tree, The Huffman Algorithm, Expression tree, Introduction to Multiway search tree and its creation(AVL, B-tree, B+ tree).

Module 6 : Graphs

Basic concepts, Graph Representation, Graph traversal (DFS & BFS).

Module 7 :

Sorting :

Sort Concept, Shell Sort, Radix sort, Insertion Sort, Quick Sort, Merge Sort, Heap Sort.

Searching :

List Search, Linear Index Search, Index Sequential Search Hashed List Search, Hashing Methods, Collision Resolution.

□□□

Data Structures

Statistical Analysis

Chapter No.	Dec. 2013
Chapter 1	05 Marks
Chapter 2	07 Marks
Chapter 3	–
Chapter 4	10 Marks
Chapter 5	15 Marks
Chapter 6	10 Marks
Chapter 7	12 Marks
Chapter 8	30 Marks
Chapter 9	13 Marks
Chapter 10	18 Marks
Repeated Questions	–

Dec. 2013

Chapter 1 : Types of Data Structure and Recursion [Total Marks : 05]

Q. 1(a) Explain linear and non linear data structures with examples. **(5 Marks)**

Ans. : The data structure can be subdivided into two categories.

(a) The **primitive data structure** are nothing but the data type such as int [Integer], float [Fractional], char [character] which indicates the type of the data.

(b) **Non-primitive data structure :**

This non-primitive data structures are again divided into two types as follows :

(i) **Linear data structures :** In this type of data structure one element will point to only one or two elements.

E.g. **Stack :** A stack is last in first out (LIFO) type of data structure. Where addition and deletion only done at one end called as top of the stack. You can implement stack statically using array or dynamically using linked list. Dynamic implementation approach is superior.

Queue : A queue is first in first out (FIFO) type of data structure. Queue addition can happened only at the end i.e. at rear position and deletion can happened at beginning i.e. at front position. A queue is created either statically with an array or dynamically with linked list. Link list queue is superior.

(ii) **Non-linear data structure :** In this type of data structures one element will point hierarchy of an elements.

E.g. **Tree :** A tree is a set of elements where one element is called as Root and remaining elements are subdivided into n sets, t_1 to t_n where each of the sets is tree.

Leaf node : A leaf node is terminal node of tree.

e.g. Leaf nodes of Fig. 1 are as follows,
G, D, E, C, H, J.

Degree of a tree : It is maximum degree of nodes in the tree. Degree of a node is number of subtrees it has. e.g. Degree of tree in Fig. 1 is three.

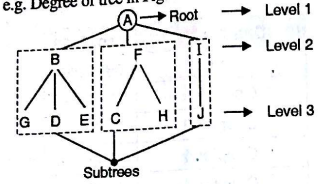


Fig. 1 : A regular tree

Parent node : It is a node having other nodes connected to it. These nodes are called children of that node.
e.g. Here node A is parent node for node B, F and I.
and B is parent node for G, D and E.
and F is parent node for C and H.
and I is parent node for J. (Refer Fig. 1)

Siblings : Children of same parent node are called as siblings.
e.g. B, F and I are siblings of A.
G, D, E are siblings of B.
C, H are siblings of F. (Refer Fig. 1)

Chapter 2 : Arrays [Total Marks : 07]

Q. 1(c) Write a 'C' program to convert decimal to binary using any appropriate data structure you have studied. **(7 Marks)**

Ans. :

Program to convert decimal to binary :

```
# include<stdio.h>
# include<math.h>        // Including math.h file for pow function. Pow( ) - power function
void conv_deci(void);
void main()
{
      clrscr();
      conv_deci();
      getch();
}
void conv_deci()
{
      int num=0;
      int dnum=0;
      int i=0, rem=0;
```



```
      printf("Enter number");
      scanf("%d", &num);
      printf("Decimal of %d is :", num);
      while(num!=0)
      {
            rem=num%10;                    // Calculating remainder of a number.
            dnum=dnum+(pow(2,i)*rem);
            num=num/10;

            i++;
      }
      printf("%d", dnum);
}
```

Output → **Enter number – 1111**
Decimal of 1111 is 15.

Chapter 4 : File Handling in C [Total Marks : 10]

Q. 6(a) Explain indexed sequential search with a suitable example. What are the advantages and disadvantages of indexed sequential search ? **(10 Marks)**

Ans. : Indexing :

Files are typically stored on magnetic disks. Records are stored in files. In database application, we have to store several files like :
(1) File of records of students (2) Marks of students (3) Fee details of students

Each record in a file s identified by a key field. Record of a student can be located by his roll number. Thus roll number is the key field.

To search for a record on disk, a record must be read in memory buffer. If the record does not contain the desired key then the next record must be read. The process of reading the next record continues until the desired record is located or all records have been read.

This can be very time consuming for a large file. The goal is to locate the record with minimum number of record accesses. Indexing can speed up retrieval of records. In indexing, we use an additional auxiliary access structures called indexes. Index structure provides an addition access path to quickly retrieve a record form the main file. An index is an orderd file whose records are of fixed length with two fields :
(a) Key field (b) Record no.

Searching a record in an indexed file : Since an index is an ordered file, a key can be searched using binary search. If search ends in a success, the index entry will give the required record no. Record no. can be used to access the record in data file.

Roll no.	Record no.
75	1
101	0
125	2
150	4

	Name	Roll no.	Age	——
0	Mohan	101	24	——
1	Sohan	75	23	——
2	Pradeep	125	25	——
3	Amit	160	24	——

Roll no.	Record no.
160	3

Index file

4

Name	Roll no.	Age	——
Deepak	150	26	——

Data file
(Student file)

Advantages of a sequential file :

(a) Reading the records in order of ordering key values is extremely efficient.

(b) Searching a record on an ordering key field results in faster access as binary search technique can be used.

Disadvantages of sequential file : Inserting and deleting records are expensive as records must remain physically ordered. To insert a record, we must find its correct position in the file and then make space in the file to insert the record in that position. similarly, to delete a record, we must find the record to be deleted and then physically move all records ahead of it, backward by 1 place.

Chapter 5 : Linked List [Total Marks : 15]

Q. 1(d) Define ADT with an example. **(3 Marks)**

Ans. : ADT :

The concept of abstraction is commonly found in computer science. The big program is never written as a monolithic piece of a program, instead it is broken down in smaller modules.

Abstraction in case of data :

Abstraction for primitives types (char, int, float) is provided by the compiler. For example, We use integer type data and also, perform various operation on them without knowing them.

1. Representation 2. How various operations are performed on them.

Example : int x , y, z; x = 13;

Constant 13 is converted to 2's complement and then stored in x. Representation is handled by the compiler x = y + z ;

Meaning of the operation ' + ' is defined by the compiler and its implementation details remain hidden from the user. Implementation details (representation) and how various operations are implemented remain hidden from the user. User is only concerned about, how to use these operations. Objects such as lists, sets and graphs along with associated operations, can be viewed as abstract data type. Integer, char, real are primitive data types and there are set of operations associated with them for the set of abstract data types [ADT].

Q. 4(a) Write a 'C' program to create a "Single Linked List" ADT. The ADT should support the following functions. **(12 Marks)**

(i) Creating a Linked List. (ii) Inserting a node after a specific node.

(iii) Deleting a node (iv) Displaying the list

Ans. : Implementation of singly linked list :

```
#include <stdio.h>
#include<conio.h>
#include <malloc.h>
struct node
{
      int info;
      struct node *link;
}*start;
main()
{
      int choice,n,m,position,i;
      clrscr();
      printf("\n\tSingle Linked List\n");
      printf("\n~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n\n");
      start=NULL;
      while(1)
      {
            printf("\t1.Create List\n");
            printf("\t2.Add at begining\n");
            printf("\t3.Add after \n");
            printf("\t4.Delete\n");
            printf("\t5.Display\n");
            printf("\t6.Quit\n");
            printf("\tEnter your choice : ");
            scanf("%d",&choice);
            switch(choice)
            {
             case 1:
                  printf("\nHow many nodes you want to insert : ");
                  scanf("%d",&n);
                  for(i=0;i<n;i++)
                  {
                        printf("\n\nEnter the element : ");
                        scanf("%d",&m);
                        create_list(m);
                  }
                  break;
            case 2:
                  printf("\n\nEnter the element : ");
                  scanf("%d",&m);
                  addatbeg(m);
```

```
                break;
            case 3:
                printf("\n\nEnter the element : ");
                scanf("%d",&m);
                printf("\nEnter the position after which this element is inserted : ");
                scanf("%d",&position);
                addafter(m,position);
                break;
            case 4:
                if(start==NULL)
                {
                    printf("List is empty\n");
                    continue;
                }
                printf("Enter the position for deletion : ");
                scanf("%d",&m);
                del(m);
                break;
            case 5:
                display();
                break;

            case 6:
                exit();
            default:
                printf("Wrong choice\n");
            }/*End of switch */
        }/*End of while */
}/*End of main()*/
create_list(int data)
{
    struct node *q,*tmp;
    tmp= malloc(sizeof(struct node));
    tmp->info=data;
    tmp->link=NULL;

    if(start==NULL) /*If list is empty */
        start=tmp;
    else
    {   /*Element inserted at the end */
        q=start;
```



```
        while(q->link!=NULL)
            q=q->link;
        q->link=tmp;
    }
}/*End of create_list()*/

addatbeg(int data)
{
    struct node *tmp;
    tmp=malloc(sizeof(struct node));
    tmp->info=data;
    tmp->link=start;
    start=tmp;
}/*End of addatbeg()*/
addafter(int data,int pos)
{
    struct node *tmp,*q;
    int i;
    q=start;
    for(i=0;i<pos-1;i++)
    {
        q=q->link;
        if(q==NULL)
        {
            printf("There are less than %d elements",pos);
            return;
        }
    }/*End of for*/

    tmp=malloc(sizeof(struct node) );
    tmp->link=q->link;
    tmp->info=data;
    q->link=tmp;
}/*End of addafter()*/
del(int data)
{
    struct node *tmp,*q;
    if(start->info == data)
    {
        tmp=start;
        start=start->link;  /*First element deleted*/
        free(tmp);
```

```
            return;
      }
      q=start;
      while(q->link->link != NULL)
      {
            if(q->link->info==data)    /*Element deleted in between*/
            {
                  tmp=q->link;
                  q->link=tmp->link;
                  free(tmp);
                  return;
            }
            q=q->link;
      }/*End of while */
      if(q->link->info==data)    /*Last element deleted*/
      {
            tmp=q->link;
            free(tmp);
            q->link=NULL;
            return;
      }
      printf("Element %d not found\n",data);
}/*End of del()*/
display()
{
      struct node *q;
      if(start == NULL)
      {
            printf("List is empty\n");
            return;
      }
      q=start;
      printf("\nList is :\n\n");
      while(q!=NULL)
      {
            printf("%d ", q->info);
            q=q->link;
      }
      printf("\n");
}/*End of display() */
```

Output :

Singly linked list :

1. Create List
2. Add at beginning

3. Add after
4. Delete
5. Display
6. Quite

Enter your choice : 1

How many nodes you want to insert : 3

Enter element : 45

Enter element : 56

Enter element : 87

1. Create List
2. Add at beginning
3. Add after
4. Delete
5. Display
6. Quite

Enter your choice: 5

List is :

45 56 87

1. Create List
2. Add at beginning
3. Add after
4. Delete
5. Display
6. Quite

Enter your choice : 3

Enter element to be inserted : 27

Enter the position after which element to be inserted : 2

1. Create List
2. Add at beginning
3. Add after
4. Delete
5. Display
6. Quite

Enter your choice : 5

List is : 45 56 27 87

Chapter 6 : Stack [Total Marks : 10]

Q. 2(b) Write a program in 'C' to evaluate a postfix expression. **(10 Marks)**

Ans. :

C program for evaluate a postfix expression :

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

```
#define MAX 50
typedef struct stack
{
        int data[MAX];
        int top;
}stack;

int  precedence(char);
void eval_postfix(char postfix[]);

void main()
 { char postfix[30];
   clrscr();
   printf("\nEnter an postfix expression : ");
   gets(postfix);
   printf("\nPostfix evaluation : ");
   eval_postfix(postfix);
   getch();
 }
void eval_postfix(char postfix[])
 {
        stack s;
        char x;
        int op1,op2,val,i;
        init(&s);
        for(i=0;postfix[i]!='\0';i++)
         {      x=postfix[i];
                if(isalpha(x))
                {       printf("\nEnter the value of %c : ",x);
                        scanf("%d",&val);
                        push(&s,val);
                }
                else
                {       //pop two operands and evaluate
                        op2=pop(&s);
                        op1=pop(&s);
                        val=evaluate(x,op1,op2);
                        push(&s,val);
                }
        }
        val=pop(&s);
        printf("\nvalue of expression = %d",val);
}
```


Chapter 7 : Queues [Total Marks : 12]

Q. 3(a) Write a program in 'C' to implement a circular queue. The following operations should be performed by the program. **(12 Marks)**

(i) Creating the queue. (ii) Deleting from the queue.
(iii) Inserting in the queue. (iv) Displaying all the elements of the queue.

Ans. :

A program for circular queue using an array :

```
#include<conio.h>
#include<stdio.h>
#define MAX 10
typedef struct Q
{
        int R,F;
        int data[MAX];
}Q;
void initialise(Q *P);
int empty(Q *P);
int full(Q *P);
void enqueue(Q *P,int x);
int dequeue(Q *P);
void print(Q *P);
void main()
{
        Q q;
        int op,x;
        initialise(&q);
        clrscr();
        do
          {
                printf("\n\n1)Insert\n2)Delete\n3)Print\n4)Quit");
                printf("\nEnter Your Choice:");
                scanf("%d",&op);
                switch(op)
                  {
                        case 1: printf("\n Enter a value:");
                                scanf("%d",&x);
                                if(!full(&q))
                                        enqueue(&q,x);
                                else
                                        printf("\nQueue is full !!!!");
                                break;
                        case 2: if(!empty(&q))
```

```
                    {
                        x=dequeue(&q);
                        printf("\Deleted Data=%d",x);
                    }
                    else
                    printf("\nQueue is empty !!!!");
                    break;
            case 3: print(&q);break;
            }
    }while(op!=4);
}
void initialise(Q *P)
{
    P->R=-1;
    P->F=-1;
}
int empty(Q *P)
{
    if(P->R==-1)
        return(1);
    return(0);
}
int full(Q *P)
{
    if((P->R+1)%MAX==P->F)
        return(1);
    return(0);
}
void enqueue(Q *P,int x)
{
    if(P->R==-1)
    {
        P->R=P->F=0;
        P->data[P->R]=x;
    }
    else
    {
        P->R=(P->R+1)%MAX;
        P->data[P->R]=x;
    }
}
int dequeue(Q *P)
{
    int x;
```



```
    x=P->data[P->F];
    if(P->R==P->F)
    {
        P->R=-1;
        P->F=-1;
    }
    else
        P->F=(P->F+1)%MAX;
    return(x);
}
void print(Q *P)
{
    int i;
    if(!empty(P))
    {
        printf("\n");
        for(i=P->F;i!=P->R;i=(i+1)%MAX)
            printf("%d\t",P->data[i]);
        printf("%d\t",P->data[i]);
    }
}
```

Chapter 8 : Trees [Total Marks : 30]

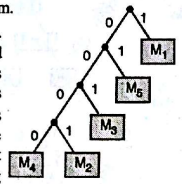
Q. 2(a) What is Huffman Coding. Construct the Huffman Tree and determine the code for the following characters whose frequencies are as given : **(10 Marks)**

Characters	A	B	C	D	E
Frequency	20	10	10	30	30

Ans. : Huffman coding :

Another application of binary trees with minimum weighted external path length is to obtain an optimal set of codes for messages $M_1, M_2, \ldots, M_{n+1}$. Each code is a binary string that is used for the transmission of the corresponding message. Huffman code is used in encoding that is encrypting or compressing the text in the WSSS communication system.

It uses patterns of zeros (0^s) and (1^s). In communication system these are used at sending and receiving end. Suppose there are n standard messages $M_1, M_2, \ldots, M_n$ then the frequency of each message is considered, that is message with highest frequency is given priority for the encoding. Suppose there are 5 messages M_1, M_2, M_3, M_4, M_5. M_1 having the highest frequency then M_5, M_3, M_2 and M_4. The encoding pattern will as follows,



$M_1 = 1$
$M_5 = 01$
$M_3 = 001$
$M_2 = 0001$
$M_4 = 0000$

Fig. 2 : Huffman codes

The tree is called encoding tree and is present at the sending end. The decoding tree is present at the receiving end and which decodes bit string to the corresponding message.

The cost of decoding is directly proportional to the number of bits in the transmitted code is equal to distance of external node from the root in the tree. Therefore the decode time for message will be,

$$M_i = \sum_{i=1}^{n} d_i q_i$$

Where, d_i = distance of external node (q_i) from the root. Therefore, decoding time can be reduced if the messages are chosen so as it decode tree with minimum weighted path length.

Characters	A	B	C	D	E
Frequency	20	10	10	30	30

Find out optimal merge pattern tree as follows :

Step 1 : Sort the frequencies in ascending order.
10, 10, 20, 30, 30

Step 2 :

Step 3 :

Step 4 :

Step 5 :

Step 6 :

Step 7 :

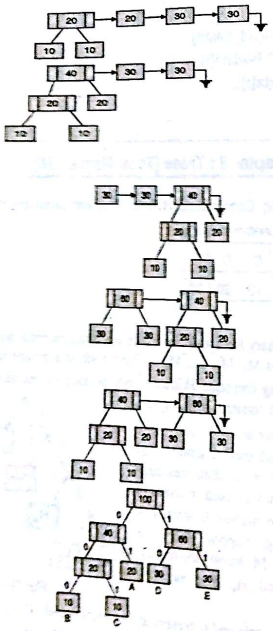


Fig. 3 : Binary tree merge pattern to obtain Huffman code for given frequencies

The Huffman codes are as follows :

A – 01
B – 000
C – 001
D – 10
E – 11

Q. 5(a) Discuss AVL trees. Insert the following elements in a AVL search tree : **(10 Marks)**
27, 25, 23 ,29, 35, 33, 34

Ans. : AVL tree :

AVL tree is a binary tree in which depth (height) of two subtrees of any node never differ by more than 1.

Balance of node = Depth of left subtree – Depth of right subtree

In AVL it will be 0 or 1 of –1 . The AVL trees are also called as balanced binary trees.

Right Heavy and Left Heavy AVL tree :

Right heavy : A node is called right heavy when its right subtree is one more than height of its left subtree.

Left heavy : A node is called left heavy when its left subtree is one more than height of its right subtree.

Balanced node : A node is called balanced if the height of right and left subtrees are equal.

The balanced factor will be 1 for left heavy, – 1 for right heavy and 0 for balanced node. So in an AVL tree each node can have only 3 values of balance factor which are – 1, 0, 1 or in other words we can say that the absolute value of balance factor should be less than or equal to 1. For example consider the following in Fig. 4.

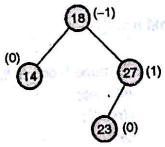
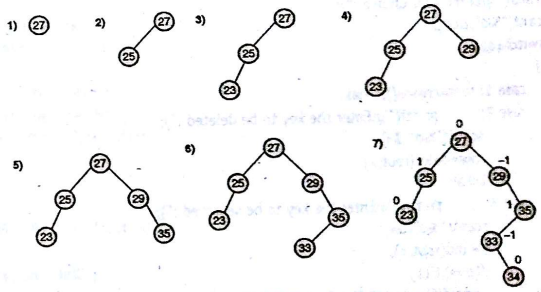


Fig. 4 : AVL Tree T

Given elements,
27, 25, 23, 29, 35, 33, 34



Q. 6(b) Write a program in 'C' for deletion of a node from a Binary Search Tree. The program should consider all the cases. **(10 Marks)**

Ans. : Program for deletion of a node from a binary search tree :

```
// program showing various operations on binary search tree
#include<conio.h>
#include<stdio.h>
#include<stdlib.h>

typedef struct BSTnode
{
        int data;
        struct BSTnode *left,*right;
}BSTnode;

BSTnode *find(BSTnode *,int);
BSTnode *insert(BSTnode *,int);
BSTnode *delet(BSTnode *,int);
BSTnode *create();
void inorder(BSTnode *T);
void preorder(BSTnode *T);
void postorder(BSTnode *T);

void main()
{
       BSTnode *root=NULL,*p;
       int x,op;
       clrscr();
       do
        { printf("\n\n1)Create\n2)Delete\n3)Search\n4)Preorder");
          printf("\n5)Inorder\n6)Postorder\n7)Insert\n8)Quit");
          printf("\nEnter Your Choice :");
          scanf("%d",&op);
          switch(op)
          {
             case 1: root=create();break;
             case 2:       printf("\nEnter the key to be deleted :");
                    scanf("%d",&x);
                    root=delet(root,x);
                    break;
             case 3:       printf("\nEnter the key to be searched :");
                    scanf("%d",&x);
                    p=find(root,x);
                    if(p==NULL)
                     printf("\n ***** Not Found*****");
                    else
                      printf("\n ***** Found******");
                    break;
             case 4: preorder(root);break;
             case 5: inorder(root);break;
             case 6: postorder(root);break;
             case 7: printf("\nEnter a data to be inserted : ");
                    scanf("%d",&x);
                    root=insert(root,x);
          }
        }while(op!=8);
}


void inorder(BSTnode *T)
{
       if(T!=NULL)
       {
              inorder(T->left);
              printf("%d\t",T->data);
              inorder(T->right);
       }
}

void preorder(BSTnode *T)
 {     if(T!=NULL)
        { printf("%d\t",T->data);
          preorder(T->left);
          preorder(T->right);
        }
 }
void postorder(BSTnode *T)
 {     if(T!=NULL)
        {
          postorder(T->left);
          postorder(T->right);
          printf("%d\t",T->data);
        }
 }

BSTnode *find(BSTnode *root,int x)
{
       while(root!=NULL)
       {
```

```
            if(x==root->data)
                return(root);
            if(x>root->data)
                root=root->right;
            else
            root=root->left;
        }
        return(NULL);
}

BSTnode *insert(BSTnode *T,int x)
{
        BSTnode *p,*q,*r;
        // acquire memory for the new node
        r=(BSTnode*)malloc(sizeof(BSTnode));
        r->data=x;
        r->left=NULL;
        r->right=NULL;
        if(T==NULL)
            return(r);
        // find the leaf node for insertion
        p=T;
        while(p!=NULL)
        {
            q=p;
            if(x>p->data)
                p=p->right;
            else
                if(x<p->data)
                    p=p->left;
                else
                {
                    printf("\nDuplicate data : ");
                    return(T);
                }
        }
        if(x>q->data)
            q->right=r;  // x as right child of q
        else
            q->left=r;   //x as left child of q
        return(T);
}

BSTnode *delet(BSTnode *T,int x)
```



```
{
        BSTnode *temp;
        if(T==NULL)
        {
            printf("\nElement not found :");
            return(T);
        }
        if(x < T->data)          // delete in left subtree
        {
            T->left=delet(T->left,x);
            return(T);
        }
        if(x > T->data)          // delete in right subtree
        {
            T->right=delet(T->right,x);
            return(T);
        }

        // element is found
        if(T->left==NULL && T->right==NULL)  // a leaf node
        {
            temp=T;
            free(temp);
            return(NULL);
        }
        if(T->left==NULL)
        {
            temp=T;
            T=T->right;
            free(temp);
            return(T);
        }
        if(T->right==NULL)
        {
            temp=T;
            T=T->left;
            free(temp);
            return(T);
        }
        // node with two children
        //go to the inorder successor of the node
        temp=T->right;
        while(temp->left !=NULL)
            temp=temp->left;
```

```
        T->data=temp->data;
        T->right=delet(T->right,temp->data);
        return(T);
}

BSTnode *create()
{
        int n,x,i;
        BSTnode *root;
        root=NULL;
        printf("\nEnter no. of nodes :");
        scanf("%d",&n);
        printf("\nEnter tree values :");
        for(i=0;i<n;i++)
        {
                scanf("%d",&x);
                root=insert(root,x);
        }
        return(root);
}
```

Chapter 9 : Graphs [Total Marks : 13]

Q. 1(b) Explain various techniques of graph representations. **(5 Marks)**

Ans. :

Representation of graph : There are two ways for representing graph in memory.

i) Adjacency matrix :

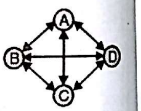


Fig. 5 : Graph G

	A	B	C	D
A	∞	1	1	1
B	1	∞	1	1
C	1	1	∞	1
D	1	1	1	∞

P [i] [j] = 1 if there is an edge from i to j.

P [i] [j] = ∞ if no edge.

Create the adjacency matrix of given graph by using above method. Write 1 if there exist an edge between two nodes otherwise write ∞.

ii) Linked-list representation : [Adjacency list]

Here you have to maintain a linked list separate for each node which indicates adjacent nodes for that node.

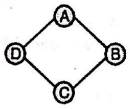
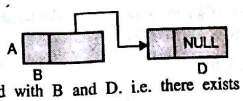


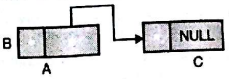
Fig. 6 : Graph G

Link list for node A.

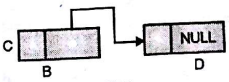


Here node A is connected with B and D. i.e. there exists edges AB and AD. Link list representation is shown above.

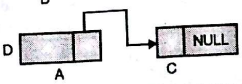
Adjacency list for Node B



Adjacency list for Node C



Adjacency list for Node D



Q. 4(b) Explain various graph traversal techniques with examples. **(8 Marks)**

Ans. :

Traversing the graphs :

Many of the times we want to visit each node of the graph in some specific order. Here traversing means visit each and every node or vertices of the graph exactly once. There are basically two techniques used for traversing of graph and they are as follows,

1. Depth First Search (DFS) :

Suppose you start from node A, then first visit A and then visit each node N along a path P which begins at A i.e. you process neighbor of A, then neighbors of A and so on.

Finally when you reach the dead end, move back to previous node in the path and traverse another path from there and so on.

e.g.

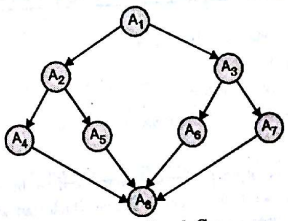


Fig. 7 : Graph G

Above Fig. 7 shows graph G. So DFS of this graph G which starts from node A_1 is as follows,

$A_1, A_2, A_4, A_8, A_5, A_3, A_6, A_7$.

Search dipper whenever possible. Choose any vertex $v \in V$ as starting point, mark it as visited. If there Q (1) is a vertex adjacent to V i.e. not marked as visited, choose this vertex as new starting node and call DFS recursively.

Every DFS gives rise to DFS spanning tree (without any cycle)

Descendants : Descendants of a node are all such nodes reachable from that node at lower level.
e.g. A_4, A_5, are descendants of A_2 (Refer Fig. 7)

Ancestors : Ancestors of node are all the nodes from the root towards that node.
e.g. A_1, A_2, are ancestors of A_4. (Refer Fig. 7)

Every depth first search gives a depth first spanning tree without any cycle. The tree is created with the help of adjacent nodes i.e. we first check the descendants of the starting node and so on. The edge A_1 to A_2 is exists only when the A_1 is said to be descendant of A_2. We create a tree node by node and it is possible only when the one node say u is ancestor of other node v. If they are not ancestors of each other then we can say that there is no edge exists between u and v.

2. Breadth First Search (BFS) :

Suppose you start from node A. Then you visit all unvisited neighbors of A. Then you visit all unvisited neighbors of A and so on. This process continues till all vertices are visited.

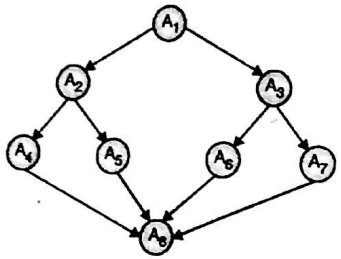


Fig. 8 : Given graph G

BFS of above graph G i.e. Fig. 8 are as follows which start from node A_1.
$A_1, A_2, A_3, A_4, A_5, A_6, A_7, A_8$.

Chapter 10 : Searching and Sorting [Total Marks : 18]

Q. 3(b) Sort the following elements using Radix Sort :
121, 70, 965, 432, 12, 577, 683.
What is the limitations of Radix Sort ?

Ans. : **(8 Marks)**

Radix sort :

Radix sort is generalization of the bucket sort. Suppose if we have a list of 100 names and we want to sort them alphabetically then first we take the name which start with alphabet 'A' and then we take the names which start from alphabet 'B' and so on. In list of names the radix is 26 (i.e. all alphabets A-Z). In second pass we take names in part on the basis of second alphabet of names. Similar process we take for other passes.

Similarly if we have list of numbers then there will be 10 parts or say radix from 0 to 9. In the first pass we take the numbers in parts on the basis of unit digit (least significant digit). In second pass the base will be tens digit, for third pass the base will be hundredth digit and so on. If largest number has three digits then number will be sorted in maximum three passes.

Consider following numbers in unsorted order and sort them by applying radix sort.
121,70,965,432,12,577,683

1. Largest number has three digits. So to sort above list we require maximum three passes.
2. To sort in ascending order collect numbers from left to right and top to bottom fashion.
3. To sort in descending order collect numbers from right to left and top to bottom fashion.

To sort in Ascending order.

Pass 1 : In pass 1 we store the numbers according unit digit or least significant digit.

Radix / Numbers	0	1	2	3	4	5	6	7	8	9
121		121								
070	070									
965						965				
432			432							
012			012							
577								577		
683				683						

To get list in ascending order we collect the numbers from left to right and top to bottom fashion as follows, i.e. after pass 1 list is,

070, 121, 432, 012, 683, 965 and 577

Pass 2 : In pass 2 we store the numbers according to 10^{th} digit or second least significant digit.

Radix / Numbers	0	1	2	3	4	5	6	7	8	9
070								070		
121			121							
432				432						
012		012								
683									683	
965							965			
577								577		

Again collect the numbers from left to right and top to bottom fashion i.e. after pass 2 list is,
012, 121, 432, 965, 070, 577 and 683.

Pass 3 : In pass three we store the numbers according to 100^{th} digit or third least significant digit.

Radix / Numbers	0	1	2	3	4	5	6	7	8	9
012	012									
121		121								
432					432					
965										965
070	070									
577						577				
683							683			

Again collect the numbers from left to right and top to bottom fashion. i.e. after pass 3 the list is as follows,

012, 070, 121, 432, 577, 683 and 965.

Q. 5(b) Using linear probing and quadratic probing insert the following values in a hash table of 10. Show how many collisions occur in each technique : **(10 Marks)**

99, 33, 23, 44, 56, 43, 19

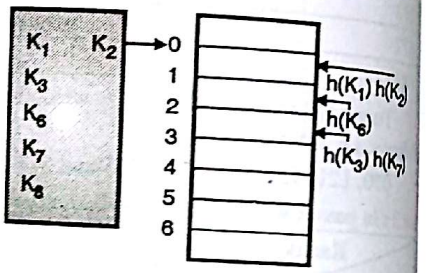
Ans. :

There is an approach, in which we compute the location of the desired record in order to retrieve it in a single access. This avoids the unnecessary comparison. In this method, the location of the desired record present in a search table depends only on the given key but not on other keys.

e.g. If we have a table of n students record each of which is defined by the roll number key. The roll number key takes value from 1 to n inclusive. If the roll number is used as an index into the student table, we can directly find the information of students in question. Therefore, array can be used to organize records in such a search table.

Concept of hash table :

A hash table is a data structure where we store a key value after applying the hash function. It is arranged in the form of an array that is addressed via a hash function. The hash table is divided into a number of buckets is in turn capable of storing a number of records. Thus we can say a bucket has number of slots and each slot is capable of holding one record.



The time required to locate any element in the hash table 0 (1). It is constant and it is not depend on number of data elements stored in table.

Collision : It is possible that two different keys k_1 and k_2 give same hash address this situation is called as collision. The methods use to solve this problem is called as collision resolution. Two basic types of collision resolution are as follows,

Linear probing : This is basic and simple method to resolve the collision. Here a collision occurs at index end then you check the further location n + 1, n + 2 and so on, until you find the vacant location or position. Linear probing is simple collision resolution technique which is easy to implement but it have disadvantage of clustering.

called clustering is occurred when we use linear probing method to resolve the ment may get crowded in particular section and because of this it the one of the main disadvantage of linear probing.

of the methods which is used to resolve collision. Here if collision heck at $n+1^2, n+2^2$, ... so on i.e. $n + i^2$ for vacant location. There is no probing. Hence fficient method than the linear probing.

Given data :

Hash table size 10.

99, 33, 23, 44, 56, 43 and 19

This keys are placed by using Folding Method .

1. key = 99

   $H = K_1 + K_2 + ... + K_n$

   $H = 9 + 9 = 18$

   Ignore Carry $\therefore$ $\boxed{H = 8}$

   i.e. we store key 99 at hash address 8.

2. key = 33

   $H = K_1 + K_2$

   $H = 3 + 3 = 6$

   i.e. we store key 33 at hash address 6.

3. key = 23

   $H = K_1 + K_2$

   $H = 2 + 3 = 5$

   i.e. we store key 23 at hash address 5.

4. key = 44

   $H = K_1 + K_2 = 4 + 4 = 8$

   but we cant store 44 at hash address 8. Because of Collision. According to linear and quadratic probing 4 stored at 9th place.

5. key = 56

   $H = K_1 + K_2 = 5 + 6 = 11$

   Ignore the carry $\boxed{H = 1}$

   $\therefore$ Key 56 stored at hash address H = 1.

6. key = 43

   $H = K_1 + K_2 = 4 + 3 = 7$

   $\therefore$ Key 43 stored at hash address 7.

7. key = 19

   $H = K_1 + K_2 = 1 + 9 = 10$

   Ignore the carry $\therefore$ $\boxed{H = 0}$

   $\therefore$ Key 19 stored at hash address 0.

The hash table address is as follows :

0	99
1	56
2	
3	
4	
5	23
6	33
7	43
8	99
9	44

→ Linear probing and Quadratic probing method gives same answer for key = 44 to resolve the collision. 44 % 9

Chapter No.	May 2019
Chapter 1	10 Marks
Chapter 2	–
Chapter 3	10 Marks
Chapter 4	–
Chapter 5	20 Marks
Chapter 6	10 Marks
Chapter 7	–
Chapter 8	35 Marks
Chapter 9	15 Marks
Chapter 10	20 Marks
Repeated Questions	–

Chapter 1 : Types of Data

Q. 1(a) Explain different types of data

Ans. :

Types of data structures :

1. **Primitive data structure :**

The integers, reals, logical data structures. These data types are available of primitive data types can be represented

2. **Non-primitive data structure :**

These data structures are derived heterogeneous data elements are stored queue, tree, graph.

3. **Linear data structure :**

Data Structures

Statistical Analysis

Chapter No.	May 2014	Dec. 2014	May 2015	Dec. 2015	May 2016
Chapter 1	10 Marks	05 Marks	15Marks	–	03 Marks
Chapter 2	–	–	–	–	–
Chapter 3	10 Marks	–	–	–	–
Chapter 4	–	07 Marks	05 Marks	–	10 Marks
Chapter 5	20 Marks	25 Marks	15 Marks	10 Marks	17 Marks
Chapter 6	10 Marks	10 Marks	16 Marks	20 Marks	18 Marks
Chapter 7	–	11 Marks	10 Marks	10 Marks	12 Marks
Chapter 8	35 Marks	25 Marks	21 Marks	60 Marks	17 Marks
Chapter 9	15 Marks	05 Marks	05 Marks	–	10 Marks
Chapter 10	20 Marks	18 Marks	33 Marks	10 Marks	25 Marks
Repeated Questions	–	03 Marks	05 Marks	10 Marks	15 Marks

May 2014

Chapter 1 : Types of Data Structure and Recursion [Total Marks : 10]

Q. 1(a) Explain different types of data structures with example. **(5 Marks)**

Ans. :

Types of data structures :

1. Primitive data structure :

The integers, reals, logical data, character data, pointers and reference are primitive data structures. These data types are available in most programming languages as built in type. Data objects of primitive data types can be operated upon by machine level instructions.

2. Non-primitive data structure :

These data structures are derived from primitive data structures. A set of homogeneous and heterogeneous data elements are stored together. **Examples :** Array, structure, union, linked-list, stack, queue, tree, graph.

3. Linear data structure :

Elements are arranged in a linear fashion (one dimension). All one-one relation can be handled through linear data structures. Lists, stacks and queues are examples of linear data structure.

4. Non-linear data structure :

All one-many, many-one or many–many relations are handled through non-linear data structures. Every data element can have a number of predecessors as well as successors. Tree graphs and tables are examples of non-linear data structures.

5. Static data structure :

In case of static data structure, memory for objects is allocated at the time of loading of the program. Amount of memory required is determined by the compiler during compilation.

Example : int a[50]

Static data structure causes under utilization of memory (in case of over allocation). Static data structure may cause overflow (under allocation). No reusability of allocated memory.

6. Dynamic data structure :

In case of dynamic data structures, the memory space required by variables is calculated and allocated during execution. Dynamic memory is managed in 'C' through a set of library function.

Allocating a block of memory in "C" :

ptr = (cast – type) malloc (byte – size) ;

The "malloc ( )" returns a pointer (of cast type) to an area of memory with size, byte – size

Q. 1 (b) Write recursive and non-recursive functions to calculate GCD of 2 numbers. **(5 Marks)**

Ans. :

Non-recursive function to calculate GCD :

```
int GCD(int x, int y)
{  int temp;
   if(x < y) /* y intechange x and y if x < y y */
       { temp = x;
          x = y;
       }
   if(x % y) = = 0
          return(y);
   return(GCD(y, x % y));
}
```

Recursive function to calculate GCD :

```
int Power(int x, int n)
{  if(n = =0)
   return(1);
   return(x *(Power(x, n – 1));
}
```

Chapter 3 : Pointers and Structures in C [Total Marks : 10]

Q. 3 (b) Write a program in 'C' which will read a text and count all occurrences of a particular word. **(10 Marks)**

Ans. :

Program in 'C' which will read a text and count all occurrences of a particular word :

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



```
void main()
{
char a[80];
int words,special,characters,i;
words=special=characters=0;
printf("\n Enter the string :");
gets(a);
for(i=0;a[i]!='\0';i++)
{
   characters++;
   if(isalnum(a[i]) && (i==0 || !isalnum(a[i-1])))
        words++;
   if(!isspace(a[i]) && !isalnum(a[i]))
        special++;
}
printf("\n No of characters = %d",characters);
printf("\n No of words = %d",words);
printf("\n No of special characters = %d",special);
}
```

Output

```
Enter the string :India,China and USA are big countries.
No of characters = 38
No of words = 7
No of special characters = 2
```

Chapter 5 : Linked List [Total Marks : 20]

Q. 4 (a) Write a program in 'C' to implement circular queue using link-list. **(10 Marks)**

Ans. :

Program in 'C' to implement circular queue using linked list :

```
//*********To implement circular queue using linked list.*******//
#include<stdio.h>
#include<conio.h>
typedef struct node
{
 int data;
 struct node *next;
}node;
void init(node **R);
void enqueue(node **R,int x);
int dequeue(node **R);
int empty(node *rear);
void print(node *rear);
void main()
```

```
{
 int x,option;
 int n = 0,i;
 node *rear;
 init(&rear);
 clrscr();
 do
 {
  printf("\n 1. Insert\n 2. Delete\n 3. Print\n 4. Quit");
  printf("\n your option:    ");
  scanf("%d",&option);
  switch(option)
  {
    case 1 :
     printf("\n Number of Elements to be inserted");
     scanf("%d",&n);
     for(i=0;i<n;i++)
     {
       scanf("\n %d",&x);
       enqueue(&rear,x);
     }
     break;
    case 2 : if(! empty(rear))
       {
    x = dequeue(&rear);
    printf("\n Element deleted = %d",x);
       }
       else
    printf("\n Uderflow..... Cannot deleted");
       break;
    case 3 : print(rear);
     break;
  }
 }while(option != 4);
 getch();
 }
 void init(node **R)
 {
  *R = NULL;
 }
 void enqueue(node **R,int x)
 {
  node *p;
  p = (node *)malloc(sizeof(node));
```



```
  p->data = x;
  if(empty(*R))
  {
     p->next = p;
     *R = p;
  }
  else
  {
     p->next = (*R)->next;
     (*R)->next = p;
     (*R) = p;
  }
 }
int dequeue(node **R)
{
  int x;
  node *p;
  p = (*R)->next;
  p->data = x;
  if(p->next == p)
  {
     *R = NULL;
     free(p);
     return(x);
  }
  (*R)->next = p->next;
  free(p);
  return(x);
}
void print(node *rear)
{
    node *p;
    if(!empty(rear))
    {
      p = rear->next;
    }
    p = p->next;
    do
    {
      printf("\n %d",p->data);
      p = p->next;
    }while(p != rear->next);
 }
 int empty(node *P)
```

```
{
if(P->next== -1)
    return(1);
return(0);
}
```

Output :

```
 1. Insert
 2. Delete
 3. Print
 4. Quit
 your option:    1

 Element to be inserted  4
        12  23   34   45
 1. Insert
 2. Delete
 3. Print
 4. Quit
 your option:    3
            12   23   34       45
 1. Insert
 2. Delete
 3. Print
 4. Quit
 your option:    2
 Element deleted = 4
 1. Insert
 2. Delete
 3. Print
 4. Quit
 your option:    3
             23   34   45
 1. Insert
 2. Delete
 3. Print
 4. Quit
 your option:    4
```

Q. 5 (a) Write a program in 'C' to implement doubly Link-list with methods insert, delete and search. **(10 Marks)**

Ans. :

Program in C to implement doubly link list :

```
/*Doubly Linked List :*/
#include <stdio.h>
#include <conio.h>
```



```
typedef struct dnode
 {
   int data;
   struct dnode *next,*prev;
 }dnode;
dnode *create()
 {
   int i,n,x;
   dnode *head,*p,*q;
   head=NULL;
   printf("\nEnter no. of data : ");
   scanf("%d",&n);
   printf("\n enter data : ");
   for(i=1;i<=n;i++)
    {
 printf("\nEnter next data : ");
 scanf("%d",&x);
 q=(dnode*)malloc(sizeof(dnode));
 q->data=x;
 q->next=q->prev=NULL;
 if(head==NULL)
  {
    p=head=q;
  }
 else
  {
  p->next=q;
  q->prev=p;
  p=q;
  }
   }
  return(head);
 }
  void display(dnode *head)
    {
 printf("\n");
 for(;head!=NULL;head=head->next)
      printf("%d   ",head->data);
    }
  dnode *Delete(dnode *head,int x)
    {
  dnode *p,*q;

  if(head==NULL)
```

```
    return(head);
  if(head->data==x)
   {
     p=head;
     head=head->next;
     head->prev=NULL;
     free(p);
     return(head);
   }
  p=head;
  while(p!=NULL && p->data !=x)
     p=p->next;
  if(p!=NULL)
   {
     if(p->next==NULL)
     {
       p->prev->next=NULL;
       free(p);
     }
    else
     {
       p->prev->next=p->next;
       p->next->prev=p->prev;
       free(p);
     }
    }
  return(head);
   }
int search(dnode *head, int x)
  {
    while(head!=NULL)
{
  if(head->data==x)
    return (1);
  head=head->next;
}
    return(0);
  }
void main()
{
   dnode *head;
   int x;
   head=create();
   printf("\nEnter the data to be searched : ");
```



```
    scanf("%d",&x);
    if(search(head,x))
printf("\nfound");
    else
printf("\nNot found");
    printf("\nEnter the data to be deleted : ");
    scanf("%d",&x);
    head=Delete(head,x);
    printf("\nLinked list after deletion : ");
    display(head);
    getch();
  }
```

Chapter 6 : Stack [Total Marks : 10]

Q. 2 (b) Write a program in 'C' to convert infix expression to postfix expression using stacks. **(10 Marks)**

Ans. :

Program in 'C' to convert infix to postfix :

```
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
#define MAX 50
typedef struct stack
{
int data[MAX];
int top;
}stack;

int  precedence(char);
void init(stack *);
int  empty(stack *);
int  full(stack *);
int  pop(stack *);
void push(stack *,int );
int  top(stack *); //value of the top element
void infix_to_postfix(char infix[],char postfix[]);

void main()
 { char infix[30],postfix[30];
   clrscr();
   printf("\nEnter an infix expression : ");
   gets(infix);
   infix_to_postfix(infix,postfix);
   printf("\nPostfix : %s  ",postfix);
```

```
}
void infix_to_postfix(char infix[],char postfix[])
{  stack s;
char x;
int i,j;//i-index for infix[],j-index for postfix
char token;
init(&s);
j=0;
for(i=0;infix[i]!='\0';i++)
 {    token=infix[i];
   if(isalnum(token))
        postfix[j++]=token;
   else
        if(token == '(')
            push(&s,'(');
        else
            if(token == ')')
              while((x=pop(&s))!='(')
                  postfix[j++]=x;
            else
            {
                while(precedence(token)<=precedence(top(&s)) && !empty(&s))
                {
                    x=pop(&s);
                    postfix[j++]=x;
                }
                push(&s,token);
            }
}
while(!empty(&s))
{
   x=pop(&s);
   postfix[j++]=x;
}
postfix[j]='\0';
}
                x)
                                    return(0);
                                    return(1);
 || x == '/' || x == '%') return(2);
(3);
```

```
void init(stack *s)
{
s->top=-1;
}

int empty(stack *s)
{
if(s->top==-1)    return(1);
return(0);
}
int full(stack *s)
{
if(s->top==MAX-1)    return(1);
return(0);
}

void push(stack *s,int x)
{
s->top=s->top+1;
s->data[s->top]=x;
}

int pop(stack *s)
{
int x;
x=s->data[s->top];
s->top=s->top-1;
return(x);
}

int top(stack * p)
{
return(p->data[p->top]);
}
```

Output

```
Enter infix expression:a*(b+c)/d+g
abc+*d/g+
```

Chapter 8 : Trees [Total Marks : 35]

Q. 1(d) Discuss practical application of trees. **(5 Marks)**

Ans. :

Practical application of trees :

When an expression is represented through a tree, it is known as an expression tree. The leaves of an expression tree are operands, such as constants or variables names and all internal nodes contain operations. Fig. 1 gives an example of an expression tree.

(a + b * c) * e + f

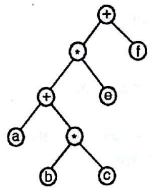


Fig. 1

A preorder traversal on the expression tree gives prefix equivalent of the expression. A postorder traversal on the expression tree gives postfix equivalent of the expression.

Prefix (expression tree of Fig. 1) = + * + a * b c e f

Postfix (expression tree of Fig. 1) = a b c * + e * f +

Q. 4 (b) Construct binary tree for the pre order and inorder traversal sequences : **(10 Marks)**

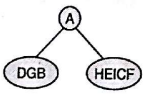
Preorder :	A	B	D	G	C	E	H	I	F
Inorder :	D	G	B	A	H	E	I	C	F

Ans. :

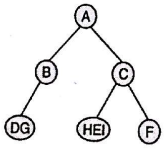
Preorder sequence : A B D G C E H I F

Inorder sequence : D G B A H E I C F

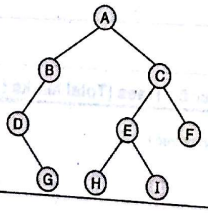
Step 1 :



Step 2 :



Step 3 :



Q. 6(a) Write short note on discuss threaded binary tree in detail. **(10 Marks)**

Ans. :

Threaded binary tree :

In a linked representation of a binary tree, there are more null links than actual pointers. These null links can be replaced by pointers, called threads to other nodes. A left null link of a node is replaced with the address of its inorder predecessor. Similarly, a right null link of a node is replaced with the address of its inorder successor.

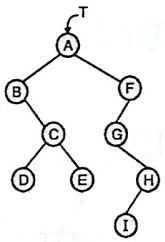


Fig. 2 : A sample tree before threading

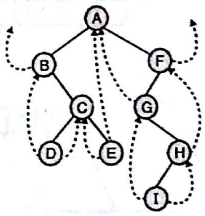


Fig. 2(a) : Tree of Fig. 2 after threading

The tree T of Fig. 2(a) has 9 nodes and 10 null links, which have been replaced by thread links. If we traverse the tree in inorder the nodes will be visited in the order BDCEAGIHF. Consider a node D. The left null link of D is replaced with a the link pointing to its inorder predecessor. The right null link of D is replaced with a thread link pointing to its inorder successor.

In the memory representation of a tree node we must be able to distinguish between threads and normal pointers. This can be done by adding two extra fields *l*bit and rbit.

*l*bit of a node = 1 left child is normal

*l*bit of a node = 0 left link is replaced with a thread

rbit of a node = 1 right child is normal

rbit of a node = 0 right link is replaced with a thread

In the Fig. 2(a), two threads have been left dangling. Node B has no inorder predecessor and the node F has no inorder successor. This problem can be solved by taking a head node.

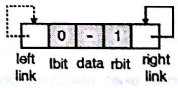


Fig. 2(b) : Initial status of head node

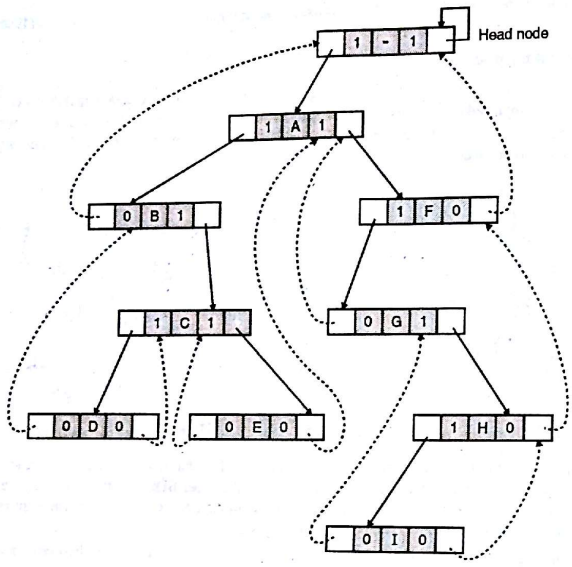


Fig. 2(c) : Memory representation of the TBT of Fig. 2(b)

C function for TBT :

```
typedef struct TBTnode
{
  char data;
  struct TBTnode * left;
  struct TBTnode  * right;
  int lbit, rbit;
} TBT node;
```

Q. 5 (b) Write a program in 'C' to implement binary search on sorted set of integers. **(10 Marks)**

Ans. :

Program in C to implement binary search :

```
//*****To implement binary search :*****//

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



```
//int stepcount=0,swapcount=0,compcount=0;

int binsearch(int a[],int i ,int j,int key);//Recursive
void main()
 {
   int a[30],key,n,i,result;
   clrscr();
   printf("\n Enter No. of elements : ");
   scanf("%d" ,&n);
   printf("\n Enter a sorted list of %d elements : ",n);
   for(i=0;i<n;i++)
scanf("%d",&a[i]);
   printf("\n Enter the element to be searched : ");
   scanf("%d",&key);
   result=binsearch(a,0,n-1,key);
   if(result==-1)
printf("\n Not found ");
   else
printf("\n Found at location= %d",result+1);

/* printf("\n No. of steps= %d",stepcount);
   printf("\n No. of swaps= %d", swapcount);
   printf("\n No. of comparisons= %d",compcount);*/
   getch();
 }

int binsearch(int a[],int i, int j,int key)
{
   int c;
   if(i>j)
{
//  compcount++;
  //stepcount+=2;
  return(-1);
}
   c=(i+j)/2;
   if(key==a[c])
{
  compcount++;
  stepcount+=2;
  return(c);
}
   if(key>a[c])
{
```

```
// stepcount++;
 return(binsearch(a,c+1,j,key));
}
 // stepcount++;
 return(binsearch(a,i,c-1,key));
}
```

Output :

```
Enter No. of elements : 5

Enter a sorted list of 5 elements : 2
4
1
9
7

Enter the element to be searched : 9

Found at location= 4
```

Chapter 9 : Graphs [Total Marks : 15]

Q. 1(c) Show with example how graphs are represented in computer memory. **(5 Marks)**

Ans. :

Representation of graph :

1. Adjacency matrix :

A two dimensional matrix can be used to store a graph. A graph G = (V, E) when V = {0, 1, 2, ... n – 1} can be represented using a two dimensional integer array of size n × n.

int adj[20][20]; can be used to store a graph with 20 vertices.

adj[i][j] = 1, indicates presence of edge between two vertices i and j

= 0, indicates absence of edge between two vertices i and j

Adjacency matrix of an undirected graph is always a symmetric matrix, i.e. an edge (i, j) implies the edge (j, i). Adjacency matrix of a directed graph is never symmetric adj[i][j] = 1, indicates a directed edge from vertex i to vertex j.

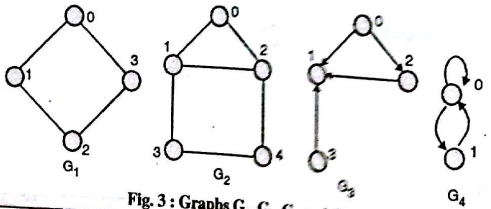


Fig. 3 : Graphs G_1, G_2, G_3 and G_4

	0	1	2	3
0	0	1	0	1
1	1	0	1	0
2	0	1	0	1
3	1	0	1	0

G_1 (Undirected graph)

	0	1	2	3	4
0	0	1	1	0	0
1	1	0	1	1	0
2	1	1	0	0	1
3	0	1	0	0	1
4	0	0	1	1	0

G_2 (Undirected graph)

	0	1	2	3
0	0	1	1	0
1	0	0	0	0
2	0	1	0	0
3	0	1	0	0

G_3 (Directed graph)

	0	1
0	1	1
1	1	0

G_4 (With self loop)

Fig. 3(a) : Adjacency matrix representation of graphs G_1, G_2, G_3 and G_4 of Fig. 3

2. Adjacency list :

A graph can be represented using a linked list. For each vertex, a list of adjacent vertices is maintained using a linked list. It creates a separate linked list for each vertex V_i in the graph G = (V, E)

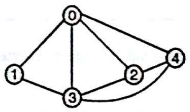


Fig. 3(b) : A graph

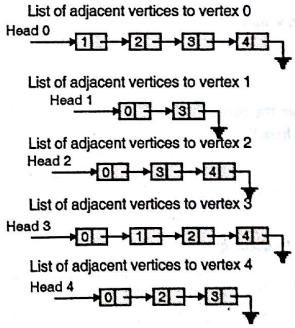


Fig. 3(c) : Adjacency list for each vertex of graph of Fig. 3(b)

Adjacency list of a graph with n nodes can be represented by an array of pointers. Each pointer points to a linked list of the corresponding vertex. Fig. 3(d) shows the adjacency list representation of graph of Fig. 3(b).

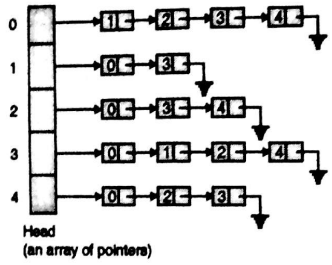


Fig. 3(d) : Adjacency list representation of the graph of Fig. 3(b)

Adjacency list representation of a graph is very memory efficient when the graph has a large number of vertices but very few edges.

Q. 8 (b) Explain BFS algorithm with example. **(10 Marks)**

Ans. :

BFS algorithm :

```
/* Array visited[] is initialize to 0 */
/* BFS traversal on the graph G is carried out beginning at vertex V */
void BFS(int V)
{
   q : a queue type variable;
   initialize q;
   visited[v] = 1;      /* mark v as visited */
   add the vertex V to queue q;
   while(q is not empty)
   {
       v ← delete an element from the queue;
       for all vertices w adjacent from V
       {
           if(!visited[w])
           {
               visited[w] = 1;
               add the vertex w to queue q;
           }
       }
   }
}
```


BFS example :

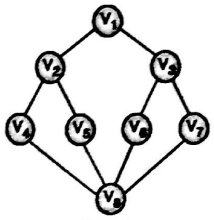


Fig. 4

Queue	Visited[ ] (1 2 3 4 5 6 7 8)	Vertex visited	Action
NULL	0 0 0 0 0 0 0 0	–	–
V_1	1 0 0 0 0 0 0 0	V_1	add (q, V_1) visit (V_1)
$V_2 V_3$	1 1 1 0 0 0 0 0	$V_1 V_2 V_3$	delete (q), add and visit adjacent vertices
$V_3 V_4 V_5$	1 1 1 1 1 0 0 0	$V_1 V_2 V_3 V_4 V_5$	delete (q), add and visit adjacent vertices
$V_4 V_5 V_6 V_7$	1 1 1 1 1 1 1 0	$V_1 V_2 V_3 V_4 V_5 V_6 V_7$	delete (q), add and visit adjacent vertices
$V_5 V_6 V_7 V_8$	1 1 1 1 1 1 1 1	$V_1 V_2 V_3 V_4 V_5 V_6 V_7 V_8$	delete (q), add and visit adjacent vertices
$V_6 V_7 V_8$	1 1 1 1 1 1 1 1	$V_1 V_2 V_3 V_4 V_5 V_6 V_7 V_8$	delete (q)
$V_7 V_8$	1 1 1 1 1 1 1 1	$V_1 V_2 V_3 V_4 V_5 V_6 V_7 V_8$	delete (q), add and visit adjacent vertices
V_8	1 1 1 1 1 1 1 1	$V_1 V_2 V_3 V_4 V_5 V_6 V_7 V_8$	delete (q)

Queue	Visited[ ]	Vertex visited	Action
NULL	1 1 1 1 1 1 1 1	$V_1\ V_2\ V_3\ V_4\ V_5\ V_6\ V_7\ V_8$	algorithm terminates as the queue is empty

Chapter 10 : Searching and Sorting [Total Marks : 20]

Q. 2 (a) What is hashing? What is mean by collision? Using modulo division method and linear probing, store the values given below in array with 10 elements.

99 33 23 44 56 43 19. **(10 Marks)**

Ans. :

Hashing :

Hashing is a away with the requirement of keeping data sorted (as in binary search). Its best case timing complexity is of constant order (O(1)). In its worst case, hashing algorithm starts behaving like linear search.

Best case timing behaviour of searching using hashing = O(1)

Worst case timing Behaviour of searching using hashing = O(n).

Collision :

Collision resolution is the main problem in hashing. If the element to be inserted is mapped to the same location, where an element is already inserted then we have a collision and it must be resolved. There are several strategies for collision resolution. The most commonly used are :

(a) Separate chaining – used with open hashing :

In this strategy, a separate list of all elements mapped to the same value is maintained.

Separate chaining is based on collision avoidance.

If memory space is tight, separate chaining should be avoided.

Additional memory space for links is wasted in storing address of linked elements.

Hashing function should ensure even distribution of elements among buckets, otherwise the timing behavior of most operations on hash table will deteriorate.

(b) Open addressing – used with closed hashing :

Separate chaining requires additional memory space for pointers. Open addressing hashing is an alternate method of handling collision. In open addressing, if a collision occurs, alternate cells are tried until an empty cell is found. Because all the data elements are stored inside the table, a larger memory space is needed for open addressing. Generally, the load factor should be below 0.5 for open addressing hashing. There are three commonly used collision resolution strategy in open addressing.

(1) Linear probing

(2) Quadratic probing

(3) Double hashing.

Example :

1. Linear probing :

	Empty table	After 99	After 33	After 23	After 44	After 56	After 43	After 19	* = Collision No. of collisions = 4
0								19*	
1									
2									
3			33	33	33	33	33	33	
4				23*	23*	23*	23*	23*	
5					44*	44*	44*	44*	
6						56	56	56	
7							43*	43*	
8									
9		99	99	99	99	99	99	99	

2. Quadratic probing :

	Empty table	After 99	After 33	After 23	After 44	After 56	After 43	After 19	* = Collision No. of collisions = 4
0								19*	
1									
2									
3			33	33	33	33	33	33	
4				23*	23*	23*	23*	23*	
5					44*	44*	44*	44*	
6						56	56	56	
7							43*	43*	
8									
9		99	99	99	99	99	99	99	

Q. 3 (a) Write a program in 'C' to perform quick sort. Show steps with example. **(10 Marks)**

Ans. :

Program in C to perform quick sort :

```
#include<conio.h>
#include<stdio.h>
void quick_sort(int[],int,int);
int partition(int[],int,int);
void main()
```

```
{
int a[30],n,i;
printf("\nEnter no of elements :");
scanf("%d",&n);
printf("\nEnter array elements :");
for(i=0;i<n;i++)
    scanf("%d",&a[i]);
    quick_sort(a,0,n-1);
    printf("\nSorted array is :");
for(i=0;i<n;i++)
    printf("%d",a[i]);
getch();
}
void quick_sort(int a[],int l,int u)
{
int j;
if(l<u)
{
    j=partition(a,l,u);
    quick_sort(a,l,j-1);
    quick_sort(a,j+1,u);
}
}
int partition(int a[],int l,int u)
{
   int v,i,j,temp;
   v=a[l];
   i=l;
   j=u+1;
   do
   {
do
   i++;
while(a[i]<v && i<=u);

do
   j--;
while(v<a[j]);
if(i<j)
{
   temp=a[i];
   a[i]=a[j];
   a[j]=temp;
}
```



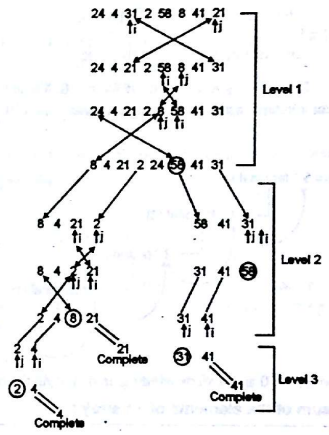
```
   }while(i<j);
   a[l]=a[j];
   a[j]=v;
   return(j);
}
```

Output

```
Enter no of elements  :      4
Enter array elements  :      23   1     55    33
Sorted array is  : 1     23    33  55
```

Example for showing steps :

Quick sort method 24, 4, 31, 2, 58, 8, 41, 21 :



Dec. 2014

Chapter 1 : Types of Data Structure and Recursion [Total Marks : 05]

Q. 1 (a) What is recursion? Write a 'C' program to calculate sum of 'n' natural numbers using recursion. **(5 Marks)**

Ans. :

Recursion :

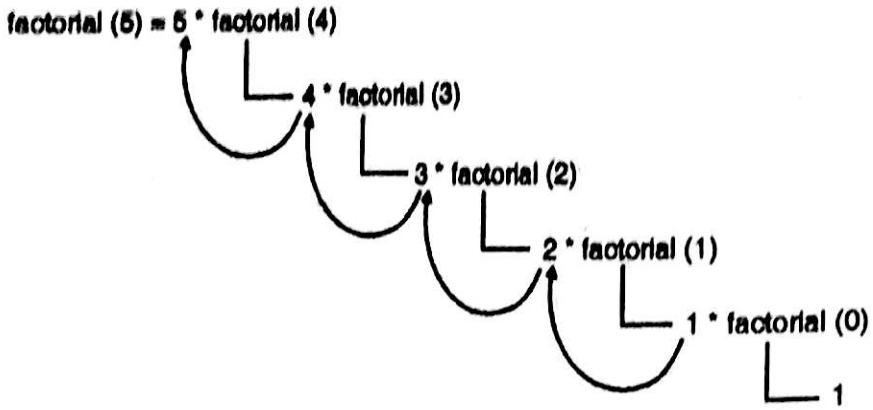
Recursion is a fundamental concept in mathematics. When a function is defined in terms of itself then it is called a recursive function.

Consider the definition of factorial of a positive integer n.

$$\text{Factorial (n)} = \begin{cases} 1 & \text{if (n=0)} \\ n * \text{factorial (n - 1)}, & \text{otherwise} \end{cases}$$

Function "factorial( )" is defined in terms of itself for $n > 0$. Value of the function at $n = 0$ is 1 and it is called the base. Recursion terminates on reaching the base.

Example :



Recursion expands when $n > 0$ and it starts winding up on hitting the base ($n = 0$).

'C' function for finding sum of the elements of an array :

```
int sum( int a[ ], int n)
{
if(n == 1)
   return( a[0]);
else
   return(a[n - 1] + sum(a, n - 1));
}
```

Chapter 4 : File Handling in C [Total Marks : 07]

Q. 2 (b) What are different types of files? Explain various file handling operations in 'C'. **(7 Marks)**

4. fprintf() and fscanf()

fprintf() is similar to printf(). printf() sends the output to monitor. Whereas, fprintf() sends the output to a file. Similarly, scanf(), reads input from the keyboard. Whereas, fscanf() reads the input from a file.

5. getw() and putw() :

These two functions are for reading and writing of integer values. The general form of getw and putw are :

```
putw(15, fp)      ;    [write 15 to a file pointed by fp]
putw(x, fp)       ;    [ write x to a file pointed by fp]
x = getw(fp)      ;    [read a number from a file pointed by fp and store it in x]
```

Chapter 5 : Linked List [Total Marks : 25]

Q. 3 (b) Write a 'C' program to implement a singly Linked List which supports the following operations :

(i) Insert a node in the beginning
(ii) Insert a node in the end
(iii) Insert a node after a specific node
(iv) Deleting a specific node
(v) Displaying the list. **(10 Marks)**

Ans. :

```
/*Operations on SLL(singly linked list) */
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
typedef struct node
  { int data;
    struct node *next;
  }node;
node *create();
node *insert_b(node *head,int x);
node *insert_e(node *head,int x);
node *insert_in(node *head,int x);
node *delete_b(node *head);
node *delete_e(node *head);
node *delete_in(node *head);
node *reverse(node *head);
void search(node *head);
void print(node *head);
node *copy(node *);
int count(node *);
node * concatenate(node *, node *);
```

[illegible]


```
    }
p->next=(node*)malloc(sizeof(node));
p=p->next;
    }
    p->next=NULL;
    scanf("%d",&(p->data));
  }
return(head);
}
node *insert_b(node *head,int x)
{  node *p;
   p=(node*)malloc(sizeof(node));
   p->data=x;
   p->next=head;
   head=p;
   return(head);
}
node *insert_e(node *head,int x)
{  node *p,*q;
   p=(node*)malloc(sizeof(node));
   p->data=x;
   p->next=NULL;
   if(head==NULL)
     return(p);
   //locate the last node
   for(q=head;q->next!=NULL;q=q->next);
   q->next=p;
   return(head);
}
node *insert_in(node *head,int x)
{  node *p,*q;
   int y;
   p=(node*)malloc(sizeof(node));
   p->data=x;
   p->next=NULL;
   printf("\n Insert after which number ? : ");
   scanf("%d",&y);
   //locate the the data 'y'
   for(q=head ; q != NULL && q->data != y ; q=q->next);
   if(q!=NULL)
     {
```

```
p->next=q->next;
q->next=p;
    }
   else
     printf("\n Data not found ");
  return(head);
}
node *delete_b(node *head)
{
 node *p,*q;
 if(head==NULL)
   {
printf("\n Underflow....Empty Linked List");
return(head);
   }
 p=head;
 head=head->next;
 free(p);
 return(head);
}
node *delete_e(node *head)
{
 node *p,*q;
 if(head==NULL)
   {
printf("\n Underflow....Empty Linked List");
return(head);
   }
 p=head;
 if(head->next==NULL)
   { // Delete the only element
     head=NULL;
     free(p);
     return(head);
   }
//Locate the last but one node
  for(q=head;q->next->next !=NULL;q=q->next)
  p=q->next;
  q->next=NULL;
  free(p);
  return(head);
```



```
}
node *delete_in(node *head)
{
 node *p,*q;
 int x,i;
 if(head==NULL)
   {
printf("\n Underflow....Empty Linked List");
return(head);
   }
 printf("\n Enter the data to be deleted : ");
 scanf("%d",&x);
 if(head->data==x)
   { // Delete the first element
     p=head;
     head=head->next;
     free(p);
     return(head);
   }
//Locate the node previous to one to be deleted
  for(q=head;q->next->data!=x && q->next !=NULL;q=q->next )
    if(q->next==NULL)
    {
     printf("\n Underflow.....data not found");
     return(head);
    }
  p=q->next;
  q->next=q->next->next;
  free(p);
  return(head);
}
void print(node *head)
{ node *p;
 printf("\n\n");
 for(p=head;p!=NULL;p=p->next)
  printf("%d  ",p->data);
}
node *reverse(node *head)
 { node *p,*q,*r;
   p=NULL;
   q=head;
```

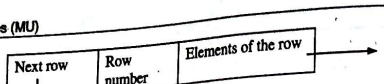
```
    if(h2==NULL)
  return(h1);
    p=h1;
    while(p->next != NULL)//go to the end of the ist linked list
  p=p->next;
    p->next=h2;
    return(h1);
  }
void split(node *h1)
 {
   node *p,*q,*h2;
   printf("\n Linked list to be split : ");
   print(h1);
   //linked list will be broken from the centre using the pointers p and q
   if(h1==NULL)
    return;
   p=h1;
   q=h1->next;
   while(q!=NULL && q->next != NULL)
    {
 q=q->next->next;
 p=p->next;//When q reaches the last node,p will reach the centre node
    }
  h2=p->next;
  p->next=NULL;
  printf("\nFirst half : ");
  print(h1);
  printf("\nSecond half : ");
  print(h2);
 }
```

Q. 5(a) Discuss how memory allocation for a sparse matrix can be optimized using a linked list. Write a C-program for the same. **(15 Marks)**

Ans. :

Memory allocation for a sparse matrix :

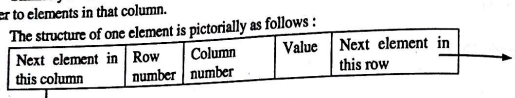
A sparse matrix has many zero entries. In representing a sparse matrix, only non-zero elements are stored to save on space and computing time. Each non-zero element is represented by : row, column and value. Unlike array representation, the matrix is not thought of as a chain of such triplets. Instead, consider "head" nodes for each row and each column pointing to the elements in a particular row or column. A head node of a row contains three parts as shown below :



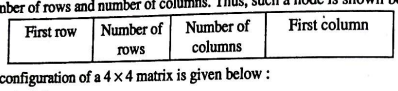
"Row number" indicates the row to which the "head" node is pointing to by the component "element". This "head" node also points to another "head" node for the next row.

Similarly a "column" consists of its column number, a pointer pointing to the next column and a pointer to elements in that column.

The structure of one element is pictorially as follows :



A sparse matrix, therefore, can be defined as a node having two pointers one pointing to the list of rows and the other pointing to the list of columns. In addition, this node should contain two integers specifying the number of rows and number of columns. Thus, such a node is shown below:



The initial configuration of a 4 × 4 matrix is given below :

$$A = \begin{vmatrix} 5 & 0 & 9 & 0 \\ 0 & 0 & 8 & 1 \\ 9 & 0 & 0 & 7 \\ 6 & 5 & 2 & 0 \end{vmatrix}$$

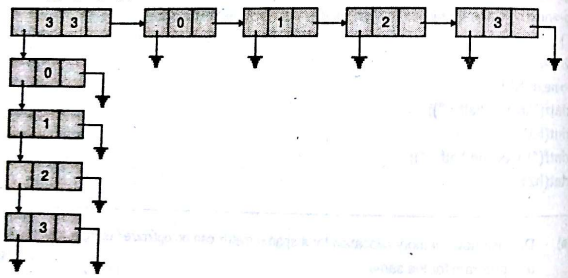


Fig. 1 : Initial configuration

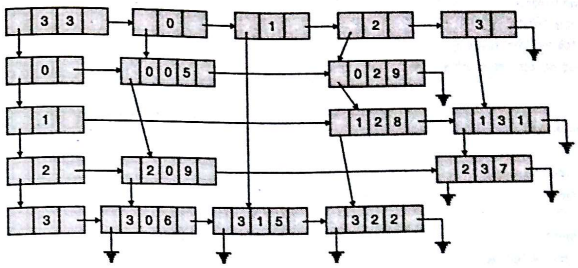


Fig. 1(a) : The linked list representation of the sparse matrix A

Elements in each row are arranged in increasing order of their column numbers. Similarly the elements in each column are arranged in increasing order of their row numbers.

In order to create a linked list version of a sparse matrix, the elements of the matrix represented as triplets of (row, column, value) must be inserted repeatedly to a partially constructed sparse matrix by linked list.

Sparse matrix can be optimized using a linked list :

```
/* Program to create a linked list representation of the sparse matrix */

#include<conio.h>
#include<iostream.h>

struct element
{  int i,j,val;
   element *right,*down;
};

struct row
{  int row_number;
   element *right;
   row *nextrow;
};

struct column
{  int column_number;
   element *down;
   column *nextcolumn;
};
```

```
struct sparse
{ row *firstrow;
  column *firstcolumn;
  int no_ofrows,no_ofcolumns;
};

void main()
{
int i,m,n,r,x,y,val;
sparse S;
row *R;
column *C;
element *e,*ec,*er;
cout<<"\n Enter the size of matrix & no of non zero elements :";
cin>>m>>n>>r;
/* create initial configuration */
S.no_ofrows=m;
S.no_ofcolumns=n;
S.firstrow=NULL;
S.firstcolumn=NULL;

/* create colums headers */
for(i=0;i<n;i++)
{
  if(S.firstcolumn==NULL)
  {
      S.firstcolumn=new column;
      C=S.firstcolumn;
      C->down=NULL;
      C->nextcolumn=NULL;
      C->column_number=i;
  }
  else
  {
      C->nextcolumn=new column ;
      C=C->nextcolumn;
      C->down=NULL;
      C->nextcolumn=NULL;
      C->column_number=i;
  }
```



```
/* create row headers */
for(i=0;i<m;i++)
{
   if(S.firstrow==NULL)
   {
        S.firstrow=new row;
        R=S.firstrow;
        R->right=NULL;
        R->nextrow=NULL;
        R->row_number=i;
   }
   else
   {
        R->nextrow=new row;
        R=R->nextrow;
        R->right=NULL;
        R->nextrow=NULL;
        R->row_number=i;
   }
}
/* read r no of data and insert them in the
  linked list one by one */
  for(i=0;i<r;i++)
  {
   cout<<"\n Enter row no,column no and the value :";
   cin>>x>>y>>val;
   /* Acquire memory for the new element  */
   e=new element;
   e->i=x;
   e->j=y;
   e->val=val;
   e->right=NULL;
   e->down=NULL;

   //locate the row header
   R=S.firstrow;
   while(R->row_number!=x)
        R=R->nextrow;
   //locate the column header
   C=S.firstcolumn;
   while(C->column_number!=y)
```

```
        C=C->nextcolumn;

    /* insert the elment pointed by e in the row */
    if(R->right==NULL)
    {
        R->right=e;
    }
    else
    {
        er=R->right;
        while(er->right!=NULL && (er->right)->j<y)
        er=er->right;
        e->right=er->right;
        er->right=e;
    }
    //insert the element pointed by e in the column
    if(C->down==NULL)
        C->down=e;
    else
    {
        ec=C->down;
        while(ec->down!=NULL && (ec->down)->i<x)
            ec=ec->down;
        e->down=ec->down;
        ec->down=e;
    }
  }
}
```

Chapter 6 : Stack [Total Marks : 10]

Q. 4 (a) Write a 'C' program to convert a polish notation to reverse polish notation. **(10 Marks)**

Ans. :

Conversion of polish notation to reverse polish notation :

```
/* Conversion of an expression from prefix into postfix */
#include<ctype.h>
#include<stdio.h>
#include<conio.h>
#define MAX 20
#include<string.h>
```



```
void convert_postfix(char x);
char stack[MAX][MAX];
void prefix_to_postfix(char prefix[],char postfix[]);
int top;
/* Array 'stack[][]' is being used as a stack of strings */

void main()
{
char postfix[30],infix[30],prefix[30];
clrscr();
printf("\nEnter a prefix expression :");
gets(prefix);
prefix_to_postfix(prefix,postfix);
printf("\nPostfix :   %s",postfix);
getch();
}
  void prefix_to_postfix(char prefix[],char postfix[])
{
 char x,st1[20];
 int i;
 top=-1;
 for(i=strlen(prefix)-1;i>=0;i--) //scan the prefix string from
  {                               //right to left.
   x=prefix[i];
   if(isalnum(x))
    {
     /* convert token to string form */
     st1[0]=x;
     st1[1]='\0';
     // push the operand on the stack s2
     top=top+1;
     strcpy(stack[top],st1);
    }
  else  // if operator, convert to postfix
     {
      convert_postfix(x);
     }
  }
  //Result is on top of the stack
strcpy(postfix,stack[top]);
}
```

```
void convert_postfix(char x)
{     char st1[30],st2[30];
st2[0]=x;
st2[1]='\0';
strcpy(st1,stack[top]);
strcat(st1,stack[top-1]);
strcat(st1,st2);
top=top-1;
strcpy(stack[top],st1);
}
```

Chapter 7 : Queues [Total Marks : 11]

Q. 1 (c) Discuss in brief any two applications of the queue data structure. **(3 Marks)**

Ans. :

Applications of the queue data structure :

(a) Scheduling of processes (Round robin algorithm).

(b) Spooling (to maintain a queue of jobs to be printed).

(c) A queue of client processes waiting to receive the service from the server process.

Various application software using non-linear data structure tree or graph requires a queue for breadth first traversal.

Simulation of a real life problem with the purpose of understanding its behavior. The probable waiting time of a person at a railway reservation counter can be found through the technique of computer simulation if the following concerned factors are known :

(1) Arrival rate (2) Service time

(3) Number of service counters

Q. 2 (a) Write a 'C' program to implement a priority queue. **(8 Marks)**

Ans. :

C program to implement a priority queue :

```
#include <stdio.h>
#include <conio.h>
#define MAX 30
typedef struct pqueue
{  int data [MAX];
   int rear, front;
} pqueuel
void initialize(pqueue *p);
int empty(pqueue *p);
int full(pqueue *p);
void enqueue(pqueue *p, int x);
```



```
int dequeue(pqueue *p);
void print(pqueue *p);
void main( )
{ int x, op, n;
  pqueue q;
  initialize (&q);
do
{
 printf("\n1)create\n2)insert\n3)Delete\n4)print\n5)Quit");
 printf("\n enter your choice:");
 scanf ("%d", &op);
    switch (op)
{
case 1 : printf("\n enter no. of elements :");
       scanf("%d", &n);
       initialize (&q);
       printf("enter the data:");
       for (i = 0; i < n; i++)
       {
          scanf ("%d" &x);
          if (full (&q))
          {   printf("\n queue is full ...");
              exit(0);
          }
          enqueue (&q, x);
   }
   break;
case 2 :  printf("\n enter the element to be inserted");
       scanf ("%d", &x);
       if (full(&q))
       {
          printf("\n queue is full ...");
          exit(0);
       }
       enqueue (&q, x);
       break;
       case 3 : if (empty (&q))
       {
              printf("\n queue is empty ...");
              exit(0);
       }
```

```
    x = dequeue (&q);
    printf("\n element = %d", x);
    break;
    case 4 : print(&q);
    break;
       default : break;
    }
} while (op! = 5);
}
void initialize (pqueue *p)
{  p -> rear = - 1;
p -> front = - 1;
}
/* A value of rear or front as -1, indicate that the queue is empty. */
int empty (pqueue *p)
{ if (P->rear == - 1)
    return (1);    /* queue is empty */
return (0); /* queue is not empty */
}
int full (pqueue *p)
{ /* if front is next rear in the circular array then the queue is full */
  if (p -> rear + 1)% MAX == p -> front)
    return (1); /* queue is full */
return (0);
}
void enqueue (pqueue *p, int x)
{ int i;
 if (full (p))
    printf("\n overflow ...");
 else
{  /* inserting in an empty queue */
    if (empty (p))
        {   p -> rear = p -> front = 0
            p -> data [0] = x;
        }
    else
    {
    /* move all lower priority data right by one place */
i = p -> rear ;
while (x > p -> data [i])
{  p -> data [(i + 1)%MAX] = p -> data [i];        /* position i on the previous element */
```



```
    i = (i - 1 + MAX) % MAX; /* anticlock wise movement inside the queue */
    if ((i + 1)% MAX = = p -> front)
        break; /* if all elements have been moved */
}
/* insert x */
    i = (i + 1)% MAX;
    p -> data [i] = x;
/* re-adjust rear */
    p -> rear = (p -> rear + 1) % MAX;
}
  }
}
int dequeue (pqueue *p)
{  int x;
if (empty (p))
        printf("\n underflow ...");
    else
    {    x = p -> data [p -> front];
        if (p -> rear = = p -> front) /* delete last element */
            initialize (p);
        else
            p -> front = (p -> front + 1) % MAX;
    }
 return (x);
 }
 void print (pqueue *p)
 {  int i, x;
  i = p -> front;
 while ( i! = p -> rear)
 {  x = p data[i];
        printf ("\n%d", x);
    i = (i + 1) % MAX;
 }
 /* print the last data */
    x = p -> data[i];
    printf ("\n%d", x);
 }
```

Chapter 8 : Trees [Total Marks : 25]

Q. 1 (b) What is a multi way search tree ? Explain with an example. **(5 Marks)**

Ans. :

Multi way search tree :

A B-tree is a M-way tree. An M-way tree can have maximum of M children.

An M-way tree contains multiple keys in a node. This leads to reduction in overall height of the tree. If a node of M-way tree holds K number of keys then it will have K+1 children.

Example :

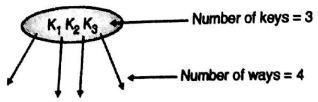
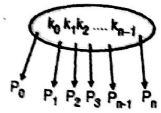


Fig. 2 : An M-way tree with 3 keys and 4 children

M is a M-way search tree with the following properties :

1. The root can have 1 to M–1 keys.
2. All nodes (except the root) have between [(M–1)/2] and M–1 keys.
3. All leaves are at the same depth.
4. If a node has t number of children then it must have (t – 1) number of keys.
5. Keys of a node are stored in ascending order.



6. K_0, K_1, K_2 ... K_{n-1} are the keys stored in the node. Subtrees are pointed by P_0, P_1 ... P_n.

then $K_0 \geq$ all keys of the subtree P_0

$K_1 \geq$ all keys of the subtree P_1

:

:

$K_{n-1} \geq$ all keys of the subtree P_{n-1}

$K_{n-1} <$ all keys of the subtree P_n

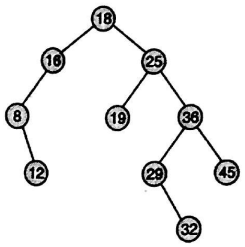
Q. 4 (b) Consider the following list of numbers : **(10 Marks)**

18, 25, 16, 36, 08, 29, 45, 12, 32, 19

Create a binary search tree using these numbers and display them in a non-decreasing order. Write a 'C' program for the same.

Ans. :

Binary search tree of the given numbers



List in non-decreasing order

8 12 16 18 19 25 29 32 36 45

Program :

```
#include <stdio.h>
#include <stadlib.h>
typedef struct BSTnode
{
   int data;
   struct BSTnode * left, * right;
} BSTnode;
BSTnode *insert (BSTnode *, int);
BSTnode * create();
void inorder (BSTnode * T)      ;
void main ()
{
   BSTnode  * root;
   root = create ();
   inorder (root);
}
void inorder (BSTnode * T)
{
if(T!=NULL)
{
   inorder(T->left);
   printf("%5d",T->data);
   inorder(T->right);
}
}
BSTnode *insert(BSTnode *T,int x)
```

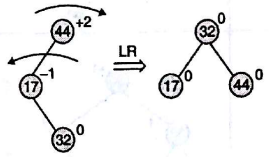
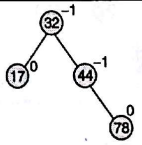
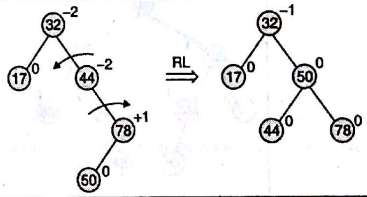
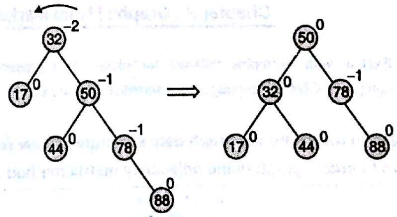
```
{
BSTnode *p,*q,*r;
// acquire memory for the new node
r=(BSTnode*)malloc(sizeof(BSTnode));
r->data=x;
r->left=NULL;
r->right=NULL;
if(T==NULL)
    return(r);
// find the leaf node for insertion
p=T;
while(p!=NULL)
{
    q=p;
    if(x>p->data)
        p=p->right;
    else
        p=p->left;
}
if(x>q->data)
    q->right=r;  // x as right child of q
else
    q->left=r;   //x as left child of q
return(T);
}
BSTnode *create()
{
int n,x,i;
BSTnode *root;
root=NULL;
printf("\nEnter no. of nodes :");
scanf("%d",&n);
printf("\nEnter tree values :");
for(i=0;i<n;i++)
{
    scanf("%d",&x);
    root=insert(root,x);
}
return(root);
}
```

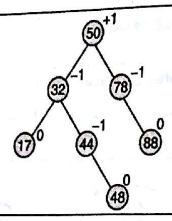
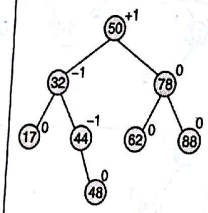
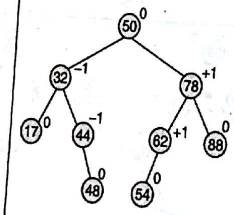

Q. 6 (a) Insert the following elements in AVL tree : **(10 Marks)**

44, 17, 32, 78, 50, 88, 48, 62, 54.

Explain the different rotations that will be used.

Ans. :

Sr. No.	Data to be inserted	Tree after insertion	Tree after rotation
1.	44		
2.	17, 32		
3.	78		
4.	50		
5.	88		

Sr. No.	Data to be inserted	Tree after insertion	Tree after rotation
6.	48		
7.	62		
8.	54		

Chapter 9 : Graphs [Total Marks : 05]

Q. 2(c) Explain with examples different techniques to represent the graph data structure on a computer. Give 'C' language representations for the same. **(5 Marks)**

Ans. :

Techniques to represent the graph data structure : Please refer Q.1(c) of May 2014.

C function to create graph using adjacency matrix method :

```
int n;                  //no. of vertices
int adj[10][10];        //adjacency matrix
```



```
void create( )
{
   int i, u, v, n1, j, w;
    printf("\n Enter no. of vertices : ");
   scanf("%d", & n);
   printf("\n Enter no. of edges : ");
   scanf("%d", & n1);
   //initialize the adjacency matrix
   for (i=0; i<n; i++)
            for (j=0; j<n; j++)
                adj[i][j]=0;
   //read edges
   printf("\n Enter edge as (u, v, w): ");
   for (i=0; i<n1; i++)
       {
           scanf("%d%d%d", &u, &v, &w);
           adj[u][v]=adj[v][u]=w;
       }
}
```

C function to create graph using adjacency list method :

```
# define MAX 30                    /* graph has maximum of 30 nodes */
typedef struct node
{
struct node * next;
int vertex;
}node;
node * head[MAX];
```

Q. 5(b) Write a function for DFS traversal of graph. Explain its working with an example. **(5 Marks)**

Ans. :

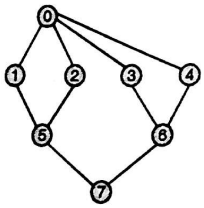
Function of DFS traversal on a graph represented using an adjacency matrix :

```
#include<conio.h>
#include<stdio.h>
void DFS(int);
int G[10][10],visited[10],n;
// n->no of vertices
// graph is stored in array G[10][10]
void main()
{
int i,j;
```

```
printf("\nEnter no of vertices: ");
scanf("%d",&n);
// read the adjecency matrix
printf("\nEnter adjecency matrix of the ghaph :");
for(i=0;i<n;i++)
   for(j=0;j<n;j++)
      scanf("%d",&G[i][j]);

// visited is initialise to zero
for(i=0;i<n;i++)
   visited[i]=0;
DFS(0);
}

void DFS(int i)
{
int j;
printf("\n%d",i);
visited[i]=1;
for(j=0;j<n;j++)
   if(!visited[j] && G[i][j]==1)
      DFS(j);
}
```

Output

```
Enter no of vertices: 8
Enter adjecency matrix of the ghaph :0 1 1 1 1 0 0 0
1 0 0 0 0 1 0 0
1 0 0 0 0 1 0 0
1 0 0 0 0 0 1 0
1 0 0 0 0 0 1 0
0 1 1 0 0 0 0 1
0 0 0 1 1 0 0 1
0 0 0 0 0 1 1 0
0
1
5
2
7
6
3
4
```



Graph used for input

Example :

DFS (G, 1) is given by

(a) Visit (1)

(b) DFS (G, 2), DFS (G, 3), DFS (G, 4), DFS (G, 5) } all nodes adjacent to 1

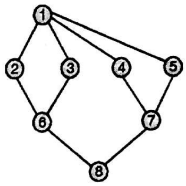
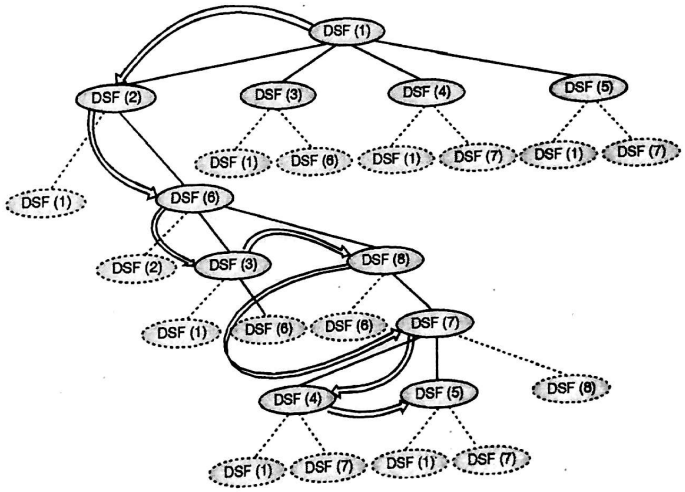


Fig. 3 : Graph G

DFS traversal on graph of Fig. 3.



DSF(i) Node i can be used for recursive traversal using DFS().

DFS(i) Node i is already visited

Traversal start from vertex 1. Vertices 2, 3, 4 and 5 are adjacent to vertex 1. Vertex 1 is marked as visited.

	1	2	3	4	5	6	7	8
Visited →	1	0	0	0	0	0	0	0

Out of the adjacent vertices 2, 3, 4 and 5, vertex number 2 is selected for further traversal. Vertices 1 and 6 are adjacent to vertex 2. Vertex 2 is marked as visited.

	1	2	3	4	5	6	7	8
Visited →	1	1	0	0	0	0	0	0

Out of the adjacent vertices 1 and 6, vertex 1 has already been visited. Vertex number 6 is selected for further traversal. Vertices 2, 3 and 8 are adjacent to vertex 6. Vertex 6 is marked as visited.

	1	2	3	4	5	6	7	8
Visited →	1	1	0	0	0	1	0	0

Out of the adjacent vertices 2, 3 and 8, vertex 2 is already visited. Vertex number 3 is used for further expansion. Vertices 1 and 6 are adjacent to vertex 3. Vertex 3 is marked as visited.

	1	2	3	4	5	6	7	8
Visited →	1	1	1	0	0	1	0	0

Vertices 1 and 6 are already visited, therefore it goes back to vertex 6. (Vertex 6 is predecessor of vertex 3 in DFS() sequence). Out of the adjacent vertices 2, 3 and 8, 2 and 3 are visited. It selects vertex 8 for further expansion. Vertices 6 an 7 are adjacent to vertex 8. Vertex 8 is marked as visited.

	1	2	3	4	5	6	7	8
Visited →	1	1	1	0	0	1	0	1

Out of the adjacent vertices 6 and 7, vertex 6 is already visited. Vertex number 7 is used for further expansion. Vertices 4, 5 and 8 are adjacent to vertex number 7. Vertex 7 is marked as visited.

	1	2	3	4	5	6	7	8
Visited →	1	1	1	0	0	1	1	1

Out of the adjacent vertices 4, 5 and 8, vertex 4 is selected for further expansion. Vertices 1 and 7 are adjacent to vertex 4. Vertex 4 is marked as visited.

	1	2	3	4	5	6	7	8
Visited →	1	1	1	1	0	1	1	1

Adjacent vertices 1 and 7 are already visited. It goes back to vertex 7 and selects the next unvisited node 5 for further expansion. Vertex 5 is marked as visited.

	1	2	3	4	5	6	7	8
Visited →	1	1	1	1	1	1	1	1

DFS, traversal sequence → 1, 2, 6, 3, 8, 7, 4, 5

Chapter 10 : Searching and Sorting [Total Marks : 18]

Q. 1(d) Compare and contrast quick sort and radix sort. **(5 Marks)**

Ans. :

Sr. No	Quick sort	Radix sort
Efficiency	Best case : O(n log n) Worst case : $O(n^2)$	O(n)
Passes	Best case : log n Worst case : n – 1	No. of digits in the largest number
Sort stability	Unstable	Stable
Best case	O(n log n)	O(n)
Worst case	$O(n^2)$	O(n)
Average case	O(n log n)	O(n)

Q. 6(b) What are the advantages of using indexed sequential search over sequential search? **(3 Marks)**

Ans. :

Advantages of using indexed sequential search :

The index file of a indexed sequential file needs substantially fewer blocks than does the data file as :

(a) There are fewer index entries than there are records in the data files.

(b) Each index entry is smaller in size than a data record. More index entries can fit in one block.

A binary search on the index file requires fewer block accesses than a binary search on the data file. With index file, the problem is compounded, if we attempt to insert a record in its correct position in data file, we have to not only move records to make space for the new record but also change index entries.

Q. 3(a) Consider the following list of numbers : **(10 Marks)**

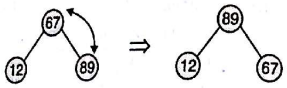
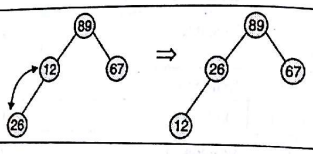
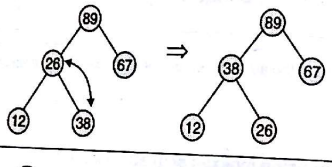
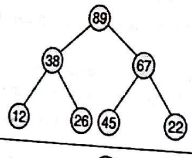
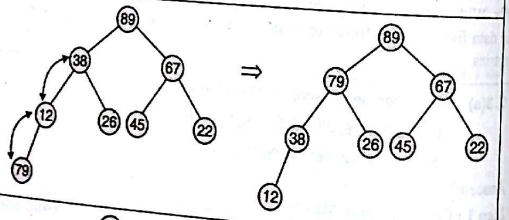
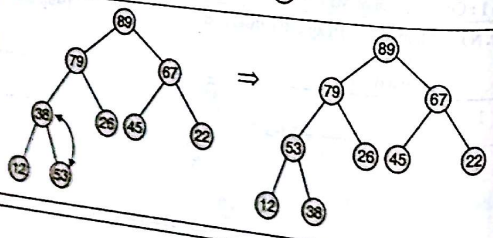
67, 12, 89, 26, 38, 45, 22, 79, 53, 9, 61.

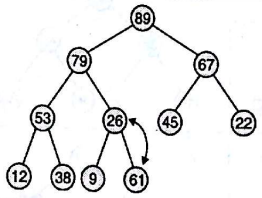
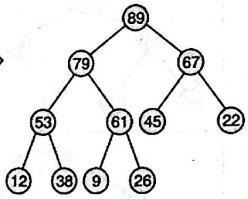
Sort these numbers using Heap Sort.

Ans. :

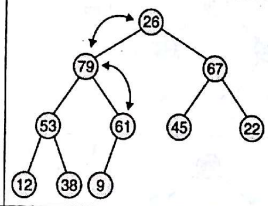
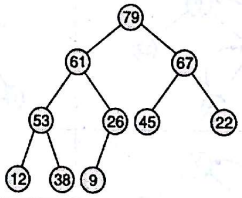
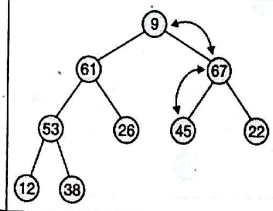
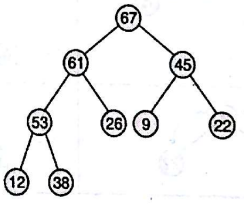
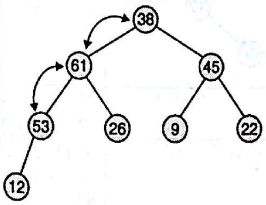
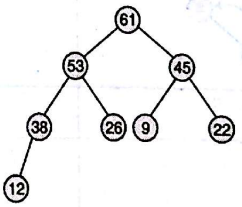
Step 1 : Construction of Max heap.

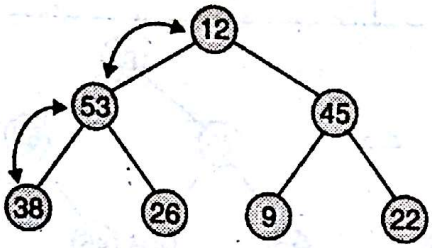
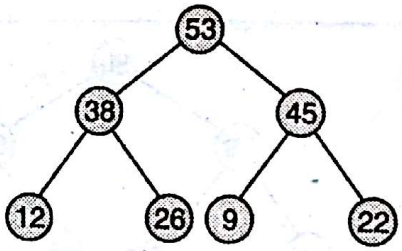
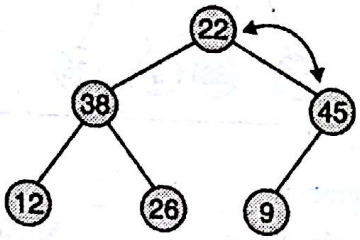
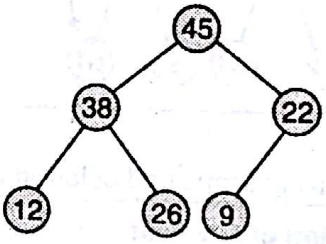
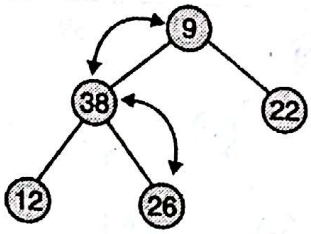
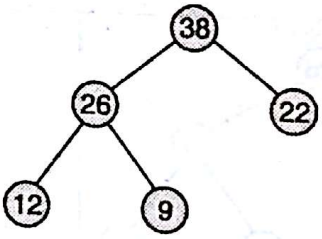
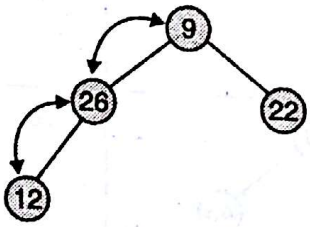
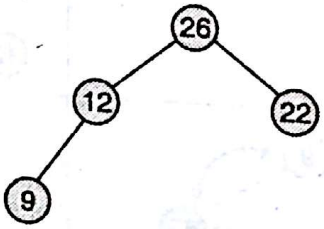
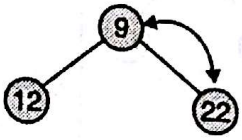
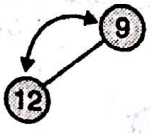
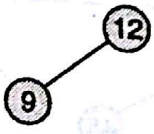
Sr. No.	Element to be inserted	Heap after insertion	Heap after adjustment
1.	67	(67)	
2.	12	(67) (12)	

Sr. No.	Element to be inserted	Heap after insertion	Heap after adjustment
3.	89		
4.	26		
5.	38		
6.	45, 22		
7.	79		
8.	53		

Sr. No.	Element to be inserted	Heap after insertion	Heap after adjustment
9.			

Step 2 : Sorting of data through repeated deletion of the root element

Sr. No.	Heap after deletion of the root	Heap after adjustment	Sorted list
1.			89
2.			79, 89
3.			67, 79, 89

Sr. No.	Heap after deletion of the root	Heap after adjustment	Sorted list
4.			61, 67, 79, 89
5.			53, 61, 67, 79, 89
6.			45, 53, 61, 67, 79, 89
7.			38, 45, 53, 61, 67, 79, 89
8.			26, 38, 45, 53, 61, 67, 79,89
9.			22, 26, 38, 45, 53, 61, 67, 79, 89
10.	9	9	12, 22, 26, 38, 45, 53, 61, 67, 79, 89
11.	–	–	9, 12, 22, 26, 38, 45, 53, 61, 67, 79, 89

May 2015

Chapter 1 : Types of Data Structure and Recursion [Total Marks : 15]

Q. 1(d) What is data structure ? List out the areas in which data structures are applied extensively ? **(5 Marks)**

Ans. : Data structure :

Data structure is logical or mathematical model of a particular organization of data.

The mathematical model explains how data is stored in memory and which operations are possible and which are economical etc.

Following are the areas in which data structures are applied extensively :

1. Compiler Design,
2. Operating System,
3. Database Management System,
4. Statistical analysis package,
5. Numerical Analysis,
6. Graphics,
7. Artificial Intelligence,
8. Simulation

Q. 3(b) Explain linear and non-linear data structures with examples. **(5 Marks)**

Ans. : Please refer Q.1(a) of Dec. 2013.

Q. 3(c) Explain the term recursion with an examples. **(5 Marks)**

Ans. : Recursion :

Recursion, in mathematics and computer science, is a method of defining functions in which the function being defined is applied within its own definition. The term is also used more generally to describe a process of repeating objects in a self-similar way.

Factorial : A classic example of a recursive procedure is the function used to calculate the factorial of an integer.

Function Definition : A recurrence relation is an equation that relates later terms in the sequence to earlier terms.

Recurrence relation for factorial: bn = n * bn-1 b0 = 1

Computing the recurrence relation for n = 4:

b4 = 4 * b3 = 4 * 3 * b2 = 4 * 3 * 2 * b1 = 4 * 3 * 2 * 1 * b0 = 4 * 3 * 2 * 1 * 1 = 4 * 3 * 2 * 1 = 4 * 3 * 2 = 4 * 6 = 4 * 6 = 24.

```
#include <stdio.h>
long int multiplyNumbers(int n);
int main()
{
   int n;
   printf("Enter a positive integer: ");
   scanf("%d", &n);
   printf("Factorial of %d = %ld", n, multiplyNumbers(n));
   return 0;
}
long int multiplyNumbers(int n)
```

```
{
   if (n >= 1)
      return n*multiplyNumbers(n-1);
   else
      return 1;
}
```

Output :

```
Enter a positive integer: 4
Factorial of 4 = 24
```

Chapter 4 : File Handling in C [Total Marks : 5]

Q. 6(c) What is a file ? Explain various file handling operations in C. **(5 Marks)**

Ans. :

File :

A **file** represents a sequence of bytes on the disk where a group of related data is stored. File is created for permanent storage of data. It is a ready made structure.

In C language, we use a structure **pointer of file type** to declare a file.

FILE *fp;

File handling operations in C :

C provides a number of functions that helps to perform basic file operations. Following are the functions,

Operation	description
fopen()	create a new file or open a existing file
fclose()	closes a file
getc()	reads a character from a file
putc()	writes a character to a file
fscanf()	reads a set of data from a file
fprintf()	writes a set of data to a file
getw()	reads a integer from a file
putw()	writes a integer to a file
fseek()	set the position to desire point
ftell()	gives current position in the file
rewind()	set the position to the begining point

Chapter 5 : Linked List [Total Marks : 15]

Q. 1(a) State differences between singly linked list and doubly linked list data structures along with their applications. **(5 Marks)**

Ans. : Difference between singly linked list and doubly linked list :

Sr. No.	Singly Linked List	Doubly Linked List
1	Singly linked list allows you to go one way direction	Doubly linked list has two way directions next and previous
2	Singly linked list uses less memory per node (one pointer)	Doubly linked list uses More memory per node than Singly Linked list (two pointers)
3	Complexity of Insertion and Deletion at known position is O (n).	Complexity of Insertion and Deletion at known position is O (1).
4	If we need to save memory in need to update node values frequently and searching is not required, we can use Singly Linked list.	If we need faster performance in searching and memory is not a limitation we use Doubly Linked List
	In single list Each node contains at least two parts: a) info b) link	In doubly linked list Each node contains at least three parts: a) info b) link to next node c) link to previous node

Linked list have many advantages. Some of them are as follows,

1. Linked list are dynamic data structure. i.e. they can grow or shrink during the program execution.
2. Utilization of memory : In linked list memory is not preallocated. Memory is allocated as per requirement and it is deallocated when it is no longer needed.
3. The insertion and deletion operations are easier and effective. Linked list provides flexibility in inserting a data item at a specified position and deletion of data item from the given position.
4. Many complex applications can be easily carried out with the help of linked list.

Q. 5(a) Write a C program to implamanrt doubly linked list. **(10 Marks)**

Provide following operations

(i) Insert at beginning (ii) Insert at location

(iii) Remove from beginning (iv) Remove from location

Ans. : Implementation of double linked list :

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

struct node
{
struct node *prev;
int info;
struct node *next;
}*start;

main()
{
```

```
int choice,n,m,po,i;
start=NULL;
clrscr();
printf("\n\tDouble Linked List");
printf("\n_____\n\n");
while(1)
{
   printf("\t1.Create List\n");
   printf("\t2.Add at begining\n");
   printf("\t3.Add after\n");
   printf("\t4.Delete\n");
   printf("\t5.Display\n");
   printf("\t6.Count\n");
   printf("\t7.Reverse\n");
   printf("\t8.exit\n");
   printf("\tEnter your choice : ");
   scanf("%d",&choice);
   switch(choice)
   {
    case 1:
        printf("\nHow many nodes you want : ");
        scanf("%d",&n);
        for(i=0;i<n;i++)
        {
            printf("\nEnter the element : ");
            scanf("%d",&m);
            create_list(m);
        }
        break;
    case 2:
        printf("\nEnter the element : ");
        scanf("%d",&m);
        addatbeg(m);
        break;
    case 3:
        printf("\nEnter the element : ");
        scanf("%d",&m);
        printf("Enter the position after which this element is inserted : ");
        scanf("%d",&po);
        addafter(m,po);
        break;
    case 4:
        printf("\nEnter the element for deletion : ");
        scanf("%d",&m);
```



```
        del(m);
        break;
    case 5:
        display();
        break;
    case 6:
        count();
        break;
    case 7:
        rev();
        break;
    case 8:
        exit();
    default:
        printf("\nWrong choice\n");
}/*End of switch*/
  }/*End of while*/
}/*End of main()*/

create_list(int num)
{
struct node *q,*tmp;
tmp= malloc(sizeof(struct node));
tmp->info=num;
tmp->next=NULL;
if(start==NULL)
{
   tmp->prev=NULL;
   start->prev=tmp;
   start=tmp;
}
else
{
   q=start;
   while(q->next!=NULL)
        q=q->next;
   q->next=tmp;
   tmp->prev=q;
}
}/*End of create_list()*/

addatbeg(int num)
{
struct node *tmp;
```

```
tmp=malloc(sizeof(struct node));
tmp->prev=NULL;
tmp->info=num;
tmp->next=start;
start->prev=tmp;
start=tmp;
}/*End of addatbeg()*/

addafter(int num,int c)
{
struct node *tmp,*q;
int i;
q=start;
for(i=0;i<c-1;i++)
{
   q=q->next;
   if(q==NULL)
   {
        printf("\nThere are less than %d elements\n",c);
        return;
   }
}
tmp=malloc(sizeof(struct node) );
tmp->info=num;
q->next->prev=tmp;
tmp->next=q->next;
tmp->prev=q;
q->next=tmp;
}/*End of addafter() */

del(int num)
{
struct node *tmp,*q;
if(start->info==num)
{
   tmp=start;
   start=start->next;  /*first element deleted*/
   start->prev = NULL;
   free(tmp);
   return;
}
q=start;
while(q->next->next!=NULL)
{
```

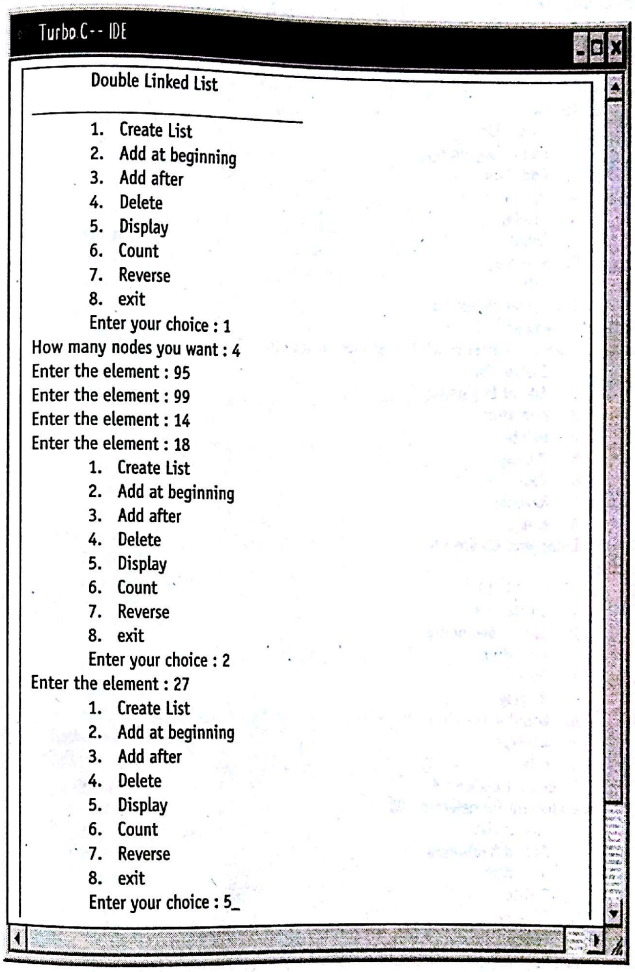


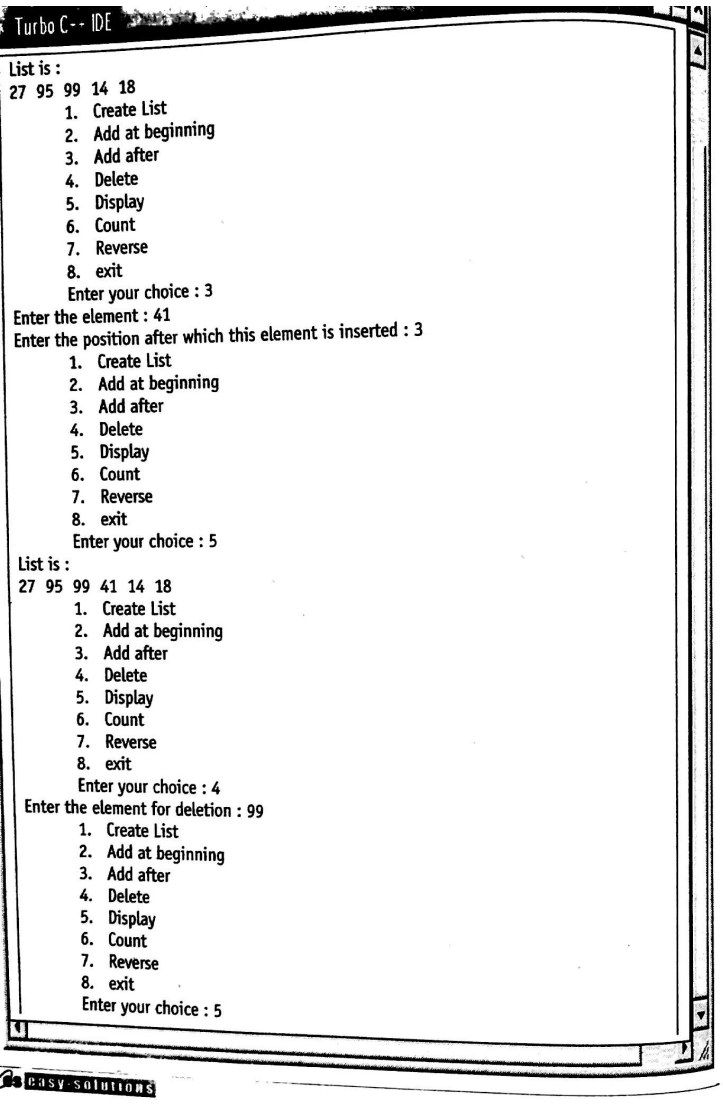
```
   if(q->next->info==num)    /*Element deleted in between*/
   {
        tmp=q->next;
        q->next=tmp->next;
        tmp->next->prev=q;
        free(tmp);
        return;
   }
   q=q->next;
}
if(q->next->info==num)   /*last element deleted*/
{  tmp=q->next;
   free(tmp);
   q->next=NULL;
   return;
}
printf("\nElement %d not found\n",num);
}/*End of del()*/

display()
{
struct node *q;
if(start==NULL)
{
   printf("\nList is empty\n");
   return;
}
q=start;
printf("\nList is :\n");
while(q!=NULL)
{
   printf("%d ", q->info);
   q=q->next;
}
printf("\n");
}/*End of display() */

count()
{  struct node *q=start;
```

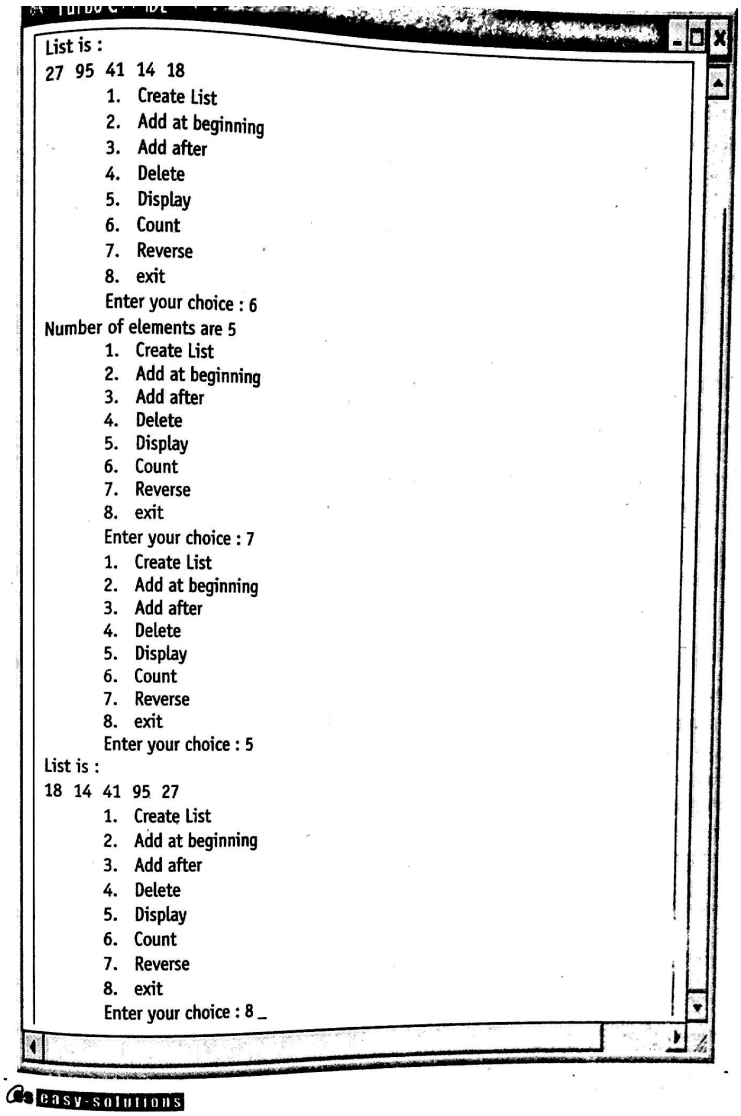
```
int cnt=0;
while(q!=NULL)
{
   q=q->next;
   cnt++;
}
printf("\nNumber of elements are %d\n",cnt);
```

```
}/*End of count()*/

rev()
{
struct node *p1,*p2;
p1=start;
p2=p1->next;
p1->next=NULL;
p1->prev=p2;
while(p2!=NULL)
{
   p2->prev=p2->next;
   p2->next=p1;
   p1=p2;
   p2=p2->prev; /*next of p2 changed to prev */
}
start=p1;
}/*End of rev()*/
```


Output :







Chapter 6 : Stack [Total Marks : 16]

Q. 2(c) Explain infix, postfix and prefix expressions with examples. **(6 Marks)**

Ans. :

Infix, Postfix and Prefix notations are three different but equivalent ways of writing expressions. It is easiest to demonstrate the differences by looking at examples of operators that take two operands.

Infix notation : X + Y

Operators are written in-between their operands. This is the usual way we write expressions. An expression such as `A * ( B + C ) / D` is usually taken to mean something like: "First add B and C together, then multiply the result by A, then divide by D to give the final answer."

Postfix notation (also known as "Reverse Polish notation") : X Y +

Operators are written after their operands. The infix expression given above is equivalent to `A B C + * D /`

The order of evaluation of operators is always left-to-right, and brackets cannot be used to change this order. Because the "+" is to the left of the "*" in the example above, the addition must be performed before the multiplication. Operators act on values immediately to the left of them. For example, the "+" above uses the "B" and "C". We can add (totally unnecessary) brackets to make this explicit.

Prefix notation (also known as "Polish notation") : + X Y

Operators are written before their operands. The expressions given above are equivalent to `/ * A + B C D`

As for Postfix, operators are evaluated left-to-right and brackets are superfluous. Operators act on the two nearest values on the right. I have again added (totally unnecessary) brackets to make this clear:

`(/ (* A (+ B C) ) D)`

Q. 4(a) Write a C program to convert infix expression into postfix expression. **(10 Marks)**

Ans. :

C program to convert infix expression into postfix expression :

```
#define SIZE 50
#include<string.h>
#include <ctype.h>
char s[SIZE];
int top=-1;

push(char elem)
{
   s[++top]=elem;
}

char pop()
```



```
{
   return(s[top--]);
}

int pr(char elem)
{
   switch(elem)
   {
   case '#': return 0;
   case ')': return 1;
   case '+':
   case '-': return 2;
   case '*':
   case '/': return 3;
   }
}

main()
{
   char infx[50],prfx[50],ch,elem;
   int i=0,k=0;
   printf("\n\nRead the Infix Expression ? ");
   scanf("%s",infx);
   push('#');
   strrev(infx);
   while( (ch=infx[i++]) != '\0')
   {
      if( ch == ')') push(ch);
      else
         if(isalnum(ch)) prfx[k++]=ch;
         else
            if( ch == '(')
            {
               while( s[top] != ')')
                  prfx[k++]=pop();
               elem=pop();
            }
            else
            {
               while( pr(s[top]) >= pr(ch) )
                  prfx[k++]=pop();
```

```
            push(ch);
        }
    }
    while( s[top] != '#')
        prfx[k++]=pop();
    prfx[k]='\0';
    strrev(prfx);
    strrev(infx);
    printf("\n\nGiven Infix Expn: %s  Prefix Expn: %s\n",infx,prfx);
}
```

Chapter 7 : Queues [Total Marks : 10]

Q. 3(a) What is a circular queue ? Write a program in C to implement circular queue. **(10 Marks)**

Ans. : Circular Queue :

In a standard queue data structure re-buffering problem occurs for each dequeue operation. To solve this problem by joining the front and rear ends of a queue to make the queue as a circular queue Circular queue is a linear data structure. It follows FIFO principle.

In circular queue the last node is connected back to the first node to make a circle. Circular linked list fallow the First In First Out principle. Elements are added at the rear end and the elements are deleted at front end of the queue. Both the front and the rear pointers points to the beginning of the array. It is also called as "Ring buffer". Items can inserted and deleted from a queue in O(1) time.

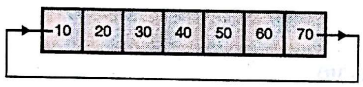


Fig. 1-Q. 3(a) : Circular queue

C Program to implement Circular queue using array.

```
#include<stdio.h>
#define max 3
int q[10],front=0,rear=-1;
void main()
{
   int ch;
   void insert();
   void delet();
   void display();
   clrscr();
   printf("\nCircular Queue operations\n");
   printf("1.insert\n2.delete\n3.display\n4.exit\n");
```



```
   while(1)
   {
      printf("Enter your choice:");
      scanf("%d",&ch);
      switch(ch)
      {
      case 1: insert();
         break;
      case 2: delet();
         break;
      case 3:display();
         break;
      case 4:exit();
      default:printf("Invalid option\n");
      }
   }
}

void insert()
{
   int x;
   if((front==0&&rear==max-1)||(front>0&&rear==front-1))
      printf("Queue is overflow\n");
   else
   {
      printf("Enter element to be insert:");
      scanf("%d",&x);
      if(rear==max-1&&front>0)
      {
         rear=0;
         q[rear]=x;
      }
      else
      {
         if((front==0&&rear==-1)||(rear!=front-1))
            q[++rear]=x;
      }
   }
}
void  delet()
{
```

```
    int a;
    if((front==0)&&(rear==-1))
    {
       printf("Queue is underflow\n");
       getch();
       exit();
    }
    if(front==rear)
    {
      a=q[front];
      rear=-1;
      front=0;
    }
    else
      if(front==max-1)
      {
        a=q[front];
        front=0;
      }
      else a=q[front++];
      printf("Deleted element is:%d\n",a);
}

void display()
{
  int i,j;
  if(front==0&&rear==-1)
  {
     printf("Queue is underflow\n");
     getch();
     exit();
  }
  if(front>rear)
  {
     for(i=0;i<=rear;i++)
        printf("\t%d",q[i]);
     for(j=front;j<=max-1;j++)
        printf("\t%d",q[j]);
     printf("\nrear is at %d\n",q[rear]);
     printf("\nfront is at %d\n",q[front]);
}
```



```
  else
  {
     for(i=front;i<=rear;i++)
     {
        printf("\t%d",q[i]);
     }
     printf("\nrear is at %d\n",q[rear]);
     printf("\nfront is at %d\n",q[front]);
  }
  printf("\n");
}
getch();
```

Chapter 8 : Trees [Total Marks : 21]

Q. 2(b) Define traversal of binary tree. Explain different types of traversals of Binary tree with examples. **(6 Marks)**

Ans. : Traversal of Binary Tree :

Traversing a tree means visiting each node of tree exactly once.

There are three ways of traversing a tree.

1) Preorder :

Here you perform following steps.

(i) Visit the root.

(ii) Traverse the left subtree in preorder.

(iii) Traverse the right subtree in preorder.

or in other words

```
Preorder (Tree T)
{
    if (T ! = 0)
    {
        Print (T → data) ;
        Preorder (T → Lchild) ;
        Preorder (T → Rchild) ;
    }
}
```

2) Inorder :

(i) Traverse the left subtree in inorder.

(ii) Visit the root.

(iii) Traverse the right subtree in inorder.

i.e. in other words.

```
Inorder (Tree T)
{
    if (T ! = 0)
```

```
    {
        Inorder (T → Lchild) ;
        Print (T → data) ;
        Inorder (T → Rchild) ;
    }
}
```

3) Postorder :

(i) Traverse the left subtree in Postorder.
(ii) Traverse the right subtree in Postorder.
(iii) Visit the root.

```
Postorder (Tree T)
{
    if (T ! = 0)
    {
        Postorder (T → Lchild) ;
        Postorder (T → Rchild) ;
        Print (T → data) ;
    }
}
```

Example : Find Preorder, Inorder, Postorder of the following tree.

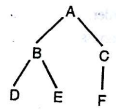


Fig 1-Q. 2(b) : Tree T

Solution :

(i) Preorder of Tree T in Fig. 1 is as follows,
Preorder → ABDECF.
(ii) Inorder of Tree T in Fig. 1 is as follows,
Inorder → DBEACF.
(iii) Postorder of Tree T in Fig. Ex.1 is as follows,
Postorder → DEBFCA.

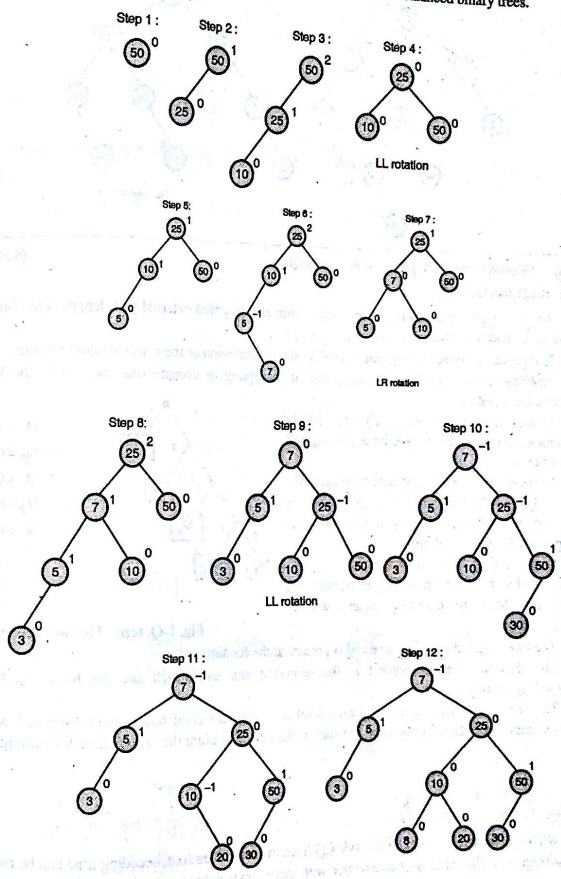
Q. 4(b) What is an AVL tree ? Construct AVL tree for the following data. Mention the type of rotation for each case.

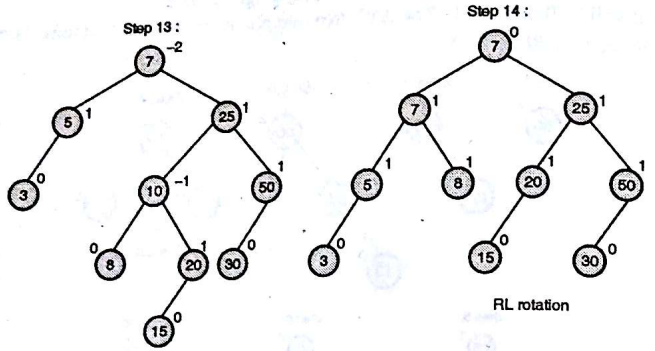
50, 25, 10, 5, 7, 3, 30, 20, 8, 15 **(10 Marks)**

Ans. : AVL Tree :

AVL tree is a binary tree in which depth (height) of two subtrees of any node never differ by more than 1.

Balance of node = Depth of left subtree – Depth of right subtree
In AVL it will be 0 or 1 of –1 . The AVL trees are also called as balanced binary trees.
50, 25, 10, 5, 7, 3, 30, 20, 8, 5





Q. 6(b) Explain Huffman Algorithm with an example. **(5 Marks)**

Ans. : Huffman Codes :

Another application of binary trees with minimum weighted external path length is to obtain an optimal set of codes for messages $M_1, M_2, ..., M_{n+1}$.

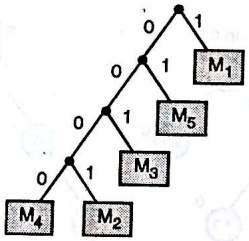
Each code is a binary string that is used for the transmission of the corresponding message.

Huffman code is used in encoding that is encrypting or compressing the text in the WSSS communication system.

It uses patterns of zeros (0^s) and (1^s). In communication system these are used at sending and receiving end.

Suppose there are n standard messages M_1, M_2, ..., M_n then the frequency of each message is considered, that is message with highest frequency is given priority for the encoding.

Suppose there are 5 messages M_1, M_2, M_3, M_4, M_5. M_1 having the highest frequency then M_5, M_3, M_2 and M_4. The encoding pattern will as follows,



$M_1 = 1$
$M_5 = 01$
$M_3 = 001$
$M_2 = 0001$
$M_4 = 0000$

Fig. 1-Q. 6(b) : Huffman codes

The tree is called encoding tree and is present at the sending end.

The decoding tree is present at the receiving end and which decodes bit string to the corresponding message.

The cost of decoding is directly proportional to the number of bits in the transmitted code is equal to distance of external node from the root in the tree. Therefore the decode time for message will be,

$$M_i = \sum_{i=1}^{n} d_i q_i$$

Where, d_i = distance of external node (q_i) from the root. Therefore, decoding time can be reduced if the messages are chosen so as it decode tree with minimum weighted path length.

Example : The data file has 1 lakh characters, suppose the frequencies of alphabets is as follows :

Message	Frequencies
A	45000
B	13000
C	12000
D	16000
E	9000
F	5000

Design the Huffman code using optimal merge binary tree.

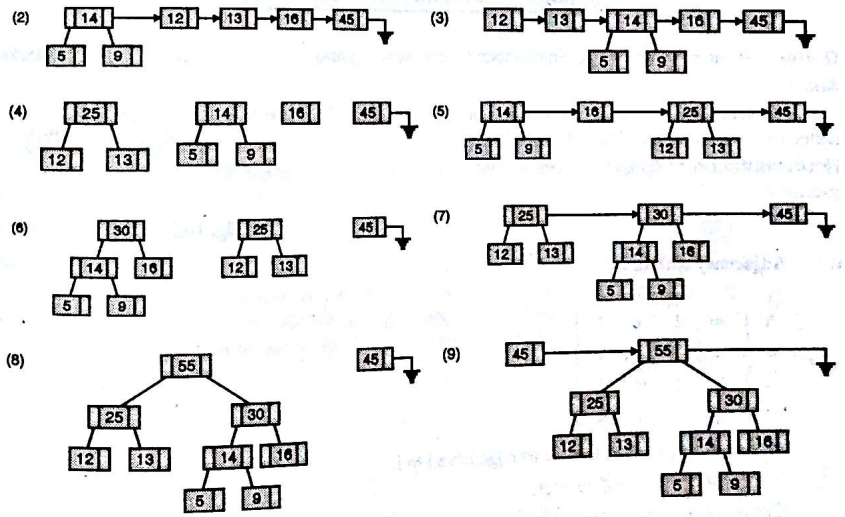
Solution :

Arrange the messages in ascending order according to frequencies.

F	5000
E	9000
C	12000
B	13000
D	16000
A	45000

(1) 5 9 12 13 16 45

(We take values without zeros just for simplicity)



(10)

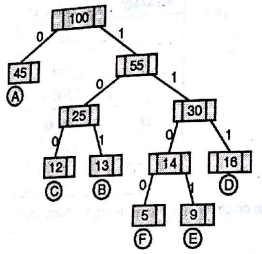


Fig. 2-Q. 6(b) : Binary tree merge pattern to obtain Huffman codes

The Huffman code for above tree is as follows,

A = 0
B = 1 0 1
C = 1 0 0
D = 1 1 1
E = 1 1 0 1
F = 1 1 0 0

Chapter 9 : Graphs [Total Marks : 5]

Q. 1(b) Whar is a graph ? Explain methods to represent a grapgh. **(5 Marks)**

Ans. :

Definition : A graph is a collection of two sets V and E where V is collection of vertices and E is collection of edges.

Representation of Graph : There are two ways for representing graph in memory.

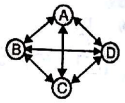


Fig. 1-Q. 1(b) : Graph G

i) **Adjacency matrix :**

	A	B	C	D
A	∞	1	1	1
B	1	∞	1	1
C	1	1	∞	1
D	1	1	1	∞

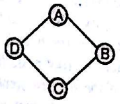
P [i] [j] = 1 if there is an edge from i to j.

P [i] [j] = ∞ if no edge.

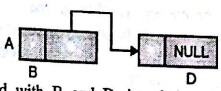
Create the adjacency matrix of given graph by using above method. Write 1 if there exist an edge between two nodes otherwise write ∞.

ii) **Linked-list representation : [Adjacency list]**

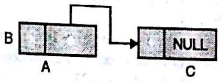
Here you have to maintain a linked list separate for each node which indicates adjacent nodes for that node.



Link list for node A.

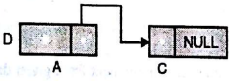


Here node A is connected with B and D. i.e. there exists edges AB and AD. Link list representation is shown above. Adjacency list for Node B.



Adjacency list for Node C

Adjacency list for Node D



Chapter 10 : Searching and Sorting [Total Marks : 33]

Q. 1(c) What is binary search tree ? Explain with an example. **(5 Marks)**

Ans. :

Binary search tree :

Binary search tree is a special type of binary tree where nodes are arranged as per their values. So that all the nodes in left subtree have values less than the root and all the nodes in right subtree have values more than the root.

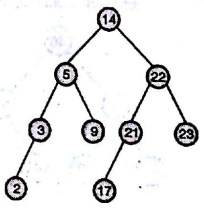


Fig. 1-Q. 1(c) : Binary search tree

Operations on Binary Search Tree :

Insertion :

Insertion means adding a new element in the data structure. For inserting a new node in the binary search tree first data to be inserted is compared with root node of the tree. If the inserted is element is equal to or greater than the root node then it is inserted at right side or right subtree of the root element. It the inserted element is less than the root element then it is inserted at the left side or left subtree of the root node. This whole procedure is repeated until the element is added at the proper place. For example, consider the following binary tree. We want to add element 21 in it.

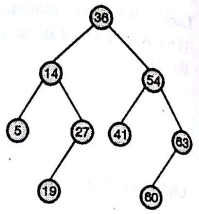


Fig. 2-Q. 1(c) : Binary search tree T

In above tree T we want to add element 21.

Step 1 : Compare element 21 with the root node 36. Since 21 < 36, it must add to the left side or in left subtree of root node 36. Left child of root node 36 is 14.

Step 2 : Compare element 21 with the node 14, since 21 > 14 it must add to right side or in right subtree of node 14. i.e. proceed in right child of 14. Right child of node 14 is 27.

Step 3 : Compare element 21 with the node 27. Since 21 < 27. Proceed in the left direction or left subtree of node 27. Left child of 27 is 19.

Step 4 : Compare element 21 with the node 19. Since 21 > 19, add to the right subtree or right child of node 19. i.e. 21 got its proper place. It shown in Fig. 3-Q. 1(c).

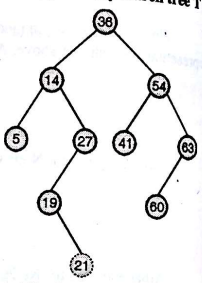


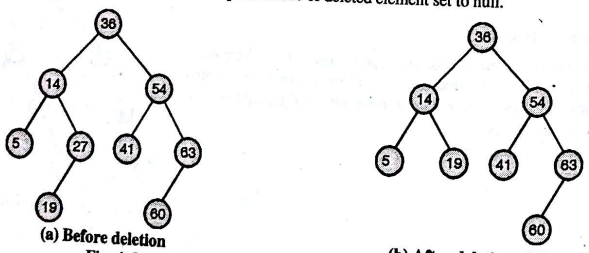
Fig. 3-Q. 1(c) : Binary search tree T_1

Deletion :

Deletion means deleting or removing an element from given data structure. There are four cases regarding the deletion in binary search tree.

Case I : The specified element which is to be deleted is not present in the tree. Then we can give message that the given or specified data is not available in the tree.

Case II : The specified element which is to be deleted no children. Then the element is deleted or removed from tree and link of parent node of deleted element set to null.



(a) Before deletion **(b) After deletion of 27**

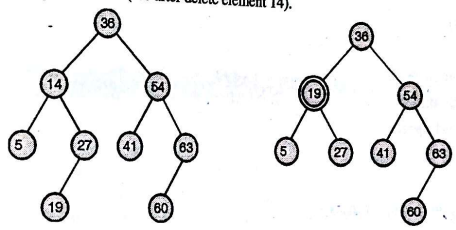
Fig. 4-Q. 1(c) : Deletion of element having exactly one child.

Case III : The specified element which is to be deleted having exactly one child either left or right. Then the element is deleted and removed from the tree and its place taken by or deleted element position is adopted element by its child element. Consider Fig. 5-Q. 1(c).

The Fig. 4-Q. 1(c) shows the deletion of node 27. The node 27 having exactly one child 19 (see before deletion). After deleting or removing the node 27 the child of 27 i.e. 19 is connected as a right child of node 14. [Place of 27 is taken by its children node 19] see after deletion.

Case IV : The specified element which is to be deleted having two children left and right both. In this case the element which is to deleted is removed from tree and its position is taken by in order successor of deleted element. The inorder successor of element having one or no child. Consider Fig. 5-Q. 1(c).

The Fig. 5-Q. 1(c) shows deletion of element 14. 14 is having two children 5 (left child) and 27 (right child).The inorder successor of element 14 is 19. So after removing node 14 the vacant place is filled by 19, i.e. now node 19 having two children 5 (left child) and 27 (right child). This satisfies the conditions of binary search tree (see after delete element 14).



(a) Before deletion **(b) After delete element 14**

Fig. 5-Q. 1(c) : Deletion of element having tow children

Q. 2(a) Write a program in C to implement the quick sort algorithm. **(8 Marks)**

Ans. :

Program of sorting elements by using Quick sort algorithm.

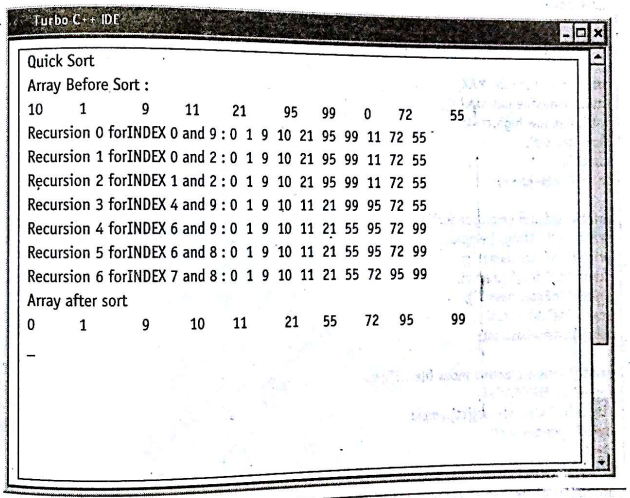
```
#include<conio.h>
#include<stdio.h>
int split(int*,int,int);
void main()
{
int arr[10]={10,1,9,11,21,95,99,0,72,55},i;
void quicksort(int*,int,int);
clrscr();
printf("\n Quick Sort\n");
printf("\n\n Array Before Sort:\n\n");
for(i=0;i<=9;i++)
```

```
    printf("%d\t",arr[i]);
 quicksort(arr,0,9);
 printf("\n \n Array after sort \n\n");
 for(i=0;i<=9;i++)
    printf("%d\t",arr[i]);
 getch();
 }
 void quicksort(int z[], int lower, int upper)
 {
 int i;
 if(upper>lower)
 {
    i=split(z,lower,upper);
    quicksort(z,lower,i-1);
    quicksort(z,i+1,upper);
 }
 }
 int split(int z[], int lower, int upper)
 {
 static int rec;
 int i,a,b,t;
 a=lower+1;
 b=upper;
 i=z[lower];
 while(b>=a)
 {
    while(z[a]<i)
       a++;
    while(z[b]>i)
       b--;
    if(a<b)
    {
       t=z[a];
       z[a]=z[b];
       z[b]=t;
```



```
    }
 }
 t=z[lower];
 z[lower]=z[b];
 z[b]=t;
 printf("\n\n Recursion %d forINDEX %d and %d : ",rec,lower,upper);
 rec++;
 for(i=0;i<=9;i++)
    printf("%d  ",z[i]);
 return b;
 }
```

Output :



Q. 5(b) What is indexed sequential search ? Write program in C to implement it. **Marks)**

Ans. : Indexed sequential search :

Indexed Sequential Access Method is a file management system developed at IBM that allows records to be accessed either sequentially (in the order they were entered) or randomly (with an index). Each index defines a different ordering of the records.

An employee database may have several indexes, based on the information being sought. For example, a name index may order employees alphabetically by last name, while a department index may order employees by their department. A key is specified in each index. For an alphabetical index of employee names, the last name field would be the key.

C program to implement indexed sequential search :

```
#include <stdio.h>
#include <conio.h>
#define MAX 10
struct mainfile
{int empid;
char name[25];
float basic;
};
struct indexfile
{int index_id;
int kindex;
};
void main()
{
struct mainfile fobj[MAX];
struct indexfile index[MAX];
int i,num,low,high,ct=4;
float basicsal;
clrscr();
for(i=0;i<MAX;i++)
{
printf("\nEnter employee id?");
scanf("%d",&fobj[i].empid);
printf("\nEnter name?");
scanf("%s",fobj[i].name);
printf("\nEnter basic?");
scanf("%f",&basicsal);
fobj[i].basic=basicsal;
}
printf("\nNow creating index file...!");
for(i=0;i<(MAX/5);i++)
{index[i].index_id=fobj[ct].empid;
index[i].kindex = ct;
ct=ct+5;
}
printf("\n\nEnter the empid to search?");
scanf("%d",&num);
for(i=0;(i<MAX/5) && (index[i].index_id<=num);i++);
low=index[i-1].kindex;
high=index[i].kindex;
for(i=low;i<=high;i++)
```



```
{if(num==fobj[i].empid)
{
printf("\nThe record is: \n\t");
printf("\nEmpid: %d",fobj[i].empid);
printf("\nName: %s",fobj[i].name);
printf("\nBasic: %f",fobj[i].basic);
getch();
return;
}
}
printf("\nNumber not found...!");
return;
}
```

Q. 6(a) What is heap ? Consider the following list of numbers : 15, 19, 10, 7, 17, 16
Sort these numbers using heap sort. **(10 Marks)**

Ans. :

Heap :

Basically a heap is a complete binary tree. There are two types of heap tree or heaps.

1. **Maxheap or descending heap :**

It is a complete binary tree in which every node has a value greater than or equal to value of every child of that node.

2. **Minheap or ascending heap :**

It is a complete binary tree in which every node has a value less than or equal to value of its every child of that node.

Generally heap is stored as sequential representation in an array. Here left and right child of the node A [K] will be A [2K + 1] and A [2K + 2] respectively.

Given numbers 15, 19, 10, 7, 17, 16.

We first generate heap tree from above numbers.

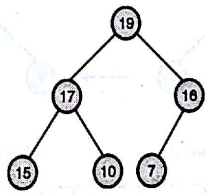
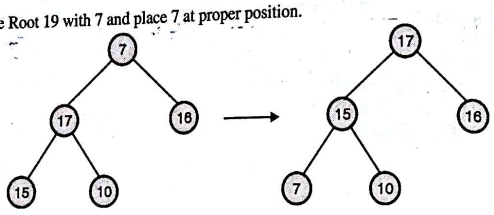


Fig. 1-Q. 6(a)

19	17	16	15	10	7

(1) Replace Root 19 with 7 and place 7 at proper position.

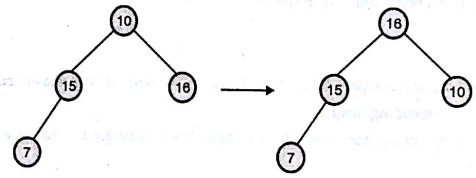


Before 7 placed at proper position **After placed at proper position**

Now position of array is

17	15	16	7	10	19

(2) Now we replace root 17 with 10 and place 10 at proper position.

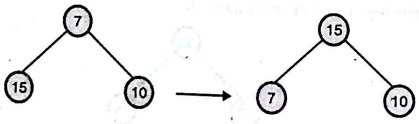


Before 10 placed at proper position **After placed at proper position**

Now position of array is

16	15	10	7	17	19

(3) Now replace root 16 with 7 and try to place 7 to the proper position in tree.

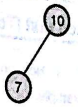


Before 7 placed to proper position **After 7 placed to proper position**

Now position of array is

15	7	10	16	17	19

(4) Now replace root 15 with 10 and try to place 10 the proper position in tree. But after replacing we seen that 10 and 7 are already at proper position.



Now position of array is

10	7	15	16	17	19

(5) Now replace 10 with 7

Now list of numbers in ascending order given below,

7	10	15	16	17	19

❑❑❑

Dec. 2015

Chapter 5 : Linked List [Total Marks : 10]

Q. 2(b) What do you mean by sparse matrix ? How one can implement sparse matrix using Linked list ? Support your answer with an example. **(10 Marks)**

Ans. :

Sparse matrix :

In computer programming, a matrix can be defined with a 2-dimensional array. Any array with 'm' columns and 'n' rows represents a mXn matrix. There may be a situation in which a matrix contains more number of ZERO values than NON-ZERO values. Such matrix is known as sparse matrix.

When a sparse matrix is represented with 2-dimensional array, we waste lot of space to represent that matrix. For example, consider a matrix of size 100 X 100 containing only 10 non-zero elements. In this matrix, only 10 spaces are filled with non-zero values and remaining spaces of matrix are filled with zero. That means, totally we allocate 100 X 100 X 2 = 20000 bytes of space to store this integer matrix. And to access these 10 non-zero elements we have to make scanning for 10000 times.

A sparse matrix can be represented by using triplet representation, as follows :

Triplet Representation :

In this representation, we consider only non-zero values along with their row and column index values. In this representation, the 0^{th} row stores total rows, total columns and total non-zero values in the matrix.

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

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 9 & 0 \\ 0 & 8 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 2 & 0 & 0 & 0 \end{bmatrix} \Rightarrow$$

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 × 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.

Sparse Matrix representation using singly Linked List :

```
#include <stdio.h>
#include <stdlib.h>
typedef struct list{
int rows, cols, value;
struct list *next;
}list;
list *create(){
list *temp = (list *)malloc(sizeof(list));
if(temp==NULL){
printf("nMemory Allocation Error !");
exit(1);
}
return temp;
}
list *makenode(int r, int c, int val){
list *temp = create();
temp->rows = r;
temp->cols = c;
temp->value = val;
temp->next = NULL;
return temp;
}
list *insert(list *head, int r, int c, int val){
list *ptr, *temp = head;
if(head == NULL){
head = makenode(r,c,val);
}
else{
while(temp->next != NULL)
temp = temp->next;
ptr = makenode(r,c,val);
temp->next = ptr;
}
return head;
}
void display(list *head){
list *temp;
if(head == NULL){
printf("nList is empty.");
exit(1);
}
temp = head;
while(temp != NULL){
printf("(%d,%d,%d->)->",temp->rows,temp->cols,temp->value);
temp = temp->next;
}
printf("bb ");
}
```

```
int main(){
int arr[3][4],i,j,m,n,ct=0;
list *head = NULL;
for(i=0; i<3; i++){
printf("nEnter the values for row %d?", i+1);
for(j=0;j<4;j++){
scanf("%d",&arr[i][j]);
if(arr[i][j] != 0)
ct++;
}
}
head = makenode(3,4,ct);
for(i=0;i<3;i++){
for(j=0;j<4;j++){
if(arr[i][j] != 0)
head = insert(head,i,j,arr[i][j]);
}
}
printf("nList representation of Sparse Matrix is: n");
display(head);
getch();
}
```

Chapter 6 : Stack [Total Marks : 20]

Q. 3(a) Explain STACK as ADT. Write a function in C to convert prefix expression to postfix expression. **(10 Marks)**

Ans. :

Stack as ADT :

The stack abstract data type is defined by the following structure and operations. A stack is structured, as described above, as an ordered collection of items where items are added to and removed from the end called the "top." Stacks are ordered LIFO. The stack operations are given below.

Stack() creates a new stack that is empty. It needs no parameters and returns an empty stack.

push(item) adds a new item to the top of the stack. It needs the item and returns nothing.

pop() removes the top item from the stack. It needs no parameters and returns the item. The stack is modified.

peek() returns the top item from the stack but does not remove it. It needs no parameters. The stack is not modified.

isEmpty() tests to see whether the stack is empty. It needs no parameters and returns a boolean value.

size() returns the number of items on the stack. It needs no parameters and returns an integer.

Program to convert Prefix expression to postfix expression :

```
#define SIZE 50
#include<string.h>
#include <ctype.h>
```



```
char s[SIZE];
int top=-1;

push(char elem)
{
   s[++top]=elem;
}

char pop()
{
   return(s[top--]);
}

int pr(char elem)
{
   switch(elem)
   {
   case '#': return 0;
   case ')': return 1;
   case '+':
   case '-': return 2;
   case '*':
   case '/': return 3;
   }
}

main()
{
   char infx[50],prfx[50],ch,elem;
   int i=0,k=0;
   printf("\n\nRead the Infix Expression ? ");
   scanf("%s",infx);
   push('#');
   strrev(infx);
   while( (ch=infx[i++]) != '\0')
   {
      if( ch == ')') push(ch);
      else
         if(isalnum(ch)) prfx[k++]=ch;
         else
            if( ch == '(')
```

```
        {
          while( s[top] != ')')
            prfx[k++]=pop();
          elem=pop();
        }
        else
        {
          while( pr(s[top]) >= pr(ch) )
            prfx[k++]=pop();
          push(ch);
        }
    }
    while( s[top] != '#')
      prfx[k++]=pop();
    prfx[k]='\0';
    strrev(prfx);
    strrev(infx);
    printf("\n\nGiven Infix Expn: %s  Prefix Expn: %s\n",infx,prfx);
}
```

Q. 3(b) Write a function in C to maintain 2 stacks in a single array. **(10 Marks)**

Ans. :

```
public:
  twoStacks(int  n)  // constructor
  {
    size = n;
    arr = new  int[n];
    top1 = -1;
    top2 = size;
  }

  // Method to push an element x to stack1
  void  push1(int  x)
  {
    // There is at least one empty space for new element
    if  (top1 < top2 - 1)
    {
      top1++;
      arr[top1] = x;
    }
    else
```



```
    {
      cout << "Stack Overflow";
      exit(1);
    }
  }

  // Method to push an element x to stack2
  void  push2(int  x)
  {
    // There is at least one empty space for new element
    if  (top1 < top2 - 1)
    {
      top2--;
      arr[top2] = x;
    }
    else
    {
      cout << "Stack Overflow";
      exit(1);
    }
  }

  // Method to pop an element from first stack
  int  pop1()
  {
    if  (top1 >= 0 )
    {
      int  x = arr[top1];
      top1--;
      return  x;
    }
    else
    {
      cout << "Stack UnderFlow";
      exit(1);
    }
  }

  // Method to pop an element from second stack
  int  pop2()
  {
```

```
    if (top2 < size)
    {
      int x = arr[top2];
      top2++;
      return x;
    }
    else
    {
      cout << "Stack UnderFlow";
      exit(1);
    }
  }
};
```

Chapter 7 : Queues [Total Marks : 10]

Q. 4(a) Explain Queue as ADT. Write a function in C to insert, delete and display elements in Circular Queue. **(10 Marks)**

Ans. :

Queue as ADT :

The queue abstract data type is defined by the following structure and operations. A queue is structured, as described above, as an ordered collection of items which are added at one end, called the "rear," and removed from the other end, called the "front." Queues maintain a FIFO ordering property. The queue operations are given below.

Queue() creates a new queue that is empty. It needs no parameters and returns an empty queue.

enqueue(item) adds a new item to the rear of the queue. It needs the item and returns nothing.

dequeue() removes the front item from the queue. It needs no parameters and returns the item. The queue is modified.

isEmpty() tests to see whether the queue is empty. It needs no parameters and returns a boolean value.

size() returns the number of items in the queue. It needs no parameters and returns an integer.

C Program to Implement Circular queue using array :

```
#include<stdio.h>
#define max 3
int q[10],front=0,rear=-1;
void main()
{
  int ch;
  void insert();
  void delet();
  void display();
  clrscr();
```



```
  printf("\nCircular Queue operations\n");
  printf("1.insert\n2.delete\n3.display\n4.exit\n");
  while(1)
  {
    printf("Enter your choice:");
    scanf("%d",&ch);
    switch(ch)
    {
    case 1: insert();
      break;
    case 2: delet();
      break;
    case 3:display();
      break;
    case 4:exit();
    default:printf("Invalid option\n");
    }
  }
}

void insert()
{
  int x;
  if((front==0&&rear==max-1)||(front>0&&rear==front-1))
    printf("Queue is overflow\n");
  else
  {
    printf("Enter element to be insert:");
    scanf("%d",&x);
    if(rear==max-1&&front>0)
    {
      rear=0;
      q[rear]=x;
    }
    else
    {
      if((front==0&&rear==-1)||(rear!=front-1))
        q[++rear]=x;
    }
  }
}
```

```
void delet()
{
  int a;
  if((front==0)&&(rear==-1))
  {
    printf("Queue is underflow\n");
    getch();
    exit();
  }
  if(front==rear)
  {
    a=q[front];
    rear=-1;
    front=0;
  }
  else
    if(front==max-1)
    {
      a=q[front];
      front=0;
    }
    else a=q[front++];
    printf("Deleted element is:%d\n",a);
}

void display()
{
  int i,j;
  if(front==0&&rear==-1)
  {
    printf("Queue is underflow\n");
    getch();
    exit();
  }
  if(front>rear)
  {
    for(i=0;i<=rear;i++)
      printf("\t%d",q[i]);
    for(j=front;j<=max-1;j++)
      printf("\t%d",q[j]);
    printf("\nrear is at %d\n",q[rear]);
```



```
    printf("\nfront is at %d\n",q[front]);
  }
  else
  {
    for(i=front;i<=rear;i++)
    {
      printf("\t%d",q[i]);
    }
    printf("\nrear is at %d\n",q[rear]);
    printf("\nfront is at %d\n",q[front]);
  }
  printf("\n");
}
getch();
```

Chapter 8 : Trees [Total Marks : 60]

Q. 1(a) Write a function to implement as HUFFMAN coding given a symbol and its frequency occurrence. **(10 Marks)**

Ans. :

Huffman Code :

Another application of binary trees with minimum weighted external path length is to obtain an optimal set of codes for messages $M_1, M_2, ..., M_{n+1}$.

Each code is a binary string that is used for the transmission of the corresponding message. Huffman code is used in encoding that is encrypting or compressing the text in the WSSS communication system.

It uses patterns of zeros (0^s) and (1^s). In communication system these are used at sending and receiving end. Suppose there are n standard messages $M_1, M_2, ..., M_n$ then the frequency of each message is considered, that is message with highest frequency is given priority for the encoding. Suppose there are 5 messages M_1, M_2, M_3, M_4, M_5. M_i having the highest frequency then M_5, M_3, M_2 and M_4. The encoding pattern will as follows,

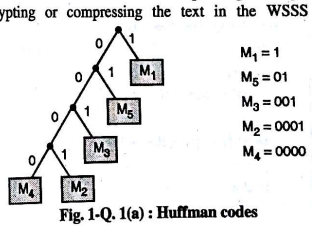


Fig. 1-Q. 1(a) : Huffman codes

The tree is called encoding tree and is present at the sending end. The decoding tree is present at the receiving end and which decodes bit string to the corresponding message. The cost of decoding is directly proportional to the number of bits in the transmitted code is equal to distance of external node from the root in the tree. Therefore the decode time for message will be,

$$M_i = \sum_{i=1}^{n} d_i q_i$$

Where, d_i = distance of external node (q_i) from the root. Therefore, decoding time can be reduced if the messages are chosen so as it decode tree with minimum weighted path length.

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

#define len(x) ((int)log10(x)+1)

/* Node of the huffman tree */
struct node{
   int value;
   char letter;
   struct node *left,*right;
};

typedef struct node Node;

/* 81 = 8.1%, 128 = 12.8% and so on. The 27th frequency is the space. Source is Wikipedia */
int englishLetterFrequencies [27] = {81, 15, 28, 43, 128, 23, 20, 61, 71, 2, 1, 40, 24, 69, 76, 20, 1, 61, 64,
91, 28, 10, 24, 1, 20, 1, 130};

/*finds and returns the small sub-tree in the forrest*/
int findSmaller (Node *array[], int differentFrom){
   int smaller;
   int i = 0;

   while (array[i]->value==-1)
      i++;
   smaller=i;
   if (i==differentFrom){
      i++;
      while (array[i]->value==-1)
         i++;
      smaller=i;
   }

   for (i=1;i<27;i++){
      if (array[i]->value==-1)
         continue;
      if (i==differentFrom)
         continue;
      if (array[i]->value<array[smaller]->value)
         smaller = i;
```



```
   }

   return smaller;

}

/*builds the huffman tree and returns its address by reference*/
void buildHuffmanTree(Node **tree){
   Node *temp;
   Node *array[27];
   int i, subTrees = 27;
   int smallOne,smallTwo;

   for (i=0;i<27;i++){
      array[i] = malloc(sizeof(Node));
      array[i]->value = englishLetterFrequencies[i];
      array[i]->letter = i;
      array[i]->left = NULL;
      array[i]->right = NULL;
   }

   while (subTrees>1){
      smallOne=findSmaller(array,-1);
      smallTwo=findSmaller(array,smallOne);
      temp = array[smallOne];
      array[smallOne] = malloc(sizeof(Node));
      array[smallOne]->value=temp->value+array[smallTwo]->value;
      array[smallOne]->letter=127;
      array[smallOne]->left=array[smallTwo];
      array[smallOne]->right=temp;
      array[smallTwo]->value=-1;
      subTrees--;
   }

   *tree = array[smallOne];

return;
}

/* builds the table with the bits for each letter. 1 stands for binary 0 and 2 for binary 1 (used to facilitate
arithmetic)*/
void fillTable(int codeTable[], Node *tree, int Code){
```

```
    if (tree->letter<27)
        codeTable[(int)tree->letter] = Code;
    else{
        fillTable(codeTable, tree->left, Code*10+1);
        fillTable(codeTable, tree->right, Code*10+2);
    }

    return;
}

/*function to compress the input*/
void compressFile(FILE *input, FILE *output, int codeTable[]){
    char bit, c, x = 0;
    int n,length,bitsLeft = 8;
    int originalBits = 0, compressedBits = 0;

    while ((c=fgetc(input))!=10){
        originalBits++;
        if (c==32){
            length = len(codeTable[26]);
            n = codeTable[26];
        }
        else{
            length=len(codeTable[c-97]);
            n = codeTable[c-97];
        }

        while (length>0){
            compressedBits++;
            bit = n % 10 - 1;
            n /= 10;
            x = x | bit;
            bitsLeft--;
            length--;
            if (bitsLeft==0){
                fputc(x,output);
                x = 0;
                bitsLeft = 8;
            }
            x = x << 1;
        }
```



```
    }

    if (bitsLeft!=8){
        x = x << (bitsLeft-1);
        fputc(x,output);
    }

    /*print details of compression on the screen*/
    fprintf(stderr,"Original bits = %dn",originalBits*8);
    fprintf(stderr,"Compressed bits = %dn",compressedBits);
    fprintf(stderr,"Saved %.2f%% of memoryn",((float)compressedBits/(originalBits*8))*100);

    return;
}

/*function to decompress the input*/
void decompressFile (FILE *input, FILE *output, Node *tree){
    Node *current = tree;
    char c,bit;
    char mask = 1 << 7;
    int i;

    while ((c=fgetc(input))!=EOF){

        for (i=0;i<8;i++){
            bit = c & mask;
            c = c << 1;
            if (bit==0){
                current = current->left;
                if (current->letter!=127){
                    if (current->letter==26)
                        fputc(32, output);
                    else
                        fputc(current->letter+97,output);
                    current = tree;
                }
            }

            else{
                current = current->right;
                if (current->letter!=127){
```

```
            if (current->letter==26)
               fputc(32, output);
            else
               fputc(current->letter+97,output);
            current = tree;
         }
      }
   }
  }

  return;
}

/*invert the codes in codeTable2 so they can be used with mod operator by compressFile function*/
void invertCodes(int codeTable[],int codeTable2[]){
  int i, n, copy;

  for (i=0;i<27;i++){
    n = codeTable[i];
    copy = 0;
    while (n>0){
      copy = copy * 10 + n %10;
      n /= 10;
    }
    codeTable2[i]=copy;
  }

return;
}

int main(){
  Node *tree;
  int codeTable[27], codeTable2[27];
  int compress;
  char filename[20];
  FILE *input, *output;

  buildHuffmanTree(&tree);

  fillTable(codeTable, tree, 0);
```



```
  invertCodes(codeTable,codeTable2);

  /*get input details from user*/
  printf("Type the name of the file to process:n");
  scanf("%s",filename);
  printf("Type 1 to compress and 2 to decompress:n");
  scanf("%d",&compress);

  input = fopen(filename, "r");
  output = fopen("output.txt","w");

  if (compress==1)
    compressFile(input,output,codeTable2);
  else
    decompressFile(input,output, tree);

  return 0;
}
```

Q. 1(b) Write a function to count the leaf nodes in Binary tree and Branch nodes in Binary tree. **(10 Marks)**

Ans. :

Function to count leaf node in Binary Tree and Binary Search Tree :

```
#include <stdio.h>
#include <stdlib.h>
struct btnode
{
  int value;
  struct btnode * l;
  struct btnode * r;
};
typedef struct btnode bt;
bt *root;
bt *new, *list;
int count = 0;
bt * create_node();
void display(bt *);
void construct_tree();
void count_leaf(bt *);
void main()
{
```

```
    construct_tree();
    display(root);
    count_leaf(root);
    printf("\n leaf nodes are : %d",count);
}
/* To create a empty node */
bt * create_node()
{
    new = (bt *)malloc(sizeof(bt));
    new->l = NULL;
    new->r = NULL;
}
/* To construct a tree */
void construct_tree()
{
    root = create_node();
    root->value = 50;
    root->l = create_node();
    root->l->value = 20;
    root->r = create_node();
    root->r->value = 30;
    root->l->l = create_node();
    root->l->l->value = 70;
    root->l->r = create_node();
    root->l->r->value = 80;
    root->l->r->r = create_node();
    root->l->r->r->value = 60;
    root->l->l->l = create_node();
    root->l->l->l->value = 10;
    root->l->l->r = create_node();
    root->l->l->r->value = 40;
}
/* To display the elements in a tree using inorder */
void display(bt * list)
{
    if (list == NULL)
    {
        return;
    }
    display(list->l);
    printf("->%d", list->value);
```



```
    display(list->r);
}
/* To count the number of leaf nodes */
void count_leaf(bt * list)
{
    if (list == NULL)
    {
        return;
    }
    if (list->l == NULL && list->r == NULL)
    {
        count++;
    }
    count_leaf(list->l);
    count_leaf(list->r);
}
```

Q. 2(a) Explain linked list as an ADT. Write a function for deletion of a node from Doubly linked list. **(10 Marks)**

Ans. :

Linked list as an ADT :

The idea behind this type of list is that any element we put into the list will have to be put into the correct position by comparing some common value held in each of the nodes. So far the Stack ADT and the Queue ADT did not impose any ordering on the elements we added. If we were looking for a basic linked list, we could use these as the basis for the structure and operations. This is a task you might attempt for yourselves.

Our sorted linked list ADT will consist of the following primitive operations:

```
Create()
Destroy()
Add(element)
Remove(element)
Traverse()
IsEmpty()
Search(element)
```

Function to delete node from Doubly linked list

```
del(int num)
{
struct node *tmp,*q;
if(start->info==num)
{
    tmp=start;
    start=start->next;  /*first element deleted*/
```

```
    start->prev = NULL;
    free(tmp);
    return;
}
q=start;
while(q->next->next!=NULL)
{
   if(q->next->info==num)    /*Element deleted in between*/
   {
        tmp=q->next;
        q->next=tmp->next;
        tmp->next->prev=q;
        free(tmp);
        return;
   }
   q=q->next;
}
if(q->next->info==num)   /*last element deleted*/
{  tmp=q->next;
   free(tmp);
   q->next=NULL;
   return;
}
printf("\nElement %d not found\n",num);
}/*End of del()*/
```

Q. 4(b) Explain the concept of threaded binary search tree. Show the declaration of a node in threaded binary search tree ? Write a function for inorder traversal of threaded binary search tree. **(10 Marks)**

Ans. :

Threaded Binary Tree :

Inorder traversal of a Binary tree is either be done using recursion or with the use of a auxiliary stack. The idea of threaded binary trees is to make inorder traversal faster and do it without stack and without recursion. A binary tree is made threaded by making all right child pointers that would normally be NULL point to the inorder successor of the node (if it exists).

There are two types of threaded binary trees. Single Threaded: Where a NULL right pointers is made to point to the inorder successor (if successor exists)

Double Threaded: Where both left and right NULL pointers are made to point to inorder predecessor and inorder successor respectively. The predecessor threads are useful for reverse inorder traversal and postorder traversal.

The threads are also useful for fast accessing ancestors of a node. Fig. 1-Q. 4(b) shows an example of Single Threaded Binary Tree. The dotted lines represent threads.

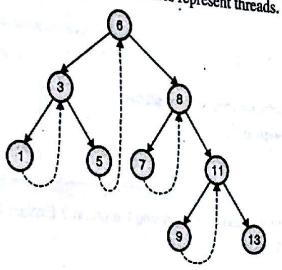


Fig. 1-Q. 4(b)

Following is C representation of a single threaded node.

```
struct Node
{
  int data;
  Node *left, *right;
  bool rightThread;
}
```

Following is C code for inorder traversal in a threaded binary tree.

```
// Utility function to find leftmost node in atree rooted with n
struct Node* leftMost(struct Node *n)
{
  if (n == NULL)
    return NULL;

  while (n->left != NULL)
    n = n->left;

  return n;
}

// C code to do inorder traversal in a threadded binary tree
void inOrder(struct Node *root)
{
  struct Node *cur = leftmost(root);
  while (cur != NULL)
  {
    printf("%d ", cur->data);
```

```
    // If this node is a thread node, then go to
    // inorder successor
    if (cur->rightThread)
      cur = cur->right;
    else // Else go to the leftmost child in right subtree
      cur = leftmost(cur->right);
  }
}
```

Q. 5(a) What are different methods for traversing the graph ? Explain DFS in detail with an example. Write a function for DFS. **(10 Marks)**

Ans. :

Traversing the graph :

Many of the times we want to visit each node of the graph in some specific order. Here traversing means visit each and every node or vertices of the graph exactly once. There are basically two techniques used for traversing of graph and they are as follows,

i) Depth First Search (DFS)
ii) Breadth First Search (BFS).

Algorithm for Depth First Search (DFS) :

1) START
2) Mark the node as visited.
3) Display the node number.
4) Repeat following steps for all routes.
   if there is an edge from start node to current node and current node is not visited.
5) START Depth First Search from current node i.e. restart from step number 2 where current node becomes starting node.

Example : Find DFS spanning tree of following graph G.

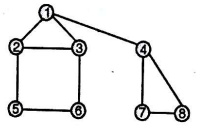


Fig. 1-Q. 5(a) : Graph G

Solution : Just perform according to algorithm

Step 1 : Initially no node is visited. So create a list for our convenience as follows. (Mark all nodes as not visited [nv]).

1	2	3	4	5	6	7	8
nv	nv	nv	nv	nv	nv	nv	nv

Step 2 : Suppose we start from node 1 i.e. DFS (1) then mark 1 as visited. Then find adjacent nodes of node 1 and check whether it is visited or not. If not then apply DFS on it.

DFS (1)
W (1) = {2, 3, 4} // Adjacent nodes of node 1
Mark [2] = not visited
List after step 2.

1	2	3	4	5	6	7	8
v	nv	nv	nv	nv	nv	nv	nv

Step 3 : Node 2 is not visited then apply DFS algorithm on it as follows.

DFS (2)
Mark [2] = visited ;
W (2) = {1, 3, 5}

Note : Discard 1 from W (2) because we already visited. Mark [3] = not visited.

List after step 3

1	2	3	4	5	6	7	8
v	v	nv	nv	nv	nv	nv	nv

Step 4 : Our node is 3, apply DFS algorithm on it.

DFS [3]
Mark [3] = visited
W (3) = {1, 2, 6}
Mark [6] = not visited.

Note : Discard 1, 2 from W (3) because we already visited.

List after step 4.

1	2	3	4	5	6	7	8
v	v	v	nv	nv	nv	nv	nv

Step 5 : Now node is 6. Apply DFS algorithm on it.

DFS [6]
Mark [6] = visited
W (6) = {3, 5}
Mark [5] = not visited.

Note : Discard 3 from W (6) because we already visited.

List after step 5.

1	2	3	6	4	5	7	8
v	v	v	v	nv	nv	nv	nv

Step 6 : Now node is 5 apply DFS algorithm on it.

DFS [5]
Mark [5] = visited
W [5] = {2, 6}

W [1] = {2, 3, 4} Here 2 and 3 visited already. Mark [4] = not visited.

List after step 6.

1	2	3	6	5	4	7	8
v	v	v	v	v	nv	nv	nv

Step 7 : Now node is 4 apply DFS algorithm on it.

DFS [4]

Mark [4] = visited

W [4] = {1, 7, 8}

Mark [7] = not visited.

List after step 7.

1	2	3	6	5	4	7	8
v	v	v	v	v	v	nv	nv

Step 8 : Now node is 7, apply DFS algorithm on it.

DFS [7]

Mark [7] = visited

W [7] = {4, 8}

Mark [8] = not visited.

Note : We discard 4 from W [7] because we already visited node 4.

List after step 8.

1	2	3	6	5	4	7	8
v	v	v	v	v	v	v	nv

Step 9 : Now node is 8, apply DFS algorithm on it.

DFS [8]

Mark [8] = visited

W [8] = {4, 7}

List after step 9.

1	2	3	6	5	4	7	8
v	v	v	v	v	v	v	v

From above list it is clearly seen that all nodes are already visited.

Generate DFS spanning tree by using list created after step 9 which as follows.

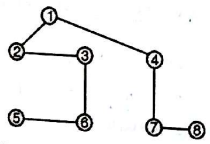


Fig. 2-Q. 5(a) : DFS spanning tree of Fig. 1-Q. 5(a)


```
int DFS(int v)
{
int i;
visited[1]=true;
printf("%d",v);
for(i=1;i<=n;i++)
{
  if(a[v][i]==1 && visited[i]==0)
  {
      DFS(i);
  }
}
}
```

Q. 5(b) Write a function for creating a tree if IN-ORDER traversal and POST-OREDER traversal of a tree is given. **(10 Marks)**

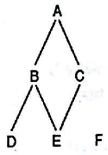
Ans. : Construct Tree from given Inorder and Preorder traversals

Let us consider the below traversals :

Inorder sequence: D B E A F C, Preorder sequence: A B D E C F

In a Preorder sequence, leftmost element is the root of the tree. So we know 'A' is root for given sequences. By searching 'A' in Inorder sequence, we can find out all elements on left side of 'A' are in left subtree and elements on right are in right subtree. So we know below structure now.

We recursively follow above steps and get the following tree.




```
/* program to construct tree using inorder and preorder traversals */
#include<stdio.h>
#include<stdlib.h>

/* A binary tree node has data, pointer to left child
  and a pointer to right child */
struct node
{
```

```
  char data;
  struct node* left;
  struct node* right;
};

/* Prototypes for utility functions */
int search(char arr[], int strt, int end, char value);
struct node* newNode(char data);

/* Recursive function to construct binary of size len from
   Inorder traversal in[] and Preorder traversal pre[]. Initial values
   of inStrt and inEnd should be 0 and len -1. The function doesn't
   do any error checking for cases where inorder and preorder
   do not form a tree */
struct node* buildTree(char in[], char pre[], int inStrt, int inEnd)
{
  static int preIndex = 0;

  if(inStrt > inEnd)
    return NULL;

  /* Pick current node from Preorder traversal using preIndex
     and increment preIndex */
  struct node *tNode = newNode(pre[preIndex++]);

  /* If this node has no children then return */
  if(inStrt == inEnd)
    return tNode;

  /* Else find the index of this node in Inorder traversal */
  int inIndex = search(in, inStrt, inEnd, tNode->data);

  /* Using index in Inorder traversal, construct left and
     right subtress */
  tNode->left = buildTree(in, pre, inStrt, inIndex-1);
  tNode->right = buildTree(in, pre, inIndex+1, inEnd);

  return tNode;
}

/* UTILITY FUNCTIONS */
```



```
/* Function to find index of value in arr[start...end]
   The function assumes that value is present in in[] */
int search(char arr[], int strt, int end, char value)
{
  int i;
  for(i = strt; i <= end; i++)
  {
    if(arr[i] == value)
      return i;
  }
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(char data)
{
  struct node* node = (struct node*)malloc(sizeof(struct node));
  node->data = data;
  node->left = NULL;
  node->right = NULL;

  return(node);
}

/* This funtcion is here just to test buildTree() */
void printInorder(struct node* node)
{
  if (node == NULL)
    return;

  /* first recur on left child */
  printInorder(node->left);

  /* then print the data of node */
  printf("%c ", node->data);

  /* now recur on right child */
  printInorder(node->right);
}

/* Driver program to test above functions */
```

```
int main()
{
 char in[] = {'D', 'B', 'E', 'A', 'F', 'C'};
 char pre[] = {'A', 'B', 'D', 'E', 'C', 'F'};
 int len = sizeof(in)/sizeof(in[0]);
 struct node *root = buildTree(in, pre, 0, len - 1);

 /* Let us test the built tree by printing Insorder traversal */
 printf("Inorder traversal of the constructed tree is \n");
 printInorder(root);
 getchar();
}
```

Chapter 10 : Searching and Sorting [Total Marks : 20]

Q. 6(a) Write an algorithm for shell sort. Sort the following numbers in asending order 23, 12, 45, 54, 76, 67, 88, 97, 54 using shell sort. Show output after each pass. **(10 Marks)**

Ans. : Shell Sort :

Shell sort is an algorithm which is improvement or say generalization of insertion sort algorithm.

Insertion sort algorithm is inefficient since it moves one position at a time i.e. insertion sort algorithm is efficient for the input which is almost in sorted order but inefficient for big size arrays. The shell sort method or algorithm was put forth by D. L. Shell in 1959. This sorting algorithm was simple to implement, easy to understand and fast.

Basically the shell sort algorithm is based on or works on following to principals,

(i) Arrange the given data list or array in 2-dimentional array.

(ii) Sort this 2-dimentional array column wise.

The above procedure or principles sort the given data partially. This procedure is repeated each time with the narrower array

Given numbers

23, 12, 45, 54, 76, 67, 88, 97, 54.

Step 1 : We arrange given numbers in 5 columns and sorted towards right.

						Sorted				
23	12	45	54	76	→	23	12	45	54	76
67	88	97	54			67	88	97	54	

Step 2 : We arrange given numbers in 3 columns and sorted towards right.

				Sorted		
23	12	45		23	12	45
54	76	67	→	54	76	54
88	97	54		88	97	67

Step 3 : We arrange given numbers in 2 columns and sorted towards right.

		Sorted	
23	12	23	12
45	54 →	45	54
76	54	67	54
88	97	76	97
67		88	

Above list is almost completely sorted. When we arranging it in one column only, 12, 54 and 88 move a little bit to get proper place. The sorted array is as follows :

12, 23, 45, 54, 54, 67, 76, 88, 97.

Q. 6(b) Explain index sequential search with an example.

Ans. : Please refer Q. 5(b) of May 2015. **(10 Marks)**

□□□

May 2016

Chapter 1 : Types of Data Structure and Recursion [Total Marks : 03]

Q. 1(a) Define ADT with an example. **(3 Marks)**

Ans. : **Please refer Q. 1(d) of Dec. 2013.**

Chapter 4 : File Handling in C [Total Marks : 10]

Q. 2(a) Discuss file I/O in C language with different library functions. **(10 Marks)**

Ans. : A file represents a sequence of bytes, regardless of it being a text file or a binary file. C programming language provides access on high level functions as well as low level (OS level) calls to handle file on your storage devices. This chapter will take you through the important calls for file management.

Opening Files :

You can use the **fopen( )** function to create a new file or to open an existing file. This call will initialize an object of the type **FILE**, which contains all the information necessary to control the stream. The prototype of this function call is as follows :

```
FILE *fopen( const char * filename, const char * mode );
```

Here, **filename** is a string literal, which you will use to name your file, and access **mode** can have one of the following values –

Mode	Description
r	Opens an existing text file for reading purpose.
w	Opens a text file for writing. If it does not exist, then a new file is created. Here your program will start writing content from the beginning of the file.
a	Opens a text file for writing in appending mode. If it does not exist, then a new file is created. Here your program will start appending content in the existing file content.
r+	Opens a text file for both reading and writing.
w+	Opens a text file for both reading and writing. It first truncates the file to zero length if it exists, otherwise creates a file if it does not exist.
a+	Opens a text file for both reading and writing. It creates the file if it does not exist. The reading will start from the beginning but writing can only be appended.

If you are going to handle binary files, then you will use following access modes instead of the above mentioned ones –

```
"rb", "wb", "ab", "rb+", "r+b", "wb+", "w+b", "ab+", "a+b"
```

Chapter 5 : Linked List [Total Marks : 17]

Q. 1(b) What are the advantages of using linked lists over arrays ? **(5 Marks)**

Ans. :

Advantages of linked lists over arrays :

Linked list have many advantages. Some of them are as follows,

1. Linked list are dynamic data structure. i.e. they can grow or shrink during the program execution.

2. Utilization of memory : In linked list memory is not preallocated. Memory is allocated as per requirement and it is deallocated when it is no longer needed.
3. The insertion and deletion operations are easier and effective. Linked list provides flexibility in inserting a data item at a specified position and deletion of data item from the given position.
4. Many complex applications can be easily carried out with the help of linked list.

Q. 4(a) Write a C progeram to implement a doubly linked list which performs the following operations.

(i) Inserting element in the beginning
(ii) Inserting element in the end
(iii) Inserting element after an element
(iv) Deleting a particular element
(v) Displaying the list

(12 Marks)

Ans. :

Implementation of Doubly linked list :

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

struct node
{
struct node *prev;
int info;
struct node *next;
}*start;

main()
{
int choice,n,m,po,i;
start=NULL;
clrscr();
printf("\n\tDouble Linked List");
printf("\n_____\n\n");
while(1)
{
   printf("\t1.Create List\n");
   printf("\t2.Add at begining\n");
   printf("\t3.Add after\n");
   printf("\t4.Delete\n");
   printf("\t5.Display\n");
   printf("\t6.Count\n");
   printf("\t7.Reverse\n");
   printf("\t8.exit\n");
```

```
    printf("\tEnter your choice : ");
    scanf("%d",&choice);
    switch(choice)
    {
    case 1:
        printf("\nHow many nodes you want : ");
        scanf("%d",&n);
        for(i=0;i<n;i++)
        {
            printf("\nEnter the element : ");
            scanf("%d",&m);
            create_list(m);
        }
        break;
    case 2:
        printf("\nEnter the element : ");
        scanf("%d",&m);
        addatbeg(m);
        break;
    case 3:
        printf("\nEnter the element : ");
        scanf("%d",&m);
        printf("Enter the position after which this element is inserted : ");
        scanf("%d",&po);
        addafter(m,po);
        break;
    case 4:
        printf("\nEnter the element for deletion : ");
        scanf("%d",&m);
        del(m);
        break;
    case 5:
        display();
        break;
    case 6:
        count();
        break;
    case 7:
        rev();
        break;
    case 8:
```



```
        exit();
    default:
        printf("\nWrong choice\n");
}/*End of switch*/
  }/*End of while*/
}/*End of main()*/

create_list(int num)
{
struct node *q,*tmp;
tmp= malloc(sizeof(struct node));
tmp->info=num;
tmp->next=NULL;
if(start==NULL)
{
   tmp->prev=NULL;
   start->prev=tmp;
   start=tmp;
}
else
{
   q=start;
   while(q->next!=NULL)
        q=q->next;
   q->next=tmp;
   tmp->prev=q;
}
}/*End of create_list()*/

addatbeg(int num)
{
struct node *tmp;
tmp=malloc(sizeof(struct node));
tmp->prev=NULL;
tmp->info=num;
tmp->next=start;
start->prev=tmp;
start=tmp;
}/*End of addatbeg()*/

addafter(int num,int c)
{
struct node *tmp,*q;
int i;
q=start;
for(i=0;i<c-1;i++)
{
```

```
    q=q->next;
    if(q==NULL)
    {
        printf("\nThere are less than %d elements\n",c);
        return;
    }
}
tmp=malloc(sizeof(struct node) );
tmp->info=num;
q->next->prev=tmp;
tmp->next=q->next;
tmp->prev=q;
q->next=tmp;
}/*End of addafter() */

del(int num)
{
struct node *tmp,*q;
if(start->info==num)
{
    tmp=start;
    start=start->next;  /*first element deleted*/
    start->prev = NULL;
    free(tmp);
    return;
}
q=start;
while(q->next->next!=NULL)
{
    if(q->next->info==num)    /*Element deleted in between*/
    {
        tmp=q->next;
        q->next=tmp->next;
        tmp->next->prev=q;
        free(tmp);
        return;
    }
    q=q->next;
}
if(q->next->info==num)   /*last element deleted*/
{   tmp=q->next;
    free(tmp);
    q->next=NULL;
    return;
}
printf("\nElement %d not found\n",num);
}/*End of del()*/
```



```
display()
{
struct node *q;
if(start==NULL)
{
    printf("\nList is empty\n");
    return;
}
q=start;
printf("\nList is :\n");
while(q!=NULL)
{
    printf("%d ", q->info);
    q=q->next;
}
printf("\n");
}/*End of display() */

count()
{   struct node *q=start;
int cnt=0;
while(q!=NULL)
{
    q=q->next;
    cnt++;
}
printf("\nNumber of elements are %d\n",cnt);
}/*End of count()*/

rev()
{
struct node *p1,*p2;
p1=start;
p2=p1->next;
p1->next=NULL;
p1->prev=p2;
while(p2!=NULL)
{
    p2->prev=p2->next;
    p2->next=p1;
    p1=p2;
    p2=p2->prev; /*next of p2 changed to prev */
}
start=p1;
}/*End of rev()*/
```


Output :

```
Double Linked List
------------------
   1. Create List
   2. Add at beginning
   3. Add after
   4. Delete
   5. Display
   6. Count
   7. Reverse
   8. exit
   Enter your choice : 1
How many nodes you want : 4
Enter the element : 95
Enter the element : 99
Enter the element : 14
Enter the element : 18
   1. Create List
   2. Add at beginning
   3. Add after
   4. Delete
   5. Display
   6. Count
   7. Reverse
   8. exit
   Enter your choice : 2
Enter the element : 27
   1. Create List
   2. Add at beginning
   3. Add after
   4. Delete
   5. Display
   6. Count
   7. Reverse
   8. exit
   Enter your choice : 5_
```


Output Continue :

```
List is :
27  95  99  14  18
   1. Create List
   2. Add at beginning
   3. Add after
   4. Delete
   5. Display
   6. Count
   7. Reverse
   8. exit
   Enter your choice : 3
Enter the element : 41
Enter the position after which this element is inserted : 3
   1. Create List
   2. Add at beginning
   3. Add after
   4. Delete
   5. Display
   6. Count
   7. Reverse
   8. exit
   Enter your choice : 5
List is :
27  95  99  41  14  18
   1. Create List
   2. Add at beginning
   3. Add after
   4. Delete
   5. Display
   6. Count
   7. Reverse
   8. exit
   Enter your choice : 4
Enter the element for deletion : 99
   1. Create List
   2. Add at beginning
   3. Add after
   4. Delete
   5. Display
   6. Count
   7. Reverse
   8. exit
   Enter your choice : 5
```

Output Continue :

```
List is :
27  95  41  14  18
        1.  Create List
        2.  Add at beginning
        3.  Add after
        4.  Delete
        5.  Display
        6.  Count
        7.  Reverse
        8.  exit
        Enter your choice : 6
Number of elements are 5
        1.  Create List
        2.  Add at beginning
        3.  Add after
        4.  Delete
        5.  Display
        6.  Count
        7.  Reverse
        8.  exit
        Enter your choice : 7
        1.  Create List
        2.  Add at beginning
        3.  Add after
        4.  Delete
        5.  Display
        6.  Count
        7.  Reverse
        8.  exit
        Enter your choice : 5
List is :
18  14  41  95  27
        1.  Create List
        2.  Add at beginning
        3.  Add after
        4.  Delete
        5.  Display
        6.  Count
        7.  Reverse
        8.  exit
        Enter your choice : 8_
```


Chapter 6 : Stack [Total Marks : 18]

Q. 2(b) Explain recursion as an application of stack with examples. **(10 Marks)**

Ans. :

Recursion as an application of stack :

Stack is not only used to perform recursion but any bunch of nested function calls. Recursion is a special case wherein all the nested function calls are to the same function, or the same function is called in a cyclic chain of function calls. Nature of nested function calling and returning is the same as that of a stack. The one that is called at top level(or the earliest) is returned the last. The later a function is called in the nested hierarchy, the earliest it returns. So, the start/ return of a function call turns out analogous to push/ pop operations of a stack and thus a stack can be used to mimic this functionality.

Recursion, in mathematics and computer science, is a method of defining functions in which the function being defined is applied within its own definition. The term is also used more generally to describe a process of repeating objects in a self-similar way.

Factorial : A classic example of a recursive procedure is the function used to calculate the factorial of an integer.

Function Definition : A recurrence relation is an equation that relates later terms in the sequence to earlier terms.

Recurrence relation for factorial: bn = n * bn-1 b0 = 1

Computing the recurrence relation for n = 4:

b4 = 4 * b3 = 4 * 3 * b2 = 4 * 3 * 2 * b1 = 4 * 3 * 2 * 1 * b0 = 4 * 3 * 2 * 1 * 1 = 4 * 3 * 2 * 1 = 4 * 3 * 2 = 4 * 6 = 4 * 6 = 24.

```
#include <stdio.h>
long int multiplyNumbers(int n);
int main()
{
  int n;
  printf("Enter a positive integer: ");
  scanf("%d", &n);
  printf("Factorial of %d = %ld", n, multiplyNumbers(n));
  return 0;
}
long int multiplyNumbers(int n)
{
  if (n >= 1)
    return n*multiplyNumbers(n-1);
  else
    return 1;
}
```

Output

```
Enter a positive integer: 4
Factorial of 4 = 24
```

Q. 5(a) Explain any one application of linked list with an example. **(8 Marks)**

Ans. :

Application of linked list :

We implement sparse matrices using linked list as shown below.

Sparse Matrices :

Most problems can be modeled by a system of linear equations.

(1) Thousands of equations (constraints)
(2) Thousands of variables (unknowns)
(3) It is not practical to allocate n by n space to store a linear system.
(4) How much space is necessary for a matrix of size 10000 × 10000 where each entry is a double ?
(5) Sparse matrix representation.

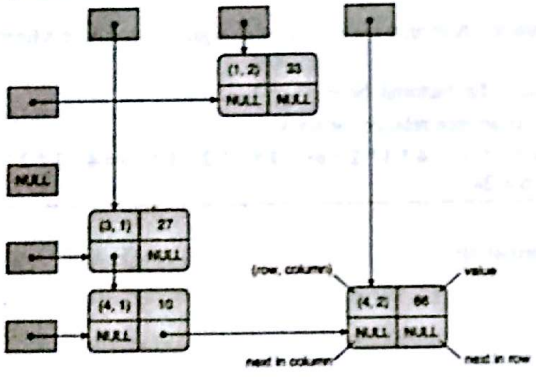


Fig. 1-Q. 5(a)

Designing the node for sparse matrix :

```
Public class node

{

    public comparable data;

    public int row, col

    public node nextrow;

    public node nextcol;
```



```
    Node(int row, int col, comparable data) {

}

public string to string() {return data toString();}

}
```

Other Advanced Operation on Linked Lists

Reversing a linked List

Reversing a Linked List

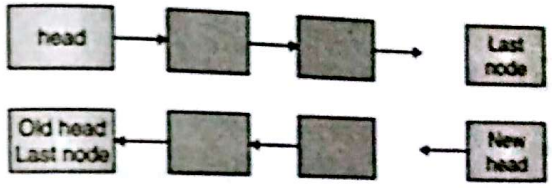


Fig. 2-Q. 5(a)

Cloning of linked List

Let us begin by evaluating the following code :

```
LinkedList list = new LinkedList ();
list.prepend (new Node ("first"));
list.appended (new Node ("last"));
LinkedList list2 = list;
```

The reference list 2 is an alias, it points to the same object as list 1. This means that once we change list, these changes occur in list 2 as well. In this case, we consider list 2 as a shallow copy of list1. In other words, list 1 and list 2 shares the same nodes in one linked list. In many cases it will be quite useful to have a deep copy of an object. To create a deep copy, we need to duplicate all nodes in the list. For this purpose, the object class provided the methods clone(). Which we will override in the Linked List class;

Chapter 7 : Queues [Total Marks : 12]

Q. 3(a) Write a menu driven program in C to implement QUEUE ADT. The program should perform the following operations : **(12 Marks)**

(i) Inserting an elements in the QUEUE
(ii) Deleting an element from the Queue
(iii) Displaying the queue
(iv) Exiting the program.

Ans. :

```
#include<iostream>
#include<conio.h>
#include<stdlib.h>
using namespace std;

class queue
{
        int queue1[5];
        int rear,front;
    public:
        queue()
         {
            rear=-1;
            front=-1;
         }
        void insert(int x)
        {
           if(rear >  4)
            {
              cout <<"queue over flow";
              front=rear=-1;
              return;
            }
            queue1[++rear]=x;
            cout <<"inserted" <<x;
         }
        void delet()
        {
           if(front==rear)
            {
               cout <<"queue under flow";
               return;
            }
            cout <<"deleted" <<queue1[++front];
         }
        void display()
         {
           if(rear==front)
            {
                cout <<" queue empty";
```



```
                return;
            }
            for(int i=front+1;i<=rear;i++)
            cout <<queue1[i]<<" ";
        }
};

main()
{
   int ch;
   queue qu;
   while(1)
    {
        cout <<"\n1.insert  2.delet  3.display  4.exit\nEnter ur choice";
        cin >> ch;
        switch(ch)
         {
          case 1:    cout <<"enter the element";
                     cin >> ch;
                qu.insert(ch);
                break;
          case 2:  qu.delet();  break;
          case 3:  qu.display();break;
          case 4: exit(0);
          }
    }
return (0);
}
```

output

```
1. insert
2. delet
3. display
4. exit
Enter ur choice1
enter the element21
inserted21
1.insert 2.delet 3.display 4.exit
Enter ur choice1
enter the element22
inserted22
```

```
1.insert 2.delet 3.display 4.exit
Enter ur choice1
enter the element16
inserted16
1.insert 2.delet 3.display 4.exit
Enter ur choice3
21 22 16
1.insert 2.delet 3.display 4.exit
Enter ur choice2
deleted21
1.insert 2.delet 3.display 4.exit
Enter ur choice3
22 16
```

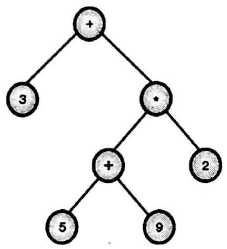
Chapter 8 : Trees [Total Marks : 17]

Q. 1(c) Describe Expression tree with an example. **(5 Marks)**

Ans. :

Expression Tree :

Expression tree is a binary tree in which each internal node corresponds to operator and each leaf node corresponds to operand so for example expression tree for 3 + ((5+9)*2) would be :



Construction of Expression Tree :

Now For constructing expression tree we use a stack. We loop through input expression and do following for every character. 1) If character is operand push that into stack 2) If character is operator pop two values from stack make them its child and push current node again. At the end only element of stack will be root of expression tree.

Q. 5(b) Write a program in C to delete a node from a binary search tree. The program should consider all the possible cases. **(12 Marks)**

And. : **Please refer Q. 6(b) of Dec. 2013.**

Q. 6(a) Write a program in C to implement the BFS traversal of a graph. Explain the code with a example.

Ans. : **(10 Marks**

C program to implement BFS traversal of a graph :

```
#include<stdio.h>
#include<conio.h>
#include<alloc.h>
#define true 1
#define false 0
#define MAX 20
int visited[MAX];
int a[MAX][MAX];
int n;
void main()
{
 int i,v,choice;
 clrscr();
 creategraph();
 display();
printf("\n Enter Starting node for Breadth First Search \n");
scanf("%d",&v);
for(i=1;i<n;i++)
{
   visited[i]=false;
}
 BFS(v);
getch();
}
int creategraph()
{
int i,maxedges,start,end;
printf("\n Enter number of nodes \n");
scanf("%d",&n);
maxedges=n*(n-1);
for(i=1;i<=maxedges;i++)
{
   printf("\n Enter edge (0 0 to quit) \n");
   scanf("%d%d",&start,&end);
   if(start ==0 && end ==0)
```

```
    {
        break;
    }
    else if(start>n||end>n||start<=0||end<=0)
    {
        printf("\n Invalid Edge");
        i--;
    }
    else
    {
        a[start][end]=1;
    }
}
}
int display()
{
int i,j;
for(i=1;i<=n;i++)
{
    printf("%d",a[i][j]);
}
printf("\n\n");
}
int BFS(int v)
{
int front,i,rear;
int Queue[20];
front=rear=-1;
printf("%d",v);
visited[v]=1;
rear++;
front++;
Queue[rear]=v;
while(front<=rear)
{
    v=Queue[front];
    front++;
    for(i=1;i<=n;i++)
    {
        if(a[v][i]==1 && visited[i]==0)
        {
```



```
            printf("%d",i);
            visited[i]=1;
            rear++;
            Queue[rear]=i;
        }
        }
}
}
```

Example : Find BFS spanning tree for following graph G.

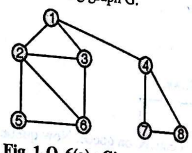


Fig. 1-Q. 6(a) : Given graph G

Solution :

Just follow the steps according to BFS algorithm.

Trace [1 ..... 8] n = no. of vertices.

Mark all vertices as not visited at initial stage. And apply BFS algorithm on each node.

1	2	3	4	5	6	7	8
nv	nv	nv	nv	nv	nv	nv	nv

At initial stage Queue Q is empty.

Q. | |

BFS (1) Apply BFS algorithm on node 1.

Mark [1] = visited.

Trace [1] = 1.

Enter 1 in Queue Q.

Q. | 1 | |

Now Queue Q is not empty. So remove first element from queue i.e. u = 1

u = 1 Pop 1 from queue

Q | |

Find each vertex adjacent to node 1.

W (1) = {2, 3, 4}

Mark [2] = = not visited

Trace [2] = = Trace [1] + 1 = 1 + 1 = 2 Print 2.

Now Enter 2 into queue Q.

Q | 2 | |

Mark [3] = = not visited.

Trace [3] = = Trace [1] + 1
Trace [3] = = 1 + 1
Trace [3] = = 2
Print 3.
Now status of queue is as follows,

Q	2	3	

Mark [4] = = not visited
Trace [4] = = Trace [1] + 1
Trace [4] = = 1 + 1
Trace [4] = = 2
Print 4.
Now status of queue is as follows.

Q	2	3	4	

All adjacent nodes of node 1 is entered on queue. Now queue is not empty. Remove first element from queue i.e. u = 2.
Now status of Q is

Q	3	4	

Find adjacent nodes of node 2.
W [2] = {1, 3, 5, 6}
Now our next node is 5.
(Mark [5] = = Not visited)
Trace [5] = = Trace [2] + 1 = 2 + 1 = 3
Print 5.
Now status of Queue is as follows,

Q	3	4	5	

Mark [6] = = not visited.
Trace [6] = = Trace [2] + 1
Trace [6] = = 2 + 1
Trace [6] = = 3
Print 6.
Now status of Queue is as follows,

Q	3	4	5	6	

All adjacent nodes of node 2 is entered on queue. Now queue is not empty. Remove first element from queue i.e. u = 3.
Now status of queue is,

Q	4	5	6	

Find adjacent nodes of node 3.
W [3] = {1, 2, 6}

Now remove first element from queue i.e. u = 4. Status of queue is as follows.

Q	5	6	

Find adjacent nodes of node 4.
W [4] = {1, 7, 8}
Mark [7] = = not visited
Trace [7] = = Trace [4] + 1
Trace [7] = = 3 + 1
Trace [7] = = 4
Print 7.
Now status of Q is as follows,

Q	5	6	7	

Mark [8] = = not visited
Trace [8] = = Trace [4] + 1
Trace [8] = = 3 + 1
Trace [8] = = 4
Print 8.
Now status of queue is as follows,

Q	5	6	7	8	

Now queue is not empty. Remove first element from queue i.e. u = 5. Now queue status is as follows,

Q	6	7	8	

Find adjacent nodes of node 5.
W [5] = {2, 6}
We already visited 2 and 6 so discard it.
Now remove first element from Queue i.e. u = 6. Queue status is as follows,

Q	7	8	

Find adjacent nodes of node 6.
W [6] = {2, 3, 5}
We already visited 2, 3 and 5 so discard it.
Now remove first element from queue i.e. u = 7. Queue status is as follows,

Q	8	

Find adjacent nodes of node 7.
W [7] = {4, 8}
We already visited 4 and 8 so discard it.
Now remove first element from queue i.e. u = 8. Queue status is as follows.

Q	

Find adjacent of node 8.

W [8] = {4, 7}

We already visited 4 and 7 so discard it.

Now queue is empty and we visit all the vertices So BFS algorithm terminates. We get trace as follows,

1	1	1	1	2	2	4	4
1	2	3	4	5	6	7	8

← Trace value at that Node.

By using this trace we find BFS spanning tree. Which is as follows,

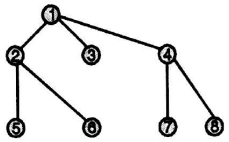


Fig. 2-Q. 6(a) : BFS spanning tree of Fig. 1-Q. 6(a)

Chapter 10 : Searching and Sorting [Total Marks : 25]

Q. 1(d) Write a program in C to implement insertion sort. **(7 Marks)**

Ans. : Program of sorting elements by using insertion sort algorithm :

```
#include<stdio.h>
#include<conio.h>
# define max 20
void main()
{
int arr[max],i,j,k,n;
clrscr();
printf("enter the number of element you want to enter:");
scanf("%d",&n);
for(i=0;i<n;i++)
{
  printf("enter the element %d:",i+1);
  scanf("%d",&arr[i]);
}
  printf("\nunsorted list is:\n");
  for(i=0;i<n;i++)
  printf("%d\t",arr[i]);
  printf("\n");
  //Insertion sort
  for(j=1;j<n;j++)
  {
```



```
k=arr[j];  //k is to be inserted at proper place
for(i=j-1;i>=0&&k<arr[i];i--)
arr[i+1]=arr[i];
arr[i+1]=k;
printf("\npass %d element inserted in the proper place:%d\n",j,k);
for(i=0;i<n;i++)
printf("%d\t",arr[i]);
printf("\n");
}
printf("\n\nsorted list is:\n");
for(i=0;i<n;i++)
printf("%d\t",arr[i]);
printf("\n");
getch();
}
```

Output :

```
enter the number of element you want to enter : 5
enter the element 1 : 95
enter the element 2 : 23
enter the element 3 : 18
enter the element 4 : 99
enter the element 5 : 14
unsorted list is :
95      23      18      99      14
pass 1 element inserted in the proper place : 23
23      95      18      99      14
pass 2 element inserted in the proper place : 18
18      23      95      99      14
pass 3 element inserted in the proper place : 99
18      23      95      99      14
pass 4 element inserted in the proper place : 14
14      18      23      95      99
sorted list is :
14      18      23      95      99
```

Q. 3(b) Write a function to implement indexed sequential search. Explain with an example. (8 Marks)

Ans. : Indexed Sequential search :

Indexed Sequential Access Method is a file management system developed at IBM that allows records to be accessed either sequentially (in the order they were entered) or randomly (with an index). Each index defines a different ordering of the records. An employee database may have several indexes, based on the information being sought. For example, a name index may order employees alphabetically by last name, while a department index may order employees by their department. A key is specified in each index. For an alphabetical index of employee names, the last name field would be the key.

Function to implement sequential search :

```
#include <stdio.h>
#include <conio.h>
#define MAX 10
struct mainfile
{int empid;
char name[25];
float basic;
};
struct indexfile
{int index_id;
int kindex;
};
void main()
{
struct mainfile fobj[MAX];
struct indexfile index[MAX];
int i,num,low,high,ct=4;
float basicsal;
clrscr();
for(i=0;i<MAX;i++)
{
printf("\nEnter employee id?");
scanf("%d",&fobj[i].empid);
printf("\nEnter name?");
scanf("%s",fobj[i].name);
printf("\nEnter basic?");
scanf("%f",&basicsal);
fobj[i].basic=basicsal;
}
printf("\nNow creating index file...!");
for(i=0;i<(MAX/5);i++)
{index[i].index_id=fobj[ct].empid;
index[i].kindex = ct;
ct=ct+5;
}
```



```
printf("\n\nEnter the empid to search?");
scanf("%d",&num);
for(i=0;(i<MAX/5) && (index[i].index_id<=num);i++);
low=index[i-1].kindex;
high=index[i].kindex;
for(i=low;i<=high;i++)
{if(num==fobj[i].empid)
{
printf("\nThe record is: \n\t");
printf("\nEmpid: %d",fobj[i].empid);
printf("\nName: %s",fobj[i].name);
printf("\nBasic: %f",fobj[i].basic);
getch();
return;
}
}
printf("\nNumber not found...!");
return;
}
```

Q. 6(b) Hash the following in a table of size 11. Use any two collision resolution techniques.

23, 55, 10, 71, 67, 32, 100, 18, 10, 90, 44 (10 Marks)

Ans. :

(i) Given key elements
23, 55, 10, 71, 67, 32, 100, 18, 10, 90, 44

(ii) Size of Hash table – 11
i.e. → 0 to 10

(iii) We use division – reminder method to solve this example.

$H = k \bmod m$

Where H = hash address
K = key
m = prime number nearest to largest address = (7)

(a) key = 23
H = 23% 7
H = 2

(b) key = 55
H = 55%7
H = 6

(c) key = 10
H = 10%7
H = 3

(d) key = 71
H = 71%7
H = 1

(e) key = 67
H = 67%7
H = 4

(f) key = 32
H = 32%7
H = 4

Key 32 stores at location 4 but already key 67 stored at that location so we use linear probing collision method. i.e. we check that next address is free or not. H = 4 + 01 = 5, 5 address is free, so we store 32 at memory location 5.

(g) key = 100
H = 100% 7
H = 2

collision occurs at location 2. We check for next free location. i.e. 7.
∴ key 100 stored at H = 7.

□□□