Date:…… **SHEET No:……**

EXPERIMENT-1

Aim:

Implement the data link layer framing methods such as character count, character stuffing and bit stuffing.

Description:

The Data Link Layer is the second layer in the OSI model, above the Physical Layer, which ensures that the error free data is transferred between the adjacent nodes in the network. It breaks the datagrams passed down by above layers and convert them into frames ready for transfer. This is called Framing. It provides two main functionalities

- Reliable data transfer service between two peer network layers
- Flow Control mechanism which regulates the flow of frames such that data congestion is not there at slow receivers due to fast senders.

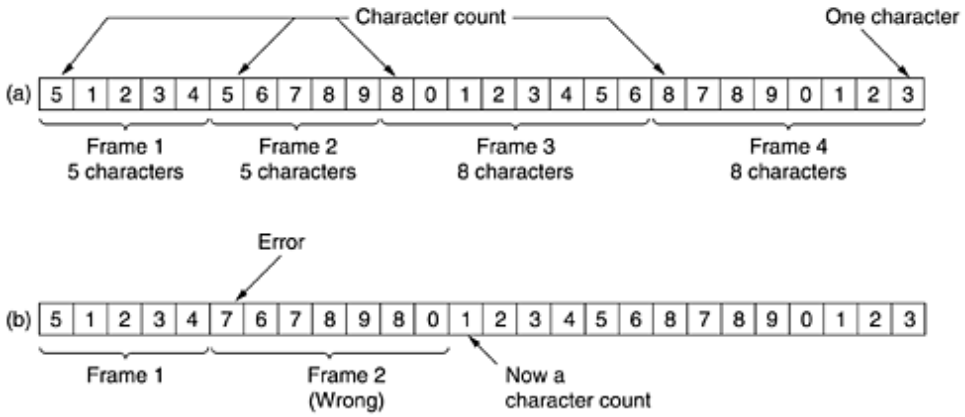
What is Framing?

Since the physical layer merely accepts and transmits a stream of bits without any regard to meaning or structure, it is upto the data link layer to create and recognize frame boundaries. This can be accomplished by attaching special bit patterns to the beginning and end of the frame. If these bit patterns can accidentally occur in data, special care must be taken to make sure these patterns are not incorrectly interpreted as frame delimiters. The four framing methods that are widely used are

- Character count
- Starting and ending characters, with character stuffing
- Starting and ending flags, with bit stuffing
- Physical layer coding violations

Character Count

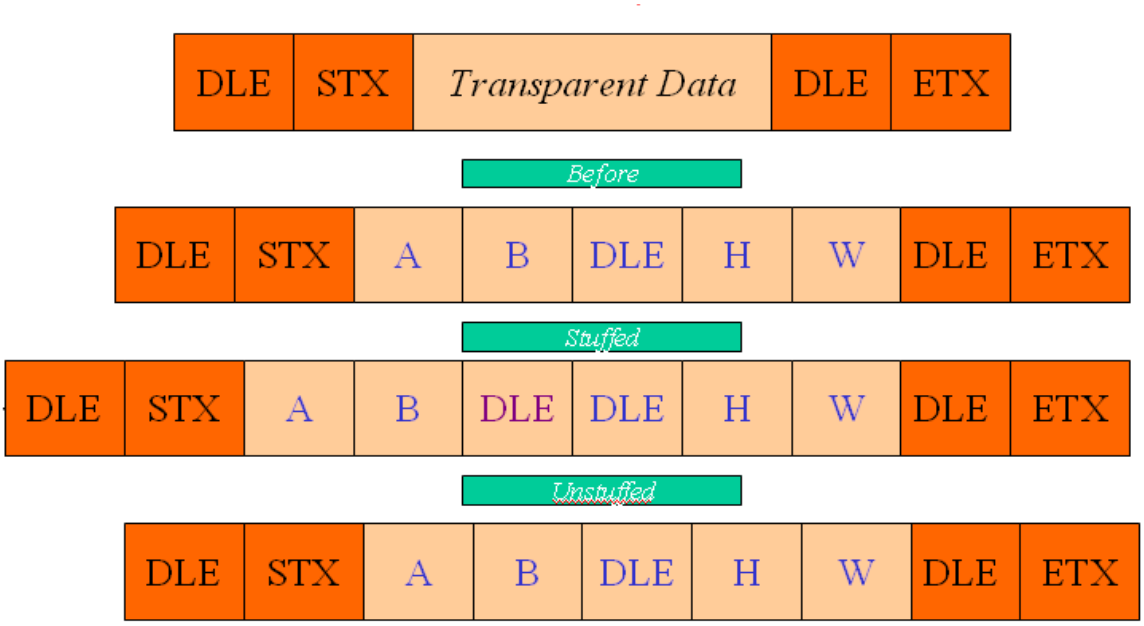
This method uses a field in the header to specify the number of characters in the frame. When the data link layer at the destination sees the character count, it knows how many characters follow, and hence where the end of the frame is. The disadvantage is that if the count is garbled by a transmission error, the destination will lose synchronization and will be unable to locate the start of the next frame. So, this method is rarely used.



A character stream. (a) Without errors. (b) With one error.

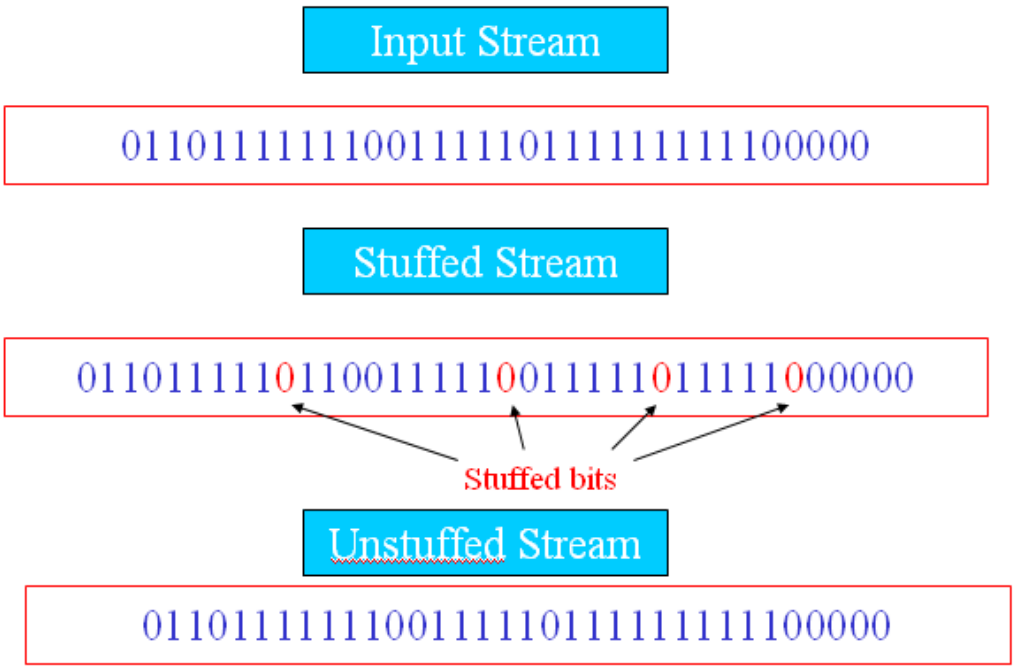
Character stuffing

In the second method, each frame starts with the ASCII character sequence DLE STX and ends with the sequence DLE ETX.(where DLE is Data Link Escape, STX is Start of TeXt and ETX is End of TeXt.) This method overcomes the drawbacks of the character count method. If the destination ever loses synchronization, it only has to look for DLE STX and DLE ETX characters. If however, binary data is being transmitted then there exists a possibility of the characters DLE STX and DLE ETX occurring in the data. Since this can interfere with the framing, a technique called character stuffing is used. The sender's data link layer inserts an ASCII DLE character just before the DLE character in the data. The receiver's data link layer removes this DLE before this data is given to the network layer. However character stuffing is closely associated with 8-bit characters and this is a major hurdle in transmitting arbitrary sized characters.



Bit stuffing

The third method allows data frames to contain an arbitrary number of bits and allows character codes with an arbitrary number of bits per character. At the start and end of each frame is a flag byte consisting of the special bit pattern 01111110 . Whenever the sender's data link layer encounters five consecutive 1s in the data, it automatically stuffs a zero bit into the outgoing bit stream. This technique is called bit stuffing. When the receiver sees five consecutive 1s in the incoming data stream, followed by a zero bit, it automatically destuffs the 0 bit. The boundary between two frames can be determined by locating the flag pattern.



Physical layer coding violations

The final framing method is physical layer coding violations and is applicable to networks in which the encoding on the physical medium contains some redundancy. In such cases normally, a 1 bit is a high-low pair and a 0 bit is a low-high pair. The combinations of low-low and high-high which are not used for data may be used for marking frame boundaries.

Date:…… **SHEET No:……**

Program:

// Character count

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

char input[10][20];
int get_input();

void make_frames(int);
int count_chars(int s);

void main()
{
        int no_of_words=get_input();
        make_frames(no_of_words);
}

int get_input()
{
        int answer;
        int i=0;
        do{
                printf("\nEnter the Word:");
                scanf("%s",input[i]);
                fflush(stdin);
                printf("\nDo you want to continue: (y: 1/n: 0)?:");
                scanf("%d",&answer);
                i++;
        }while(answer!=0);

        return i;
}

void make_frames(int num_words){
        int i=0;
        printf("\nThe Transmitted Data is:\n\t");

        for(;i<num_words;i++)
                printf("%d%s",(count_chars(i)+1),input[i]);

        printf("\n\n");
}
```

```
int count_chars(int index)
{
        int i=0;
        while(input[index][i]!='\0')
                i++;
        return i;
}
```

Output:

Enter the Word:cat
Do you want to continue: (y: 1/n: 0)?:1

Enter the Word:dog
Do you want to continue: (y: 1/n: 0)?:1

Enter the Word:apple
Do you want to continue: (y: 1/n: 0)?:0

The Transmitted Data is:

4cat4dog6apple

//Bit stuffing

```
#include<stdio.h>
#include<string.h>
#define DELIM_BIT_PATTERN "01111110"
#define SNDR_INPUT 0
#define SNDR_OUTPUT 1
#define REC_INPUT 2
#define REC_OUTPUT 3

char data[4][100];

int valid_data(void);
void sender_bit_stuff(void);
void receiver_process_data(void);

int main()
{
        int ans;
        do{
        printf("\nEnter Data from Netwrok Layer in Binary Form:");
        scanf("%s",data[SNDR_INPUT]);
        if(!valid_data())
                continue;
        sender_bit_stuff();
        printf("\nSenders Physical Layer Data:%s\n",data[SNDR_OUTPUT]);
        strcpy(data[REC_INPUT],data[SNDR_OUTPUT]);
        receiver_process_data();
        printf("\nReceiver's Network Layer Data: %s\n",data[REC_OUTPUT]);
        printf("\n\nDo you want to continue?(y: 1/n: 0)");
        scanf("%d",&ans);
        }while(ans!=0);
}


int valid_data(){
        char *p=data[SNDR_INPUT];
        if(*p=='\0'){
                printf("\n***Enter Some DAta***\n");
                return 0;
        }
        while(*p!='\0'){
                if(*p!='1' && *p!='0'){
                printf("** this is not binary data. please Enter 0's and 1's\n");
                  }
```

```
                p++;
        }
return 1;
}


void sender_bit_stuff(void){
        char *src=data[SNDR_INPUT];
        char *dst=data[SNDR_OUTPUT];
        int count=0;
        strcpy(dst,DELIM_BIT_PATTERN);
        dst+=strlen(DELIM_BIT_PATTERN);
        while(*src!='\0')
        {
                if(count==5){
                        *dst='0';
                        dst+=1;
                        count=0;
                }
                if(*src=='1')
                        count++;
                else
                        count=0;
                *dst++=*src++;
        }
        if(*src=='\0' && count==5){
                *dst='0';
                dst+=1;
        }
        strcpy(dst,DELIM_BIT_PATTERN);
        dst+=strlen(DELIM_BIT_PATTERN);
        *dst='\0';
}


void receiver_process_data(void){
        char *src=data[REC_INPUT];
        char *dst=data[REC_OUTPUT];
        char *end;
        int count=0;
        src+=strlen(DELIM_BIT_PATTERN);
        end=data[REC_INPUT]+strlen(data[REC_INPUT])-strlen(DELIM_BIT_PATTERN);
        while(src<=end)
        {
                if(count==5)
                        src+=1;
```

```
                count=0;
            if(*src=='1')
                count++;
            else
                count=0;
            *dst++=*src++;
        }
        *(dst-1)='\0';

return;
}
```

Output:

Enter Data from Netwrok Layer in Binary Form:01111111111110

Senders Physical Layer Data:01111110011111011111011001111110

Receiver's Network Layer Data: 0111110111110110

Do you want to continue?(y: 1/n: 0)0

//Byte stuffing

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

#define FLAG_BYTE "$"
#define ESCAPE_BYTE "#"

void byte_stuff();
char input_buf[100];
char output_buf[100];

main(){
        int ans;
        do{
        input_buf[0]='\0';
        output_buf[0]='\0';
        printf("\nFLAG_BYTE:$,ESC_BYTE=#\n");
        printf("\nEnter th data from Network Layer:");
        scanf("%s",input_buf);
        byte_stuff();
        printf("\nData to the physical Layer:%s",output_buf);
        printf("\nDo you want to continue?(Y: 1/N: 0):");
        scanf("%d",&ans);
        }while(ans!=0);
return 0;
}


void byte_stuff(void){
        int i=0,j=1;
        output_buf[0]='$';
        for(;input_buf[i]!='\0';i++,j++)
        {
                if(input_buf[i]!='$' && input_buf[i]!='#')
                        output_buf[j]=input_buf[i];
                else{
                        output_buf[j++]='#';
                        output_buf[j]=input_buf[i];
                }
        }
        output_buf[j]='$';
        output_buf[j++]='\0';
}
```

Date:…… **SHEET No:……**

Output:

```
FLAG_BYTE:$,ESC_BYTE=#
Enter the data from Network Layer:cat
Data to the physical Layer:$cat
Do you want to continue?(Y: 1/N: 0):1
FLAG_BYTE:$,ESC_BYTE=#
Enter th data from Network Layer:apple
Data to the physical Layer:$apple
Do you want to continue?(Y: 1/N: 0):0
```

Result:

Thus the program for bit stuffing and character stuffing is executed

Date:…… **SHEET No:……**

EXPERIMENT-2

Aim:

Implement the error correcting code Cyclic Redundancy Check (CRC) of data link layer using various polynomials like CRC-CRC 12, CRC 16 and CRC CCIPP.

Description:

The Data Link Layer is the second layer in the OSI model, above the Physical Layer, which ensures that the error free data is transferred between the adjacent nodes in the network. It breaks the datagram passed down by above layers and converts them into frames ready for transfer. This is called Framing. It provides two main functionalities

- Reliable data transfer service between two peer network layers
- Flow Control mechanism, which regulates the flow of frames such that data congestion is not there at slow receivers due to fast senders.

There are two basic strategies for dealing with errors. One way is to include enough redundant information (extra bits are introduced into the data stream at the transmitter on a regular and logical basis) along with each block of data sent to enable the receiver to deduce what the transmitted character must have been. The other way is to include only enough redundancy to allow the receiver to deduce that error has occurred, but not which error has occurred and the receiver asks for a retransmission. The former strategy uses Error-Correcting Codes and latter uses Error-detecting Codes.

CRC (Cyclic Redundancy Check)

This Cyclic Redundancy Check is the most powerful and easy to implement technique. Unlike checksum scheme, which is based on addition, CRC is based on binary division. In CRC, a sequence of redundant bits, called cyclic redundancy check bits, are appended to the end of data unit so that the resulting data unit becomes exactly divisible by a second, predetermined binary number. At the destination, the incoming data unit is divided by the same number. If at this step there is no remainder, the data unit is assumed to be correct and is therefore accepted. A remainder indicates that the data unit has been damaged in transit and therefore must be rejected.

1. Bit strings are created as representation of polynomials with coefficients ‘0’ and ‘1’only.
2. A k-bit frame is regarded as coefficients list for a polynomial with ‘k’ terms (x^{k-1} to x^0)

   Eg: $x^5 + x^4 + x^0 = 110001$

When this method is used, the sender and the receiver should agree upon a generator polynomial,

G(x) in advance.

Both the high and low order bits of G(x) must be ‘1’

To compute checksum for some frame with 'm' bits (polynomial = M(x), append 'r' zero bits to the lower end of the frame (r = degree of the generator polynomial) so that this check summed frame is divisible by G(x).

Divide M(x) by G(x) using modulo-2 division and subtract the remainder from M(x) using modulo-2subtraction. let the resultant be called as T(x)

T(x) is passed to the receiver and the receiver divides it by G(x).

If there is a remainder, there has been a transmission error.

Algorithm for computing checksum:

1. Let 'r' be the degree of G(x). Append 'r' to the lower end of the frame so that it contains (m + r) bits.
2. Divide M(x) by G(x) using MOD-2 division.
3. Subtract the remainder from M(x) using MOD-2 subtraction.
4. The result is the check summed frame to be transmitted.

Eg: frame = 1101011011

G(x) = $x^4 + x + 1$ = 10011

è degree = 4

Therefore, frame = 1101011011 + 0000

èM(x) = 11010110110000

Commonly used divisor polynomials are:

CRC 12 : $x^{12} + x^{11} + x^3 + x^2 + x + 1$

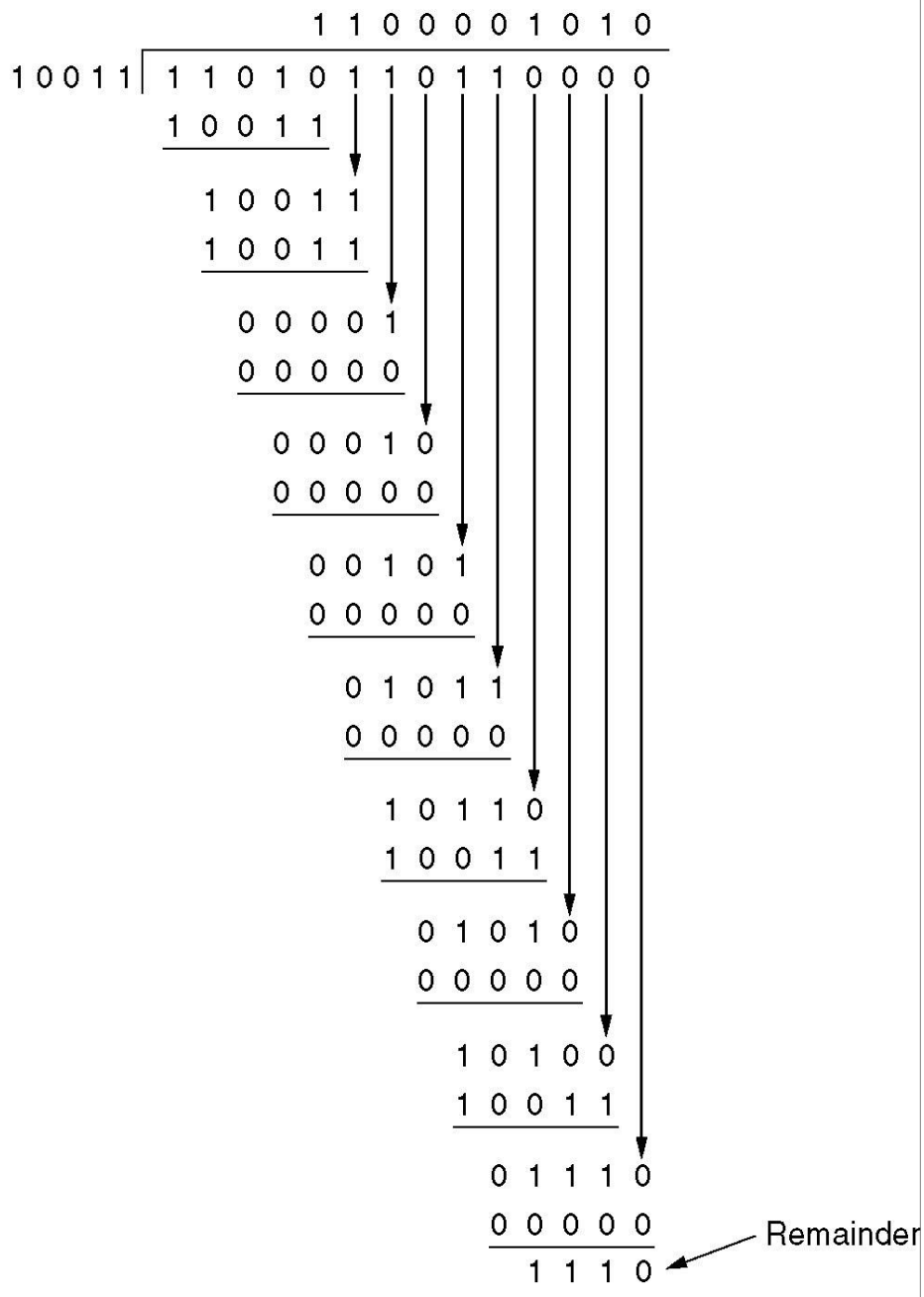
CRC 16 : $x^{16} + x^{15} + x^2 + 1$

CRC CCITT : $x^{16} + x^{12} + x^5 + 1$

Frame : 1 1 0 1 0 1 1 0 1 1

Generator: 1 0 0 1 1

Message after 4 zero bits are appended: 1 1 0 1 0 1 1 0 1 1 0 0 0 0



Transmitted frame: 1 1 0 1 0 1 1 0 1 1 1 1 0

Date:…… **SHEET No:……**

Performance:

CRC is a very effective error detection technique. If the divisor is chosen according to the previously mentioned rules, its performance can be summarized as follows:

- CRC can detect all single-bit errors
- CRC can detect all double-bit errors (three 1's)
- CRC can detect any odd number of errors (X+1)
- CRC can detect all burst errors of less than the degree of the polynomial.
- CRC detects most of the larger burst errors with a high probability.
- For example CRC-12 detects 99.97% of errors with a length 12 or more.

Program:

```
#include <stdio.h>
#include <string.h>
#define N strlen(g)
char t[28],cs[28],g[28];
int a,e,c,b;
void xor()
{
        for(c=1;c<N;c++)
                cs[c]=((cs[c]==g[c])?'0':'1');
}
void crc()
{
        for(e=0;e<N;e++)
                cs[e]=t[e];
        do
        {
                if(cs[0]=='1')
                        xor();
                for(c=0;c<N-1;c++)
                        cs[c]=cs[c+1];
                        cs[c]=t[e++];
        }while(e<=a+N-1);
}
```

```
int main()
{
int flag=0;
do{
        printf("\n1.crc12\n2.crc16\ncrc ccit\n4.exit\n\nEnter your option.");
        scanf("%d",&b);
        switch(b)
        {
                case 1:strcpy(g,"1100000001111");
                break;
                case 2:strcpy(g,"11000000000000101");
                break;
                case 3:strcpy(g,"10001000000100001");
                break;
                case 4:return 0;
        }
printf("\n enter data:");
scanf("%s",t);
printf("\n----------------------\n");
printf("\n generating polynomial:%s",g);
a=strlen(t);
for(e=a;e<a+N-1;e++)
        t[e]='0';
printf("\n-------------------------\n");
printf("mod-ified data is:%s",t);
printf("\n----------------------\n");
crc();
printf("checksum is:%s",cs);
for(e=a;e<a+N-1;e++)
        t[e]=cs[e-a];
printf("\n----------------------\n");
printf("\n final codeword is : %s",t);
printf("\n----------------------\n");
printf("\ntest error detection 0(yes) 1(no)?:");
scanf("%d",&e);
if(e==0)
{
        do{
                printf("\n\tenter the position where error is to be inserted:");
                scanf("%d",&e);
        }
        while(e==0||e>a+N-1);
        t[e-1]=(t[e-1]=='0')?'1':'0';
        printf("\n----------------------\n");
        printf("\n\terroneous data:%s\n",t);
}
```

```
crc();
for(e=0;(e<N-1)&&(cs[e]!='1');e++);
if(e<N-1)
        printf("error detected\n\n");
else
        printf("\n no error detected \n\n");
printf("\n----------------------");

}while(flag!=1);
}
```

Output:

```
1.crc12
2.crc16
3.crc ccit
4.exit

Enter your option.1

enter data:1100110011100011
-----------------------
generating polynomial:1100000001111
------------------------
mod-ified data is:1100110011100011000000000000011100000001111
-----------------------
checksum is:1101110110001
-----------------------
final codeword is : 1100110011100011110111011000111000000001111
------------------------
test error detection 0(yes) 1(no)?:1
no error detected
-----------------------

1.crc12
2.crc16
3.crc ccit
4.exit
Enter your option.2
enter data:11001100111000
-----------------------

generating polynomial:11000000000000101
------------------------
mod-ified data is:1100110011100000000000000000000000000000000101
-----------------------
```

```
checksum is:11111111110110000
----------------------
final codeword is : 1100110011100011111111110110000000000000000101
-----------------------
test error detection 0(yes) 1(no)?:1
no error detected
----------------------
1.crc12
2.crc16
3.crc ccit
4.exit
Enter your option.3
enter data:11001100111000
----------------------
generating polynomial:10001000000100001
-------------------------
mod-ified data is:1100110011100000000000000000000001000000100001
----------------------
checksum is:11100111100111010
----------------------
final codeword is : 1100110011100011100111100111010010000000100001
-----------------------
test error detection 0(yes) 1(no)?:0

Enter the position where error is to be inserted:3
----------------------
erroneous data:1110110011100011100111100111010010000000100001
error detected


----------------------
1.crc12
2.crc16
3.crc ccit
4.exit
Enter your option.4
```

Result:

Thus the program for cyclic redundancy check is executed.

Date:…… **SHEET No:……**

EXPERIMENT-3

Aim:

Implement Dijkstra ‘s algorithm to compute the Shortest path through a graph.

Description:

It is a static routing algorithm. It is used to build a graph of the subnet, with each node of graph representing a router and each arc of the graph representing a communication line. To choose a route between a given pair of routers, the algorithm just finds the shortest path between them on the graph. Different ways of measuring the path length is the number of Hops, Geographical distance in kmts, Mean Queuing delay, Transmission delay, Functions of distance, Bandwidth, Average traffic, communication cost etc.,

Several algorithms for computing the shortest path between two nodes of a graph are known. This one is due to Dijkstra (1959). Each node is labeled (in parentheses) with its distance from the source node along the best known path. Initially, no paths are known, so all nodes are labeled with infinity. As the algorithm proceeds and paths are found, the labels may change, reflecting better paths. A label may be either tentative or permanent. Initially, all labels are tentative. When it is discovered that a label represents the shortest possible path from the source to that node, it is made permanent and never changed thereafter.

To illustrate how the labeling algorithm works, look at the weighted, undirected graph of Fig. (a), where the weights represent, for example, distance. We want to find the shortest path from A to D. We start out by marking node A as permanent, indicated by a filled-in circle. Then we examine, in turn, each of the nodes adjacent to A (the working node), relabeling each one with the distance to A. Whenever a node is relabeled, we also label it with the node from which the probe was made so that we can reconstruct the final path later. Having examined each of the nodes adjacent to A, we examine all the tentatively labeled nodes in the whole graph and make the one with the smallest label permanent, as shown in Fig. (b). This one becomes the new working node.

We now start at B and examine all nodes adjacent to it. If the sum of the label on B and the distance from B to the node being considered is less than the label on that node, we have a shorter path, so the node is relabeled.

After all the nodes adjacent to the working node have been inspected and the tentative labels changed if possible, the entire graph is searched for the tentatively-labeled node with the smallest value. This node is made permanent and becomes the working node for the next round. Figure shows the first five steps of the algorithm.

To see why the algorithm works, look at Fig. (c). At that point we have just made E permanent. Suppose that there were a shorter path than ABE, say AXYZE. There are two possibilities: either node Z has already been made permanent, or it has not been. If it has, then E has already been probed

(on the round following the one when Z was made permanent), so the AXYZE path has not escaped our attention and thus cannot be a shorter path.

Now consider the case where Z is still tentatively labeled. Either the label at Z is greater than or equal to that at E, in which case AXYZE cannot be a shorter path than ABE, or it is less than that of E, in which case Z and not E will become permanent first, allowing E to be probed from Z.

This algorithm is given in Fig. 5-8. The global variables n and dist describe the graph and are initialized before shortest_path is called. The only difference between the program and the algorithm described above is that in Fig. 5-8, we compute the shortest path starting at the terminal node, t, rather than at the source node, s. Since the shortest path from t to s in an undirected graph is the same as the shortest path from s to t, it does not matter at which end we begin (unless there are several shortest paths, in which case reversing the search might discover a different one). The reason for searching backward is that each node is labeled with its predecessor rather than its successor. When the final path is copied into the output variable, path, the path is thus reversed. By reversing the search, the two effects cancel, and the answer is produced in the correct order.

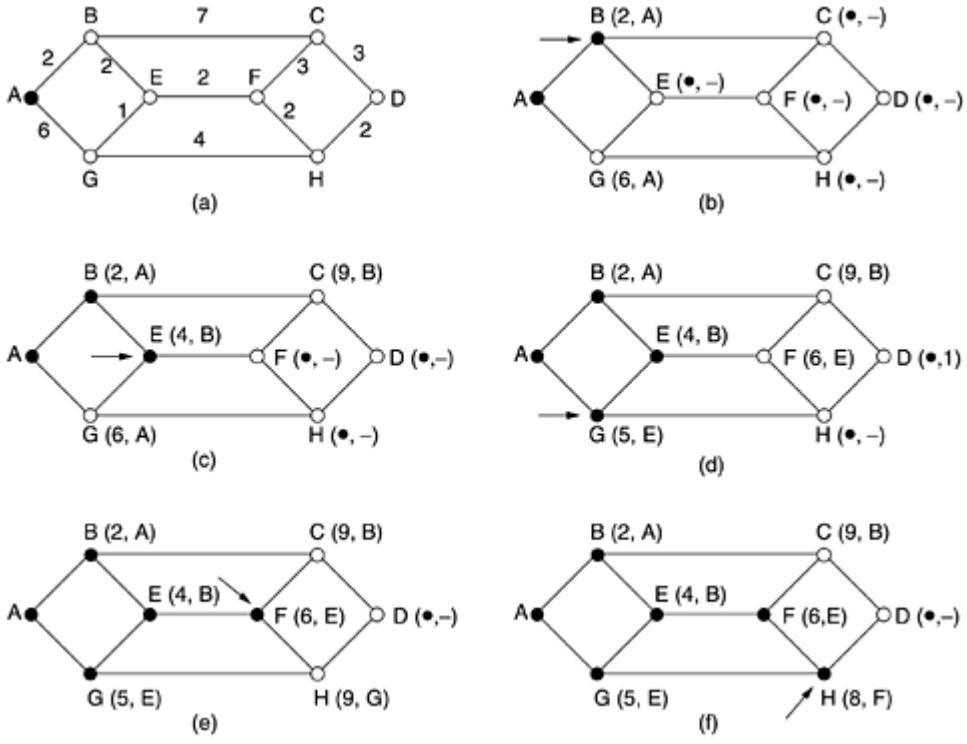


Figure: The first five steps used in computing the shortest path from A to D. The arrows indicate the working node

Finally, Destination 'D' is relabeled as D(10,H). The path is **(D-H-F-E-B-A)** as follows:

D(10,H) = H(8,F)

= F(6,E)

= E(4,B)

= B(2,A)

= A

Program:

```
#include<stdio.h>
#include<stdlib.h>
#define MAX_NODES 1024
#define INFINITY 1000

int n=8,cost=0,dist[8][8]={{0,2,0,0,0,0,6,0},
                           {2,0,7,0,2,0,0,0},
                           {0,7,0,3,0,3,0,0},
                      {0,0,3,0,0,0,0,2},
                      {0,2,0,0,0,2,1,0},
                      {0,0,3,0,2,0,0,2},
                      {6,0,0,0,1,0,0,4},
                      {0,0,0,2,0,2,4,0}};

int shortest_dist(int s,int t,int path[])
{
        int i,k,min;
        struct state
        {
                int pre;
                int length;
                int label;
        }state[1024];

        struct state *p;

        for(p=&state[0];p<&state[n];p++)
        {
                p->pre=-1;
                p->length=INFINITY;
                p->label=0;
        }
```

```
        state[0].length=0;
        state[0].label=1;

        state[0].pre=-1;
        k=t;

do
        {
        for(i=0;i<n;i++)
                if(dist[k][i]!=0 && state[i].label==0)
                {
                        if(state[k].length+dist[k][i]<state[i].length)
                        {
                                state[i].pre=k;
                                state[i].length=state[k].length+dist[k][i];
                        }
                }

        k=0;
        min=INFINITY;

        for(i=0;i<n;i++)

                if(state[i].label==0 && state[i].length<min)
                {
                        min=state[i].length;
                        k=i;
                }
                state[k].label=1;

        }while(k!=s);
        i=0;
        k=s;
        do
        {

                path[i++]=k;
                k=state[k].pre;
                cost+=state[k].length;

        }while(k>=0);
return i;
}

void main()
```

```
{
        int i,j,m,path[102],q,p;

        printf("\nEnter Number of nodes(1-8): ");
        scanf("%d",&n);

        printf("\nEnter Source Vertex(1-8): ");
        scanf("%d",&p);

        printf("\nEnter Destination vertex(1-8): ");
        scanf("%d",&q);

        m=shortest_dist(q-1,p-1,path);
        for(i=0;i<m;i++)
        {
                printf(" %c-> ",path[i]+'A');


        }

        printf("\nCost is:  %d \n",cost);
}
```

Output:

```
Enter Number of nodes(1-8): 7

Enter Source Vertex(1-8): 1

Enter Destination vertex(1-8): 7
 G->  E->  B->  A->
Cost is:  6
```

Result:

Thus the program Implement Dijkstra ‘s algorithm to compute the Shortest path through a graph is executed

Date:…… **SHEET No:……**

Aim:

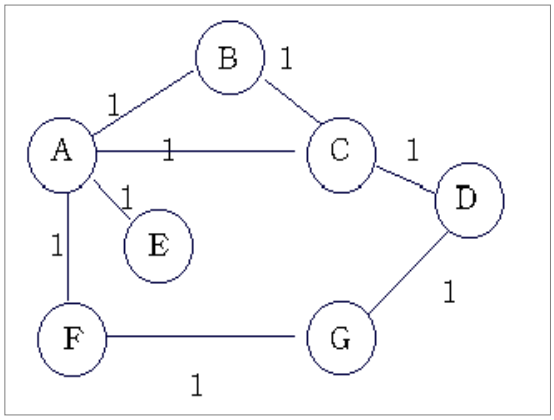
Take an example subnet graph with weights indicating delay between nodes. Now obtain Routing table art each node using distance vector routing algorithm

Description:

Each node constructs a one-dimensional array containing the "distances"(costs) to all other nodes and distributes that vector to its immediate neighbors.

The starting assumption for distance-vector routing is that each node knows the cost of the link to each of its directly connected neighbors.
A link that is down is assigned an infinite cost.

Example.



Information Stored at Node	**Distance to Reach Node**						
	A	**B**	**C**	**D**	**E**	**F**	**G**
A	0	1	1	?	1	1	?
B	1	0	1	? ?		?	?

C	1	1	0	1	?	?	?
D	?	?	1	0	?	?	1
E	1	?	?	?	0	?	?
F	1	?	?	?	?	0	1
G	?	?	?	1	?	1	0

Table 1. Initial distances stored at each node(global view).

We can represent each node's knowledge about the distances to all other nodes as a table like the one given in Table 1.

Note that each node only knows the information in one row of the table.

3. Every node sends a message to its directly connected neighbors containing its personal list of distance. (for example, A sends its information to its neighbors B,C,E, and F.)
4. If any of the recipients of the information from A find that A is advertising a path shorter than the one they currently know about, they update their list to give the new path length and note that they should send packets for that destination through A. (node B learns from A that node E can be reached at a cost of 1; B also knows it can reach A at a cost of 1, so it adds these to get the cost of reaching E by means of A. B records that it can reach E at a cost of 2 by going through A.)
5. After every node has exchanged a few updates with its directly connected neighbors, all nodes will know the least-cost path to all the other nodes.
6. In addition to updating their list of distances when they receive updates, the nodes need to keep track of which node told them about the path that they used to calculate the cost, so that they can create their forwarding table. (for example, B knows that it was A who said " I can reach E in one hop" and so B puts an entry in its table that says " To reach E, use the link to A.)

Information Stored at Node	**Distance to Reach Node**						
	A	**B**	**C**	**D**	**E**	**F**	**G**
A	0	1	1	2	1	1	2
B	1	0	1	2	2	2	3
C	1	1	0	1	2	2	2
D	□	2	1	0	3	2	1
E	1	2	2	3	0	2	3
F	1	2	2	2	2	0	1
G	□	3	2	1	3	1	0

Table 2. final distances stored at each node (global view).

In practice, each node's forwarding table consists of a set of triples of the form:

(Destination, Cost, NextHop).

For example, Table 3 shows the complete routing table maintained at node B for the network in figure1.

Destination	**Cost**	**NextHop**
A	1	A
C	1	C

D	2	C
E	2	A
F	2	A
G	3	A

Table 3. Routing table maintained at node B.

Program:

```
#include<stdio.h>

struct node
{
unsigned dist[20];
unsigned from[20];
}rt[10];


int main()
{
int dmat[20][20];
int n,i,j,k,count=0;
printf("enter the number of nodes:");
scanf("%d",&n);
printf("enter the cost matrix :\n");

for(i=0;i<n;i++)
for(j=0;j<n;j++)
{
scanf("%d",&dmat[i][j]);
dmat[i][i]=0;
rt[i].dist[j]=dmat[i][j];
rt[i].from[j]=j;
}

do
{
count=0;
for(i=0;i<n;i++)
```

```
for(j=0;j<n;j++)
for(k=0;k<n;k++)
if(rt[i].dist[j]>dmat[i][k]+rt[k].dist[j])
{
rt[i].dist[j]=rt[i].dist[k]+rt[k].dist[j];
rt[i].from[j]=k;
count++;
}

}while(count!=0);

for(i=0;i<n;i++)
for(j=0;j<n;j++)
{
printf("\n state value for router %d is\n",i+1);
for(j=0;j<n;j++)
{
printf("\n node %d via %d Distance %d", j+1,rt[i].from[j]+1,rt[i].dist[j]);
}
}
printf("\n");
}
```

Output:

```
enter the number of nodes:3
enter the root matrix:
0 2 4
2 0 5
4 5 0
state value for router 1 is

node 1 via 1 distance 0
node 2 via 2 distance 2
node 3 via 3 distance 4

state value for router 2 is

node 1 via 1 distance 2
node 2 via 2 distance 0
node 3 via 3 distance 5

state value for router 3 is

node 1 via 1 distance 4
node 2 via 2 distance 5
```

node 3 via 3 distance 0

Result:

Thus the program for obtain Routing table art each node using distance vector routing algorithm is executed.

Date:…… **SHEET No:……**

EXPERIMENT-5

Aim:

Take an example subnet of hosts. Obtain broadcast tree for it.

Description:

Sending a packet to all destinations simultaneously is called broadcasting. The set of optimal routes from a source to all destinations form a tree rooted at the source. Such a tree is called a sink tree. It is illustrated in Fig., where the distance metric is the number of hops. Note that a sink tree is not necessarily unique; other trees with the same path lengths may exist. The goal of all routing algorithms is to discover and use the sink trees for all routers

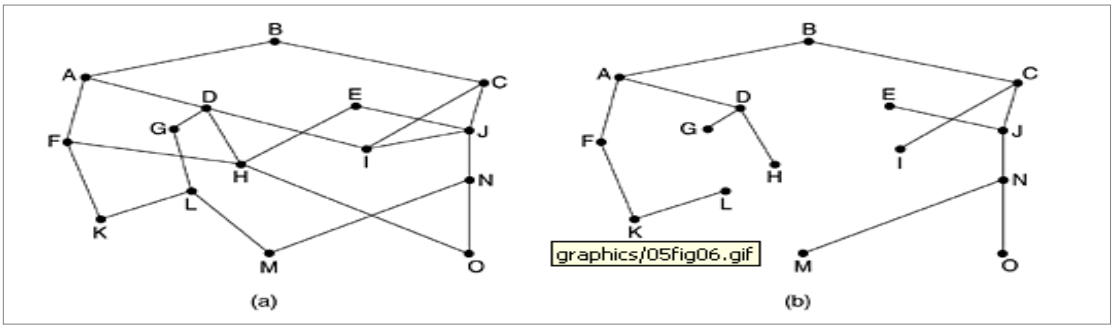


Figure: a) subnet b) sink tree for router B

A broadcast algorithm makes explicit use of the sink tree for the router initiating the broadcast. A sink tree is a subset of the subnet that includes all the routers but contains no loops. If each router knows which of its lines belong to the sink tree, it can copy an incoming broadcast packet onto all the sink tree lines except the one it arrived on. This method makes excellent use of bandwidth, generating the absolute minimum number of packets necessary to do the job. The only problem is that each router must have knowledge of some sink tree for the method to be applicable.

Program:

```
#include<stdio.h>
#include<stdlib.h>
#define MAX_NODES 20
#define INFINITY 1000

void run_prim(int i,int num_nodes,int *hops);
int dist[MAX_NODES][MAX_NODES];
int path[MAX_NODES][MAX_NODES];
int main()
{
        int num_nodes=0;
        int i=0,j=0;
```

```
        int path_len,hops[MAX_NODES];

        printf("enter the number of nodes:");
        scanf("%d",&num_nodes);
        fflush(stdin);
        printf("enter the connection matrix, 0 if not connected\n");
        printf("otherwise,the distance\n");
        for(;i<num_nodes;i++)
        {
                printf("enter the distances for node num:%d\n",i);
                for(j=0;j<num_nodes;j++)
                {
                        if(i==j)
                        {
                                dist[i][j]=0;
                                continue;
                        }
                        printf("distance from %d to %d= ",i,j);
                        scanf("%d",&dist[i][j]);
                        fflush(stdin);
                        if(dist[i][j]==0)
                                dist[i][j]=INFINITY;
                }
        }

        printf("\nenter the root node");
        scanf("%d",&i);
        run_prim(i,num_nodes,hops);
        for(j=0;j<num_nodes;j++)
        {
                for(i=0;i<hops[j];i++)
                        printf("->%d",path[j][i]);
                printf("\n");
        }
        return 0;
}
void run_prim(int s,int n,int *hops)
{
        struct state
        {
        int prev;
                int length;
                enum {perm,tent} label;
        }state[MAX_NODES];
        int i,j,k,min;
        int count=0;
```

```
        struct state *p;
        for(p=&state[0];p<&state[n];p++)
        {
                p->prev= -1;
                p->length= INFINITY;
                p->label= tent;
        }
        state[s].length= 0;state[s].label= perm;
        k=s;
        do
        {
        for(i=0;i<n;i++)
                        if(dist[k][i]!=0&&state[i].label==tent)
                        {
                                if(state[k].length+dist[k][i]<state[i].length)
                                {
                                        state[i].prev=k;
                                        state[i].length=state[k].length+dist[k][i];
                                }
                        }
k=0;
                min=INFINITY;
                for(i=0;i<n;i++)
                        if(state[i].label==tent&&state[i].length<min)
                        {
                                min=state[i].length;
                                k=i;
                        }
                state[k].label=perm;
                count++;
        }while(count<n);

        for(j=0;j<n;j++)
        {
                i=0;k=j;
                do
                {
                        path[j][i++]=k;
                        k=state[k].prev;
                }

                while(k>=0);
                hops[j]=i;
        }
}
```

Date:…… **SHEET No:……**

Output:

```
enter the number of nodes:4
enter the connection matrix, 0 if not connected otherwise,the distance
enter the distances for node num:0
distance from 0 to 1= 2
distance from 0 to 2= 1
distance from 0 to 3= 6
enter the distances for node num:1
distance from 1 to 0= 5
distance from 1 to 2= 1
distance from 1 to 3= 3
enter the distances for node num:2
distance from 2 to 0= 5
distance from 2 to 1= 1
distance from 2 to 3= 3
enter the distances for node num:3
distance from 3 to 0= 5
distance from 3 to 1= 4
distance from 3 to 2= 6
enter the root node2

->0->2
->1->2
->2
->3->2
```

Result:

Thus the program to obtain broadcast tree for an example subnet of hosts is executed.