10.1 Implement priority queues as a linked list.

10.2 Demonstrate the working of an ordinary queue, implementing it as a linked list.

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

// Queue Element structure
struct QElement
{
   int data;
   struct QElement *nxtelem;
};

typedef struct QElement QElem;

// Class Implementation of Queue
class Q
{
   // Private members
   private:
          int count;
          QElem *front,*rear;

   //Public members
   public:
          Q();
          int qFull();
          int qEmpty();
          int qFront(int &);
          int qRear(int &);
          int qCount();
          int enqueue(int);
          int dequeue(int &);
          int qDestroy();
          void qPrint();
};

Q :: Q()
{
   count=0;
   front=NULL;
   rear=NULL;
}

int Q :: qDestroy()
{
   int val,flag;
   while(front)
   {
          flag=dequeue(val);
   }
   return flag;
```

```
}

int Q :: qFull()
{
   QElem * qtmpelem=new QElem;
   if(qtmpelem)
   {
          delete qtmpelem;
          return 0;
   }
   else
          return 1;
}

int Q :: qEmpty()
{
   if(!front)
          return 1;
   else
          return 0;
}

int Q :: qFront(int &dataout)
{
   if(qEmpty())
          return 0;
   dataout=front->data;
   return 1;
}

int Q :: qRear(int &dataout)
{
   if(qEmpty())
          return 0;
   dataout=rear->data;
   return 1;
}

int Q :: qCount()
{
   return count;
}

int Q :: enqueue(int datain)
{
   if(qFull())
          return 0;
   QElem *newelem = new QElem;
   newelem->data=datain;
   newelem->nxtelem=NULL;
   if(qEmpty())
          front=newelem;
   else
```

```
            rear->nxtelem=newelem;
    rear=newelem;
    count++;
    return 1;
}

int Q :: dequeue(int &dataout)
{
    if(qEmpty())
            return 0;
    dataout=front->data;
    QElem *temp=front;
    front=front->nxtelem;
    if(!front)
            rear=NULL;
    delete temp;
    count--;
    return 1;

}

void Q :: qPrint()
{
    if(!qEmpty())
    {
            cout<<"\nData: \n";
            cout<<"----\n";
            QElem *temp=front;
            while(temp)
            {
                    cout<<temp->data<<endl;
                    temp=temp->nxtelem;
            }
    }
    else
            cout<<"Queue empty... :(";
}

void main()
{
    clrscr();
    Q q1;
    q1.qPrint();
    q1.enqueue(10);
    q1.enqueue(20);
    q1.enqueue(30);
    q1.qPrint();
      q1.dequeue();
    q1.qPrint();
    q1.qFront();
    q1.qRear();
    q1.qPrint();
    q1.qDestroy();
```

```
   q1.qPrint();
}
```

Q 10.3 Implement circular queue as a linked list.

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

// Queue Element structure
struct QElement
{
   int data;
   struct QElement *nxtelem;
};

typedef struct QElement QElem;

// Class Implementation of Queue
class Q
{
   // Private members
   private:
          int count;
          QElem *rear;

   //Public members
   public:
          Q();
          int qFull();
          int qEmpty();
          int qFront(int &);
          int qRear(int &);
          int qCount();
          int enqueue(int);
          int dequeue(int &);
          int qDestroy();
          void qPrint();
};

Q :: Q()
{
   count=0;
   rear=NULL;
}

/*void Q :: qCreate()
{
   count=0;
   front=rear=NULL;
}*/

int Q :: qDestroy()
{
```

```
    int val,flag;
    //QElem *temp=rear->nxtelem;
    while(rear)
    {
            flag=dequeue(val);
    }
    return flag;
}

int Q :: qFull()
{
    QElem * qtmpelem=new QElem;
    if(qtmpelem)
    {
            delete qtmpelem;
            return 0;
    }
    else
            return 1;
}

int Q :: qEmpty()
{
    if(rear)
            return 0;
    else
            return 1;
}

int Q :: qFront(int &dataout)
{
    if(qEmpty())
            return 0;
    dataout=rear->nxtelem->data;
    return 1;
}

int Q :: qRear(int &dataout)
{
    if(qEmpty())
            return 0;
    dataout=rear->data;
    return 1;
}

int Q :: qCount()
{
    return count;
}

int Q :: enqueue(int datain)
{
    if(qFull())
```

```
        return 0;
    QElem *newelem = new QElem;
    newelem->data=datain;
    newelem->nxtelem=NULL;
    if(qEmpty())
    {
        rear=newelem;
        rear->nxtelem=newelem;
    }
    else
    {
        newelem->nxtelem=rear->nxtelem;
        rear->nxtelem=newelem;
        rear=newelem;

    }
    count++;
    return 1;
}

int Q :: dequeue(int &dataout)
{
    if(qEmpty())
        return 0;
    QElem *temp=rear->nxtelem;
    dataout=temp->data;
    if(rear==temp)
        rear=NULL;
    else
        rear->nxtelem=temp->nxtelem;
    delete temp;
    count--;
    return 1;

}

void Q :: qPrint()
{
    if(!qEmpty())
    {
        cout<<"\nData: \n";
        cout<<"----\n";
        QElem *temp=rear;
        do
        {
            temp=temp->nxtelem;
            cout<<temp->data<<endl;
        }while(temp!=rear);
    }
    else
        cout<<"Queue empty... :(";
}
```

```
void main()
{
   clrscr();
   Q q1;
   q1.qPrint();
   q1.enqueue(10);
   q1.enqueue(20);
   q1.enqueue(30);
   q1.qPrint();
q1.dequeue();
   q1.qPrint();
   q1.qFront();
   q1.qRear();
   q1.qPrint();
   q1.qDestroy();
   q1.qPrint();
}
```

Q 10.3 Write a C++ program to implement a circular queue through an array with all the standard operations.


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

class Q
{
   private:
            int *arr, front, rear, count, max;
   public:
            Q();
            int qCreate();
            int enqueue(int);
            int dequeue();
            int qFront();
            int qRear();
            int qFull();
            int qEmpty();
            int qCount();
            void qDestroy();
            void qPrint();
};

Q :: Q()
{
   front=rear=-1;
   count=0;
   cout<<"\nReading Array Size:\n";
   cout<<"Array Size: ";
   cin>>max;
```

```
}

int Q :: qCreate()
{
   arr=new int[max];
   assert(arr);
   return 1;
   //cout<<"\nCreating Array\n";
}

int Q :: enqueue(int no)
{
   if(qFull())
          return -1;
   rear++;
   if(rear==max)
          rear=0;
   arr[rear]=no;
   if(count==0)
   {
          front=0;
          count=1;
   }
   else
          count++;
   return 1;
}

int Q :: dequeue()
{
   if(qEmpty())
          //Queue empty.
          return -1;
   int dataout=arr[front];
   front++;
   if(front==max)
          //Wrapping queue. Front holds value 0.
          front=0;
   if(count==1)
          rear=front=-1;

          count--;
   return dataout;
}

int Q :: qFront()
{
   if(!count)
          return -1;
   return arr[front];
}

int Q :: qRear()
```

```
{
   if(!count)
          return -1;
   return arr[rear];
}

int Q :: qFull()
{
   return (count==max);
}

int Q :: qEmpty()
{
   return (count==0);
}

int Q :: qCount()
{
   return count;
}

void Q :: qDestroy()
{
   while(count!=0)
   {
          dequeue();
   }
   delete arr;
}

void  Q :: qPrint()
{
   int cnt=count;
   int pos=front;
   while(cnt)
   {
          cout<<arr[pos]<<"\t";
          cnt--;
          pos++;
          if(pos==max)
                 pos=0;
   }
}

void main()
{
   Q q1;
   q1.qCreate();
   cout<<endl<<q1.qFull()<<"\t"<<q1.qEmpty()<<endl;
   q1.enqueue(10);
   cout<<endl<<q1.qCount();
   q1.enqueue(20);
   cout<<endl<<q1.qCount();
```

```
    q1.enqueue(30);
    cout<<endl<<q1.qCount();
    q1.qPrint();
    cout<<endl<<q1.qFull()<<"\t"<<q1.qEmpty();
    cout<<endl<<q1.qFront();
    cout<<endl<<q1.qRear();
    cout<<q1.dequeue();
    cout<<q1.qCount();
    q1.enqueue(40);
    cout<<q1.qCount();
    cout<<endl<<q1.qFront();
    cout<<endl<<q1.qRear();
}
```

Q10.4 Write algorithms to (for a linked list) (University):

a.)createQ

Algorithm createQueue
Creates and initializes queue structure.
    Pre queue is a metadata structure
    Post metadata elements have been initialized
    Return queue head
1. allocate queue head
2. set queue front to null
3. set queue rear to null
4. set queue count to 0
5. return queue head
End createQueue

b.)Enqueue

Algorithm enqueue(queue, dataIn)
This algorithm inserts data into queue.
    Pre queue is a metadata structure
    Post dataIn has been inserted
    Return true if successfull, false if overflow
1. If (queue full)
    1. return false
2. end if
3. allocate (new node)
4. move dataIn to new node data
5. set new node next to null pointer
6. if (empty queue)
    1. set queue front to address of new data
7. else
    1 .set next pointer of rear node to address of new node
8. end if
9. set queue rear to address of new node
10. increment queue count
11. return true
End enqueue

c.)Dequeue

Algorithm dequeue(queue, item)
This algorithm deletes a node from a queue.
    Pre queue is a metadata structure
    Post item is a reference to calling algorithm variable data at queue front returned to user through item and front element deleted
    Return true if successful, false if underflow
1. if(queue empty)
    1. return false
2. end if
3. move front data to item
4. if (only 1 node in queue)
    1. set queue rear to null
5. end if
6. set queue front to queue front next
7. decrement queue count
8. return true
End dequeue

d.)qFront

Algorithm queueFront(queue, dataOut)
Retrieve data at the front of the queue without changing queue contents.
    Pre queue is a metadata structure
         dataOut is a reference to calling algorithm variable
    Post data passed back to caller
    Return true if successful, false if underflow
1. if(queue empty)
    1. return false
2. end if
3. move data at front of queue to dataOut
4. return true
End queueFront

e.)qRear

Algorithm queueRear(queue,dataOut)
Retrieve data at the end of the queue without changing queue contents.
    Pre queue is a metadata structure
         dataOut is a reference to calling algorithm variable
    Post data passed back to caller
    Return true if successful, false if underflow
1.if(queue empty)
    1.return false
2.end if
3.move data at the end of queue to dataOut
4.return true
Emd queueRear

f.)emptyQ

Algorithm emptyQueue(queue)
This algorithm checks to see if a queue is empty.
    Pre queue is a metadata structure
    Return true if empty, false if queue has data
1. if(queue cost equal 0)
    1. return true
2. else
    1. return false
End emptyQueue

g.)fullQ

Algorithm fullQueue(queue)
This algorithm checks to see if a queue is full. The queue is full if memory cannot be allocated for another node.
    Pre queue is a metadata structure
    Return true if full, false if room for another node
1. if(memory not available)
    1. return true
2. else
    1. return false
3. end if
End fullQueue

h.)countQ

Algorithm queueCount(queue)
This algorithm returns the number of elements in the queue.
    Pre queue is a metadata structure
    Return queue count
1. return queue count
End queueCount

i.)Destroy

Algorithm destroyQueue(queue)
This algorithm deletes all data from queue.
    Pre queue is a metadata structure
    Post all data have been deleted
1.if(queue not empty)
    1.loop(queue not empty)
        1.delete front node
    2.end loop
2.end if
3.delete head structure
End destroyQueue

Q 10.5 Explain how Huffman's code helps in data compression.

The American Standard Code for Information Interchange (ASCII) is a fixed – length code; that is, the character length does not vary.

Each ASCII character consists of 7 bits. Although the character E occurs more frequently than the character Z, both are assigned the same number of bits. This consistency means that every character uses the maximum number of bits.

Huffman code, on the other hand, makes character storage more efficient. In Huffman code we assign shorter codes to characters that occur more frequently and longer codes to those that occur less frequently.

For example, E and T, two characters that occur frequently in the English language, could be assigned one bit each. A, O, R, and N, which also occur frequently but less frequently than E and T, could be assigned two bits each. S, U, I, D, M, C, and G are the next most frequent and could be assigned three bits each, and so forth.

In a given piece of text, only some of the characters require the maximum bit length. When used in a network transmission, the overall length of the transmission is shorter if Huffman - encoded characters are transmitted rather than fixed – length encoding; Huffman code is therefore a popular data compression algorithm.

Huffman code is widely used for data compression; it reduces the number of bits sent or stored.

Q 10.6 Solve the following university questions:
M2012-Q1(b)

Given the set of symbols and corresponding frequency table as below, explain the steps to find the Huffman code.

Symbol	E	T	A	O	R	N	S	U	I	D	M	C	G	K
Frequency	15	12	10	8	7	6	5	5	4	4	3	3	2	2

Steps to find Huffman code is as follows:

1. Arrange the symbol in descending order of their frequency.
2. Now find the two nodes with the smallest frequency weights and join them to form a third node. The weight of the third node is the combined weights of the original two nodes. This new node is eligible to be combined with other nodes.
3. Repeat step 1 until all of the nodes on every level are combined into a single tree.
4. Assigning 1 to right child of a node and 0 to left child of a node.
5. Starting from root node till the leaf node, noting the generated Huffman code for each symbol.

STEP 1:-

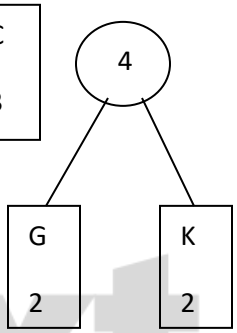
Arranging symbols in descending order of frequency.

E	T	A	O	R	N	S	U	I	D	M	C	G	K
15	12	10	8	7	6	5	5	4	4	3	3	2	2

STEP 2:-

Combining smallest frequency node G and K and forming new node.

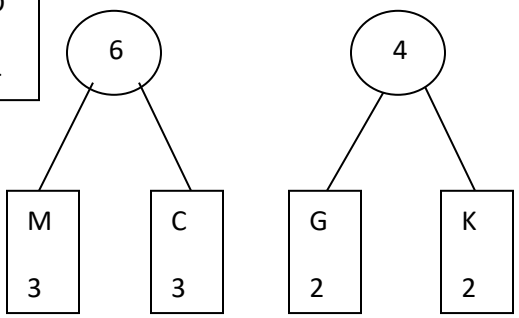
E	T	A	O	R	N	S	U	I	D	M	C
15	12	10	8	7	6	5	5	4	4	3	3



STEP 3:-

Combining smallest frequency node M and C and forming new node.

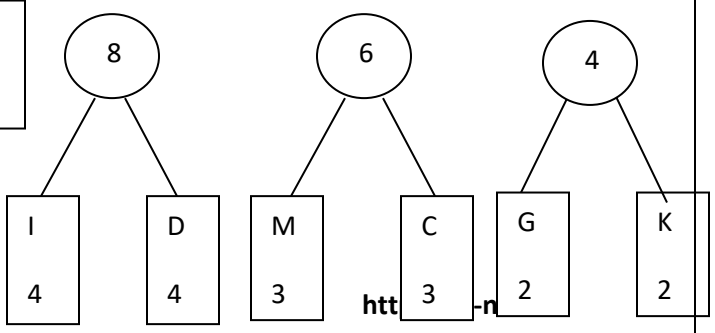
E	T	A	O	R	N	S	U	I	D
15	12	10	8	7	6	5	5	4	4



STEP 4:-

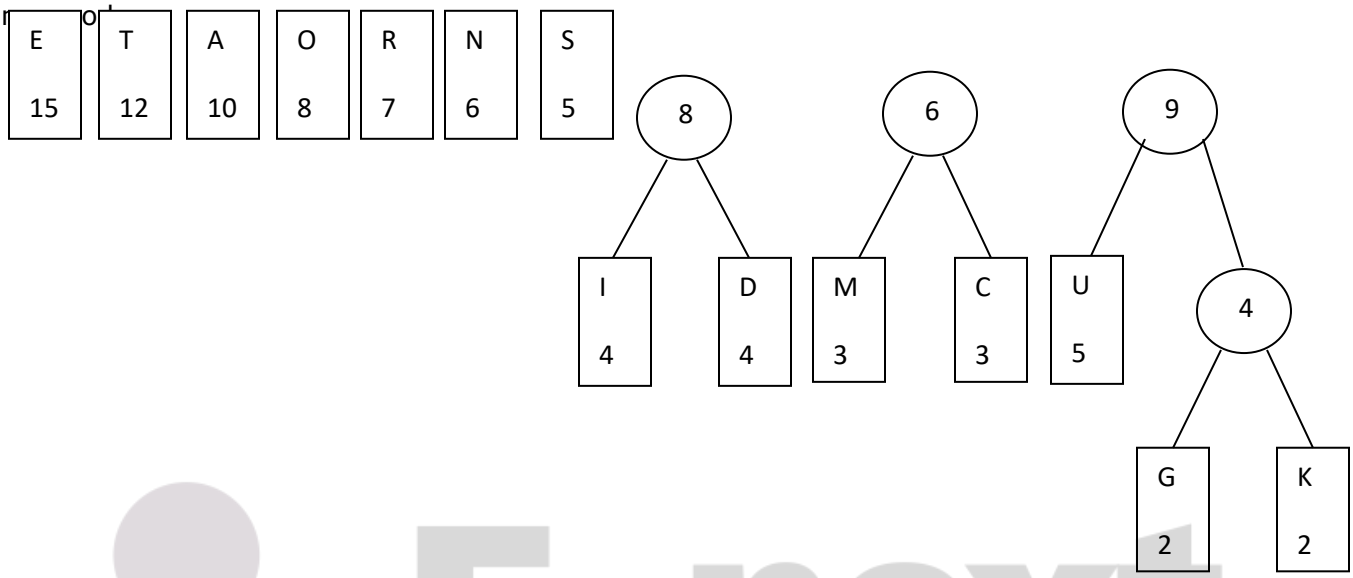
Combining smallest frequency node I and D and forming new node.

E	T	A	O	R	N	S	U
15	12	10	8	7	6	5	5



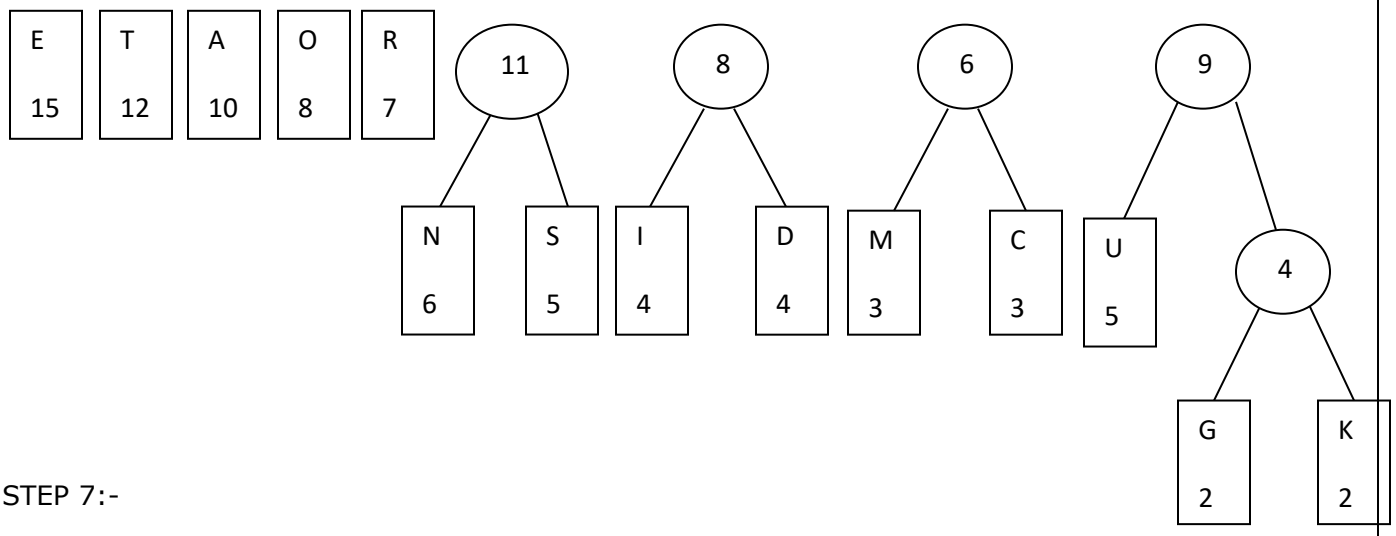
STEP 5:-

Combining smallest frequency node U and newly formed node of frequency 4 and forming new node.



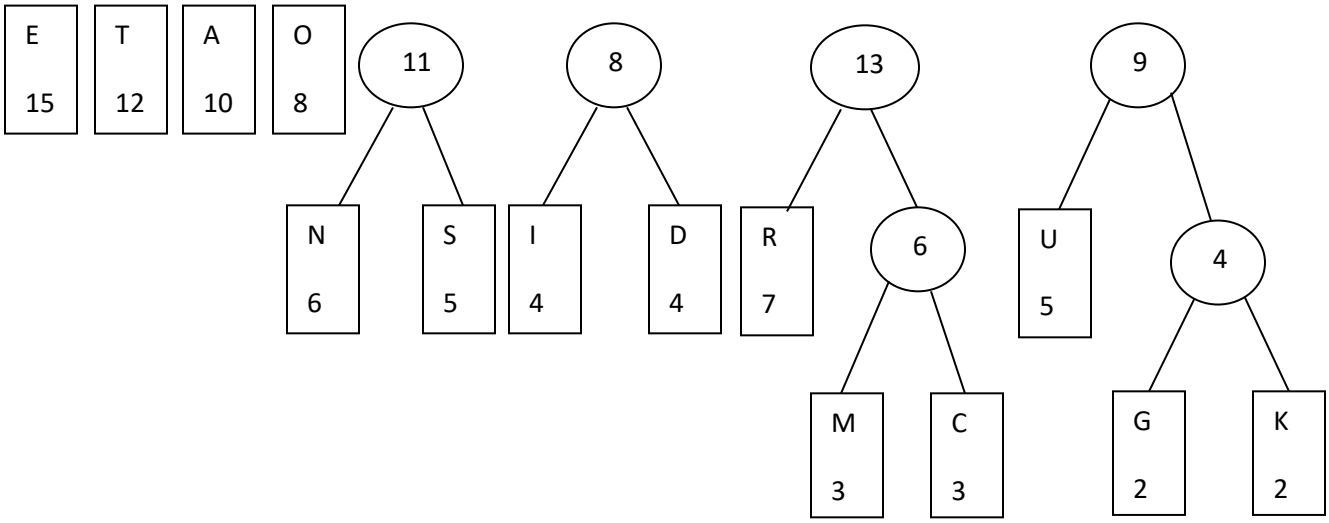
STEP 6:-

Combining smallest frequency node N and S and forming new node.



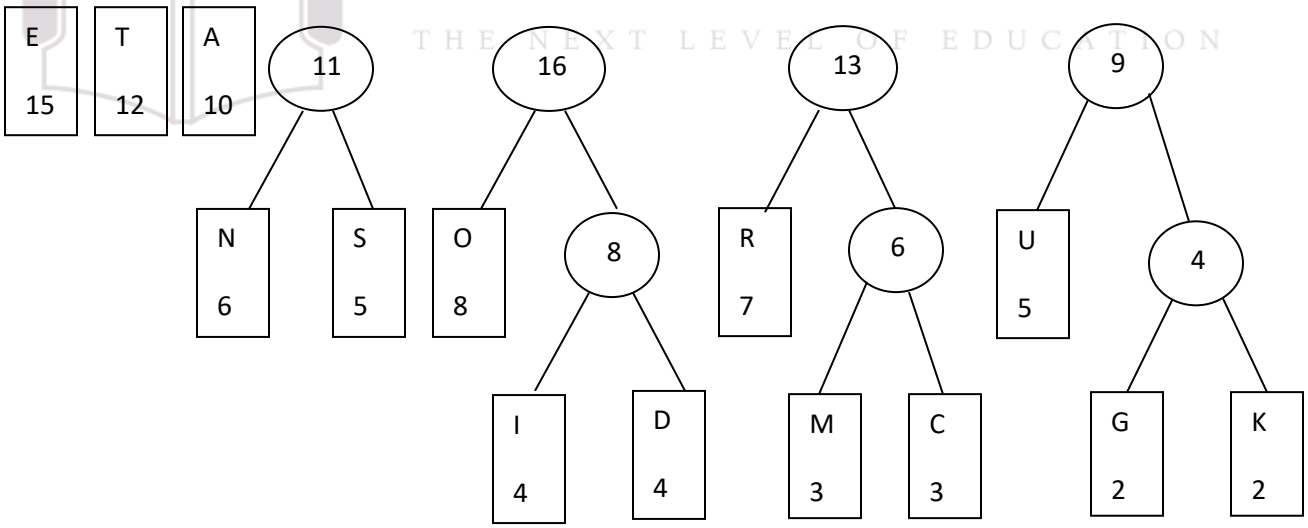
STEP 7:-

Combining smallest frequency node R and newly formed node of frequency 6 and forming new node.



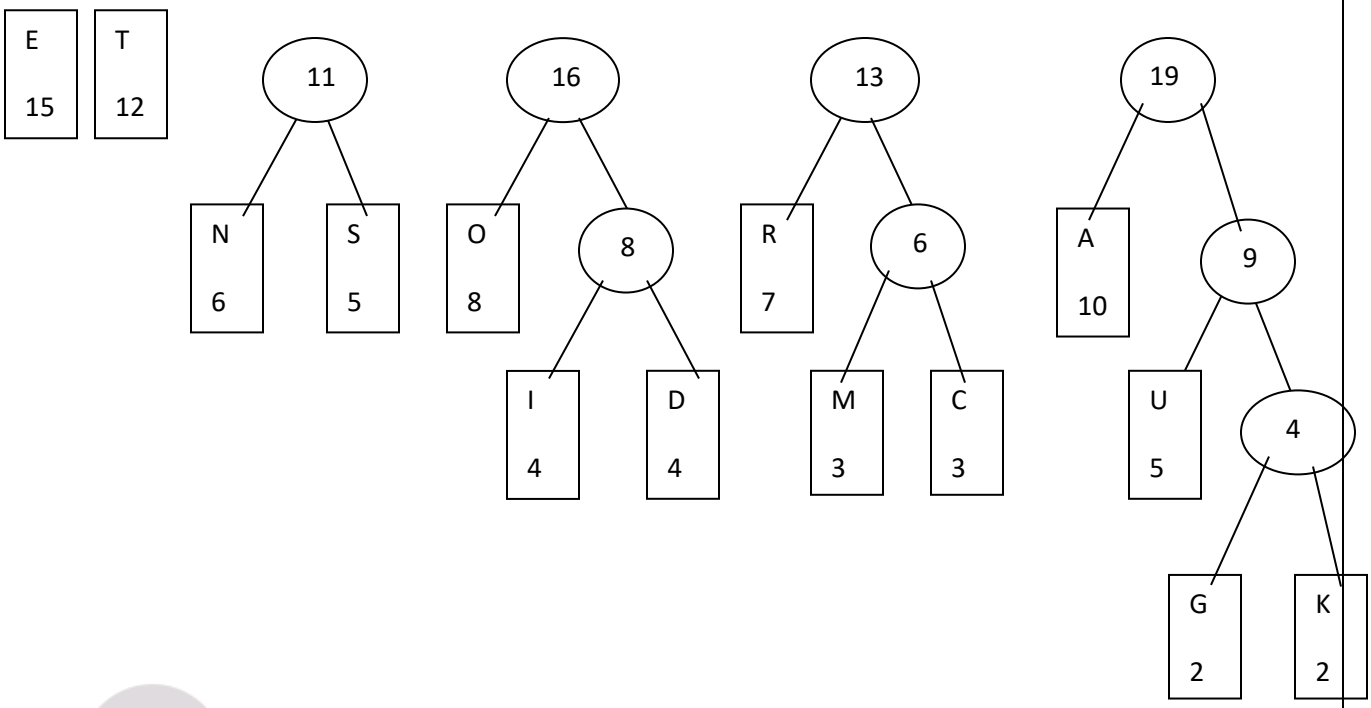
STEP 8:-

Combining smallest frequency node O and newly formed node of frequency 8 and forming new node.



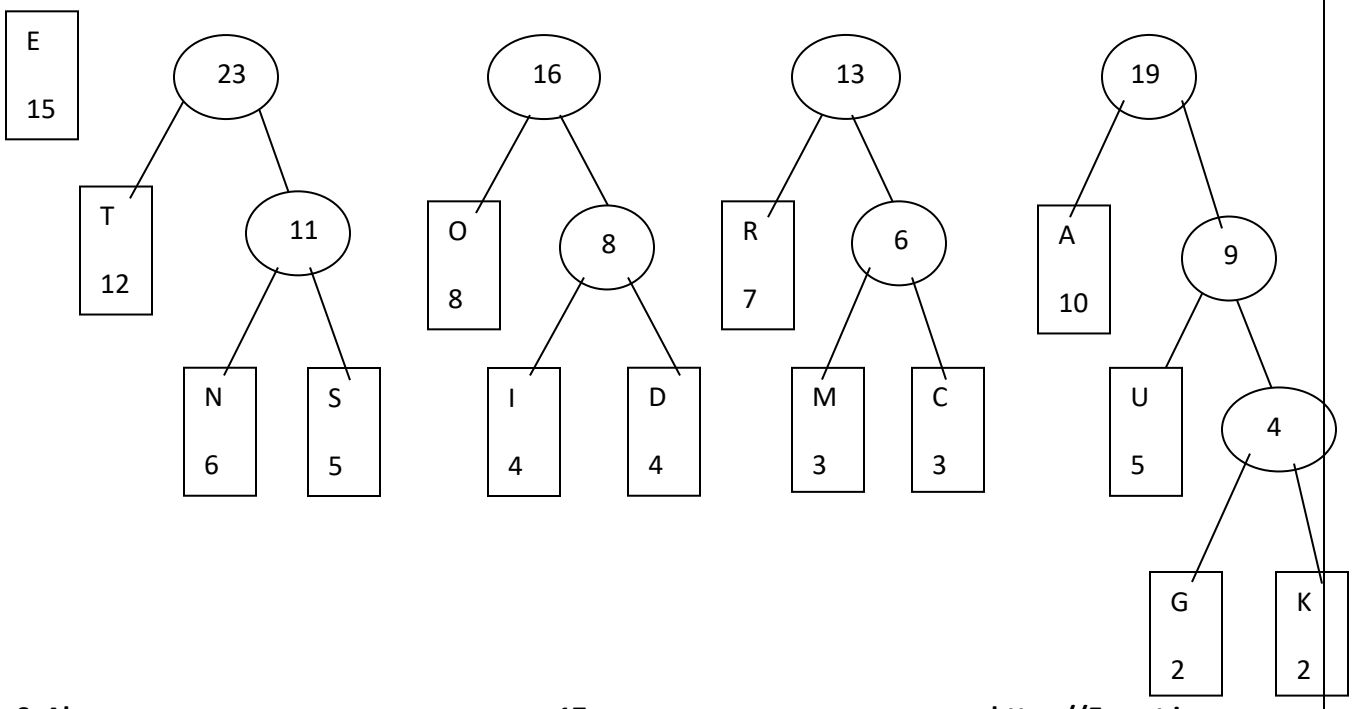
STEP 9:-

Combining smallest frequency node A and newly formed node of frequency 9 and forming new node.



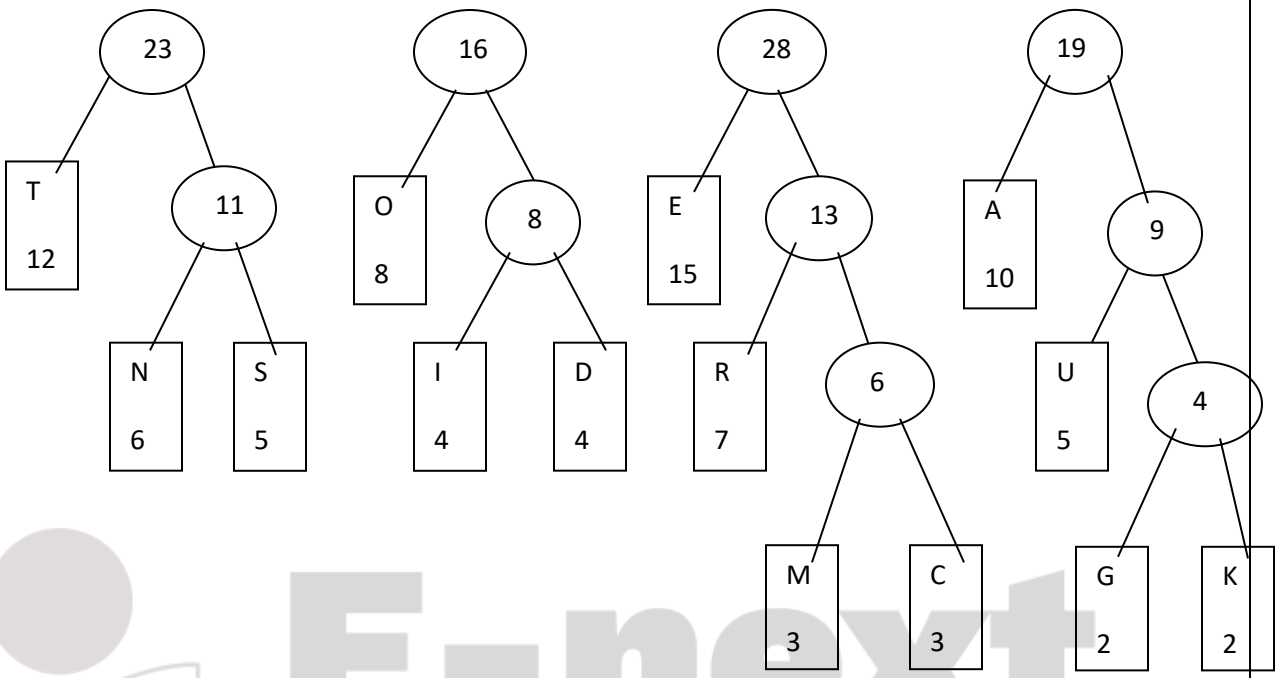
STEP 10:-

Combining smallest frequency node T and newly formed node of frequency 11 and forming new node.

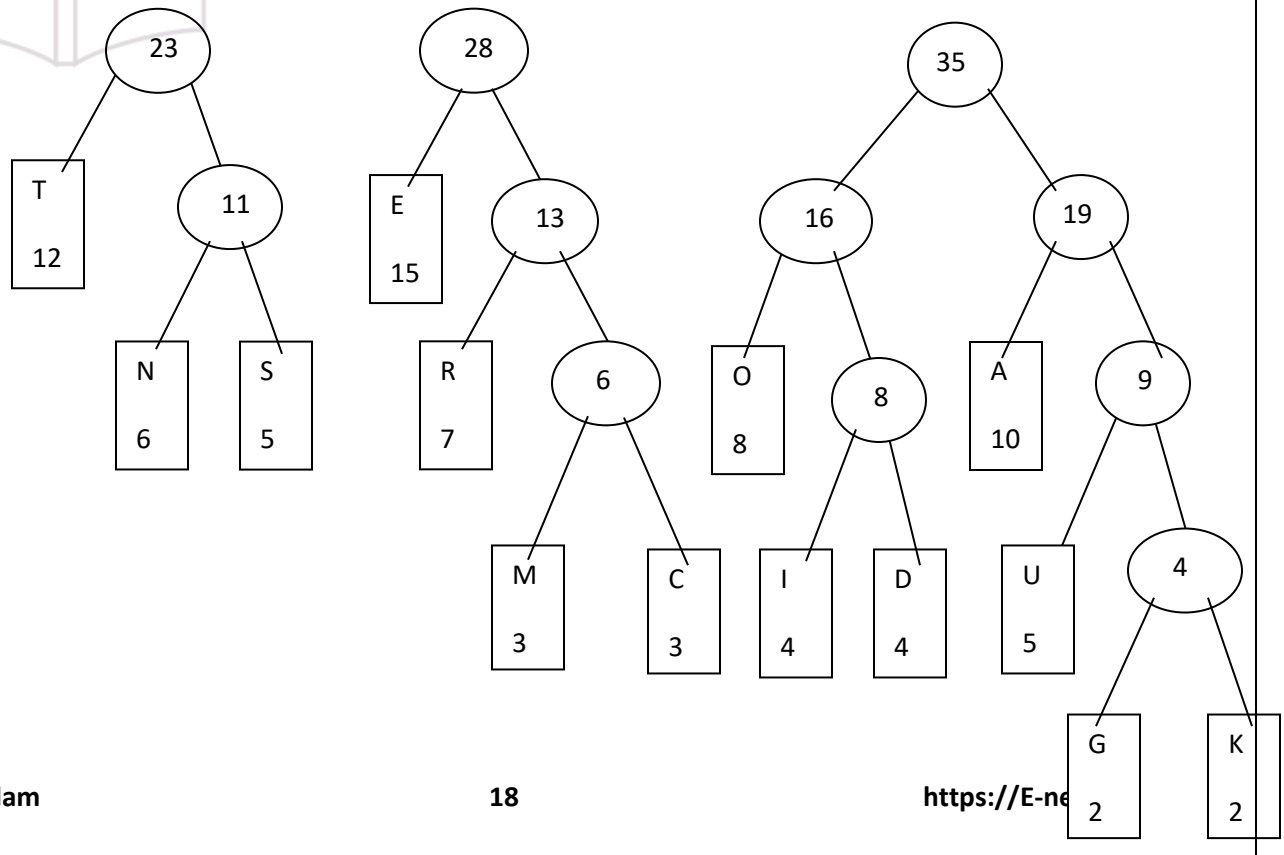


STEP 11:-

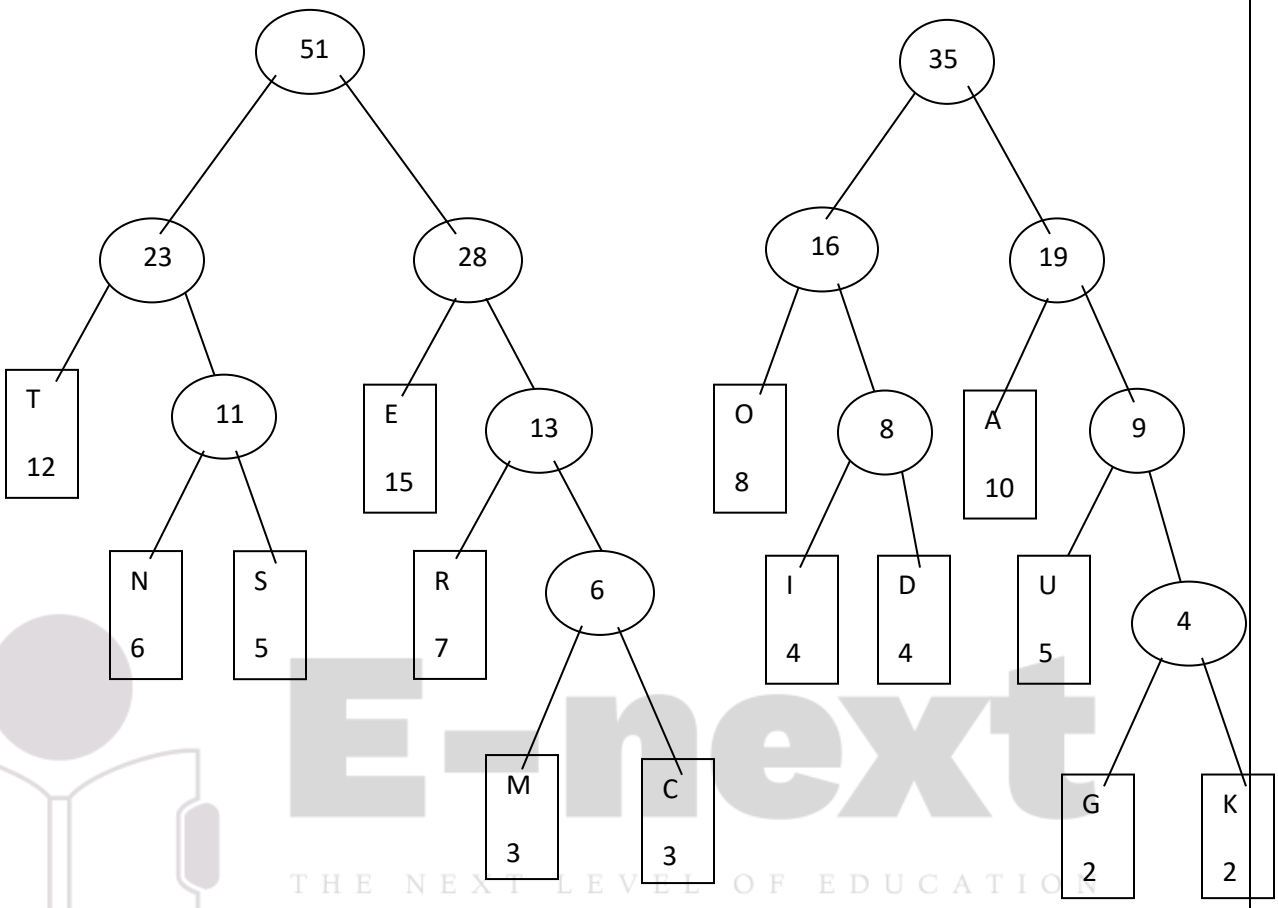
Combining smallest frequency node E and newly formed node of frequency 13 and forming new node.



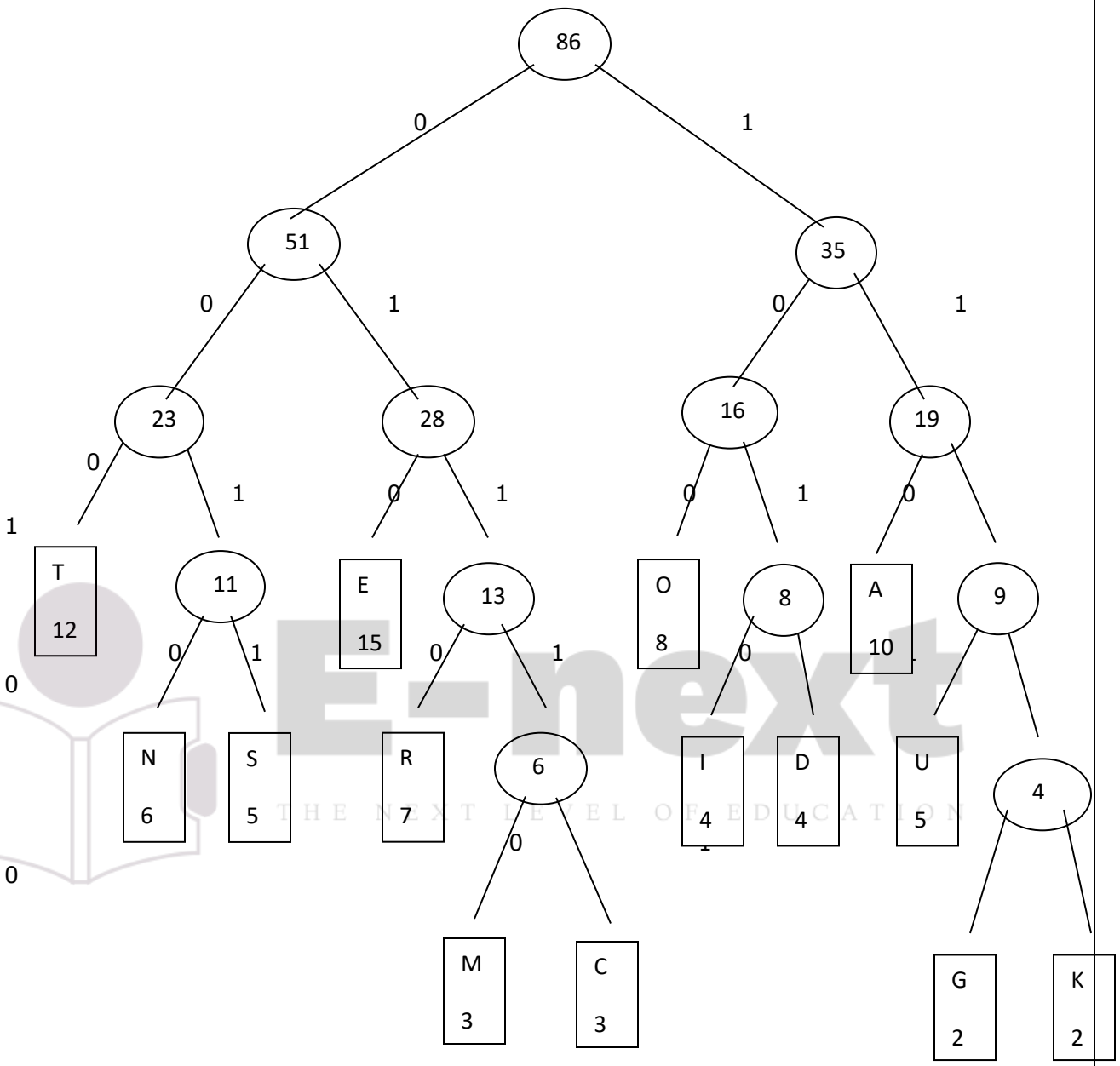
STEP 12:- Combining frequency 11 with frequency 19 and forming new node.



STEP 13:- Combining frequency 23 with frequency 28 and forming new node.



STEP 14:- finally combining frequency 51 with frequency 35 and forming new node.



The huffmam code generates the small code for the most frequent character and larger code for the less frequent character.

The advantage of Huffman code is that it saves the memory space by generatingsmaller codes.

So the code generated are :

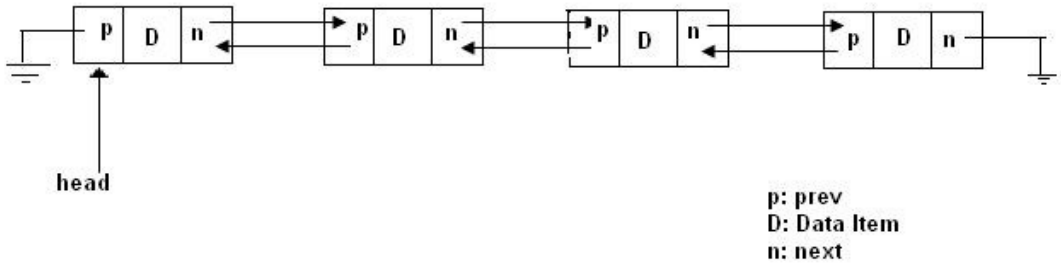
Character code :-

T - 000
E - 010
A - 110
O - 100
R - 0110
N - 0010
S - 0011
U - 1110
I - 1010
D - 1011
M - 01110
C - 01111
G - 11110
K - 11111

From the above codes it is clear that every element gets a unique code. Also , the most frequent characters are assigned with lesser length codes and less frequent characters are assigned with greater length codes.

Define a doubly linked list. Write algorithm for :
(i) Sorting the number of elements in a singly link list.
(ii) Searching an element in a singly linked list.

Doubly-linked list(DLL) is a more sophisticated kind of linked list. In DLL, each node has two links: one points to previous node and one points to next node. The previous link of first node in the list points to a Null and the next link of last node points to Null.



Sorting Algorithm :

```
Algoritm Sort(int list[] ,int last)
{
  Int hold;
  Int walker;
    For(int current = 1 ; current < = last ; current++)
    {
        Hold = list[current];
       Walker >= 0 && hold < list[walker];
       Walker--;
       List[walker+1]=list[walker];
       List[walker+1]=hold;
    }
 return;
}
```

Searching Algorithm :

The code is shown in Program 7-6.

Search User Interface

```
1  /*================== searchList ================
2    Interface to search function.
3       Pre    pList pointer to initialized list.
4              pArgu pointer to key being sought
5       Post   pDataOut contains pointer to found data
6         -or- NULL if not found
7       Return boolean true successful; false not found
8  */
```

Search User Interface (continued)

```
 9  bool searchList (LIST*  pList, void* pArgu,
10                   void** pDataOut)
11  {
12  //Local Definitions
13    bool  found;
14
15    NODE* pPre;
16    NODE* pLoc;
17
18  //Statements
19    found = _search (pList, &pPre, &pLoc, pArgu);
20    if (found)
21         *pDataOut = pLoc->dataPtr;
22    else
23         *pDataOut = NULL;
24    return found;
25  } // searchList
```

Given the set of symbols and corresponding frequency table as below,

Explain the steps to find the Huffman code:

Symbol	A	B	C	D	E	F	G	H	I
Frequency	7	6	4	6	5	1	10	7	8

File compression, particularly for multimedia data, is widely used to reduce Internet traffic and transfer times. Two common compression formats for images are GIF and JPEG. Both of these encoding formats throw away information about the images, so the original image cannot be reconstructed exactly from the compressed image. GIF and JPEG are *lossy compression techniques*. Lossy compression can be very effective for multimedia data. JPEG encoding, for example, can reduce the size of an image by a factor of 20 or more without a noticeable loss of image quality.

Lossy compression cannot be used for text and data files, because you want to get the original file back when you uncompress it (i.e. *lossless compression*). Many lossless compression schemes use variable length encoding. One of the earliest and most commonly used is *Huffman coding*.

Every file can be thought of as a sequence of bytes (values from 0 to 255). Uncompressed files use 8 bits for each possible byte value. The idea of Huffman coding is simply to devise an encoding scheme so that the most frequently occurring byte values are represented by a short code (fewer bits) and the less frequently occurring byte values are represented by a longer code (more bits). Huffman encoding significantly reduces the size of files, but it requires that you be able to read or write a bit at a time.

The algorithm for Huffman coding generates a binary tree whose left and right branches are labeled by 0 and 1 respectively as shown in the diagram below. The leaves of the tree are unique bytes that appear in the file (the *alphabet*). The path from the root to the leaf gives the encoding of the byte represented by that node. Leaves that are close to the root have short encodings and leaves that are farther away have longer encodings. The trick is to generate a tree where the most frequently occurring byte values are placed in leaves close to the root.

Algorithm Overview

Huffman coding is a statistical technique which attempts to reduce the amount of bits required to represent a string of symbols. The algorithm accomplishes its goals by allowing symbols to vary in length. Shorter codes are assigned to the most frequently used symbols, and longer codes to the symbols which appear less frequently in the string

Building a Huffman Tree

The Huffman code for an alphabet (set of symbols) may be generated by constructing a binary tree with nodes containing the symbols to be encoded and their probabilities of occurrence. The tree may be constructed as follows:

Step 1. Create a parentless node for each symbol. Each node should include the symbol and its probability.
Step 2. Select the two parentless nodes with the lowest probabilities.
Step 3. Create a new node which is the parent of the two lowest probability nodes.
Step 4. Assign the new node a probability equal to the sum of its children's probabilities.
Step 5. Repeat from Step 2 until there is only one parentless node left.

Once a Huffman tree is built, Canonical Huffman codes, which require less information to rebuild, may be generated by the following steps:

Step 1. Remember the lengths of the codes resulting from a Huffman tree generated per above.
Step 2. Sort the symbols to be encoded by the lengths of their codes (use symbol value to break ties).
Step 3. Initialize the current code to all zeros and assign code values to symbols from longest to shortest code as follows:

A. If the current code length is greater than the length of the code for the current symbol, right shift off the extra bits.
B. Assign the code to the current symbol.
C. Increment the code value.
D. Get the symbol with the next longest code.
E. Repeat from A until all symbols are assigned codes.

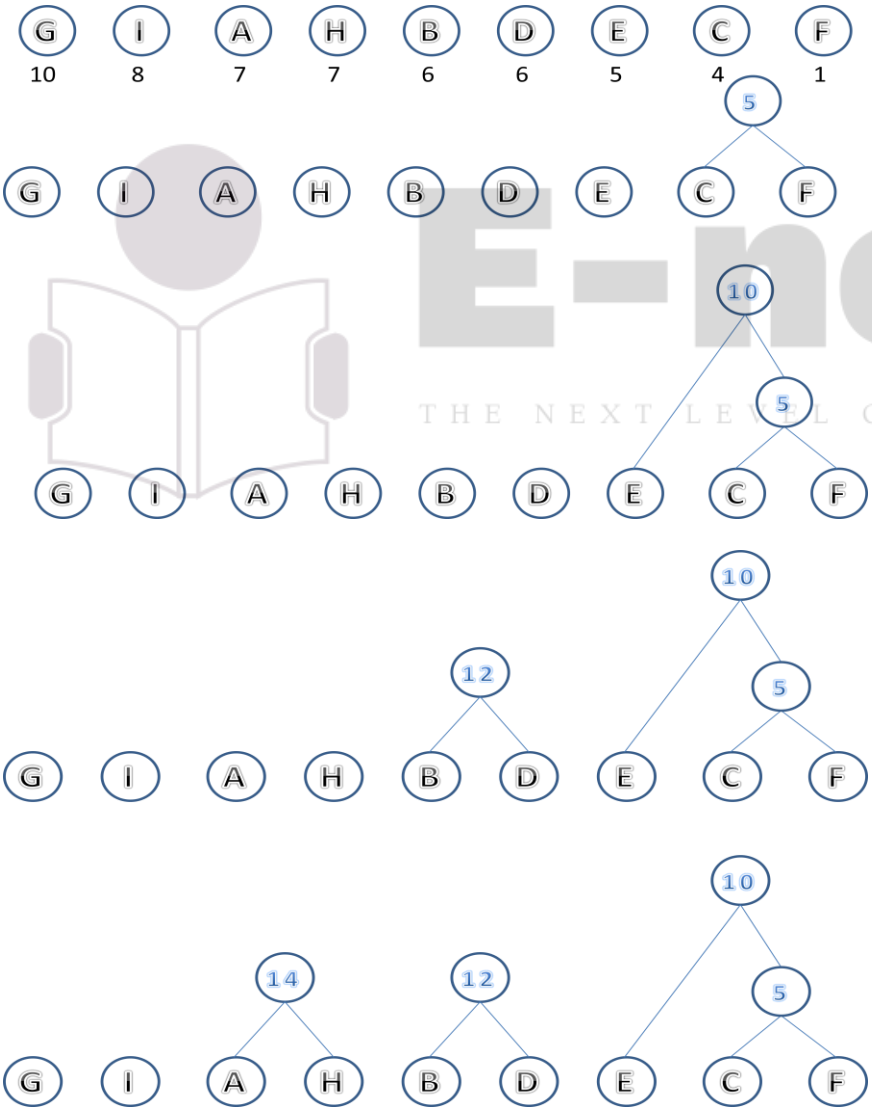
Encoding Data

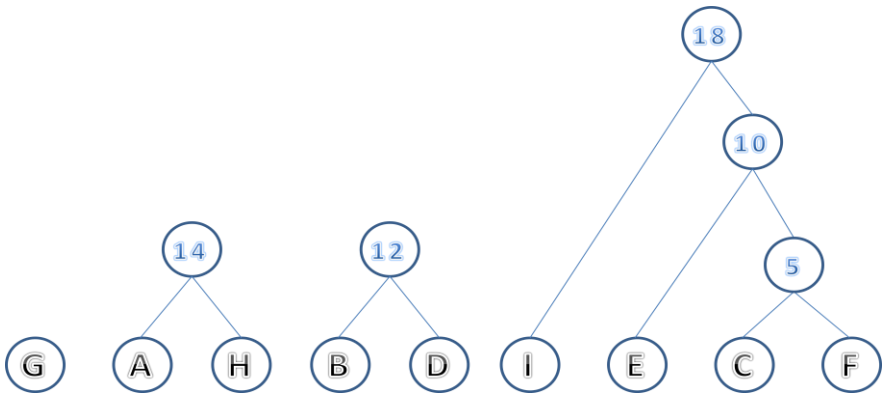
Once a Huffman code has been generated, data may be encoded simply by replacing each symbol with it's code.

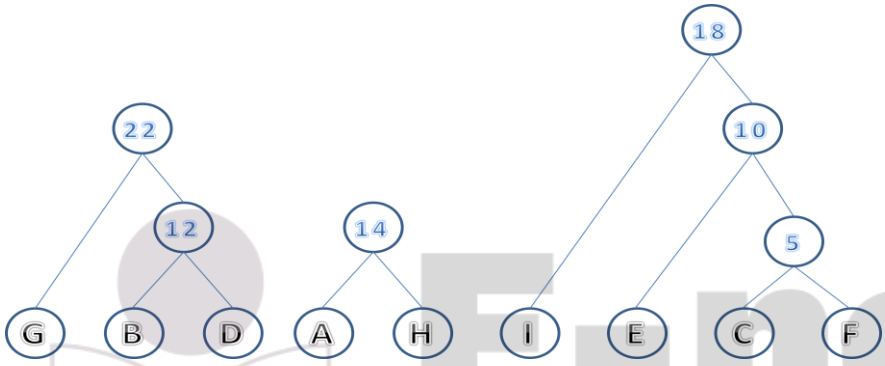
Decoding Data

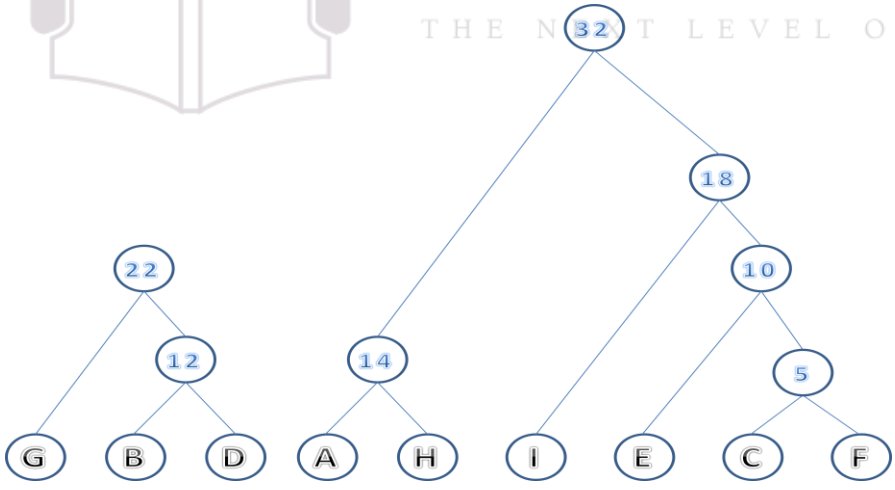
If you know the Huffman code for some encoded data, decoding may be accomplished by reading the encoded data one bit at a time. Once the bits read match a code for symbol, write out the symbol and start collecting bits again.

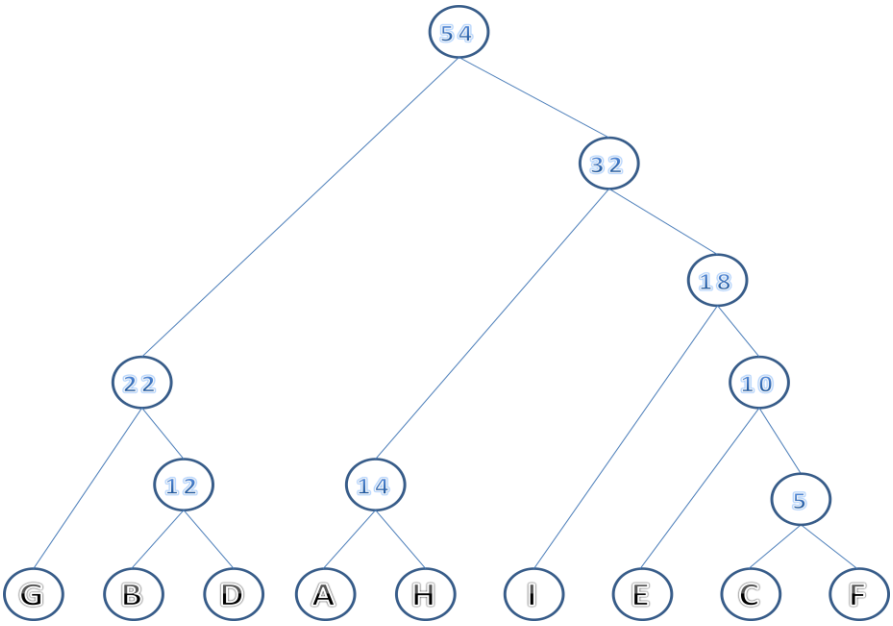
Example:

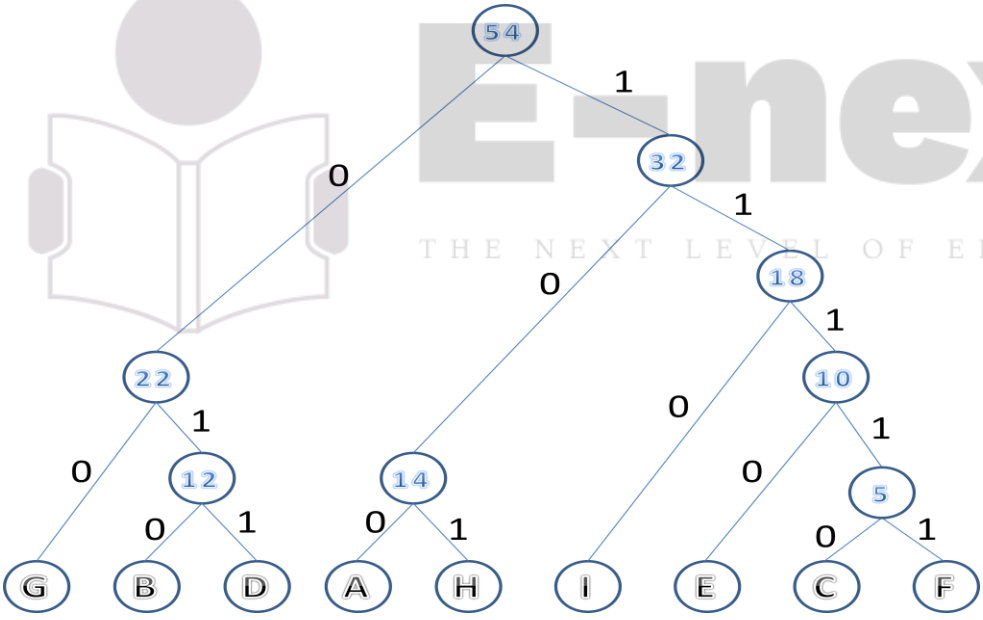












Symbol	Code
A	100
B	010
C	11110
D	011
E	1110
F	11111
G	00
H	101
I	110