```
/*
 * Author: Abhishek Yadav
   Roll No: CS12B032

   // this is the implementation of "Towers of Hanoi problem" that I did as a part of my third semester curriculum
    here
   For this assignment, the second parameter will always be 3.
   where first input is the number of stones on the first peg and second is always three indicating that
   we have to use only three pegs to move all the stones from first peg to any one of other two keeping
   third as a buffer
Example input: 4 3

Output should be continuous print of the 3 lists (in the same order) after every step. The last line should be the number of steps taken to move the disks.

 */

#include "LList.h"
#include <stdio.h>
#include"LList.c"

int play_TOH(LList* lst_1, LList* lst_2 ,LList* lst_3){
    int num_steps=0;
    int lst_1_size,lst_2_size;
    int k=llist_size(lst_1);
    int p=llist_size(lst_3);
    printf("\nTransfer\tpeg1\tpeg2\tpeg3\n");
    if(k%2!=0){
    while(p!=k)
    {
      switch(num_steps%3)
      {

     case 0:
           if(lst_3->head==NULL || lst_1->head->data < lst_3->head->data)
           {
              lst_3=llist_prepend(lst_3,lst_1->head->data);
              lst_1=llist_remove_first(lst_1);
              printf("1 -> 3");
           }
           else
            {
              lst_1=llist_prepend(lst_1,lst_3->head->data);
              lst_3=llist_remove_first(lst_3);
              printf("3 -> 1");
           }
           num_steps++;
           break;

    case 1:
```

```
        if(lst_2->head==NULL || lst_1->head->data < lst_2->head->data)
          {
             lst_2=llist_prepend(lst_2,lst_1->head->data);
             lst_1=llist_remove_first(lst_1);
             printf("1 -> 2");
          }else{
          if(lst_1->head->data==1000)
          lst_1=llist_remove_first(lst_1);
            lst_1=llist_prepend(lst_1,lst_2->head->data);
             lst_2=llist_remove_first(lst_2);
             printf("2 -> 1");
          }
          num_steps++;
          break;

    case 2:
        if(lst_3->head==NULL || lst_2->head->data < lst_3->head->data)
          {
             lst_3=llist_prepend(lst_3,lst_2->head->data);
             lst_2=llist_remove_first(lst_2);
             printf("2 -> 3");
          }
          else
           {
             lst_2=llist_prepend(lst_2,lst_3->head->data);
             lst_3=llist_remove_first(lst_3);
             printf("3 -> 2");
          }
          num_steps++;
          break;
        default :
           break;
     }
     if(lst_1->head==NULL)
     lst_1=llist_prepend(lst_1,1000);
     p=llist_size(lst_3);
     lst_1_size=llist_size(lst_1);
     lst_2_size=llist_size(lst_2);
    if(lst_1->head->data==1000)
     lst_1_size-=1;
     printf("\t\t %d\t%d\t%d\n",lst_1_size,lst_2_size,p);
   }
}else{
     while(p!=k)
     {
      switch(num_steps%3)
      {
      case 0:
          if(lst_2->head==NULL || lst_1->head->data < lst_2->head->data)
          {
             lst_2=llist_prepend(lst_2,lst_1->head->data);
             lst_1=llist_remove_first(lst_1);
```

```
            printf("1 -> 2");
        }
        else
         {
          if(lst_1->head->data==1000)
          lst_1=llist_remove_first(lst_1);
            lst_1=llist_prepend(lst_1,lst_2->head->data);
            lst_2=llist_remove_first(lst_2);
            printf("2 -> 1");
        }
        num_steps++;
        break;

case 1:
     if(lst_3->head==NULL || lst_1->head->data < lst_3->head->data)
        {
            lst_3=llist_prepend(lst_3,lst_1->head->data);
            lst_1=llist_remove_first(lst_1);
            printf("1 -> 3");
        }
        else
         {
            lst_1=llist_prepend(lst_1,lst_3->head->data);
            lst_3=llist_remove_first(lst_3);
            printf("3 -> 1");
        }
        num_steps++;
        break;

case 2:
     if(lst_3->head==NULL || lst_2->head->data < lst_3->head->data)
        {
            lst_3=llist_prepend(lst_3,lst_2->head->data);
            lst_2=llist_remove_first(lst_2);
            printf("2 -> 3");
        }
        else
         {
            lst_2=llist_prepend(lst_2,lst_3->head->data);
            lst_3=llist_remove_first(lst_3);
            printf("3 -> 2");
        }
        num_steps++;
        break;
   default:
        break;
  }
   if(lst_1->head==NULL)
  lst_1=llist_prepend(lst_1,1000);
  p=llist_size(lst_3);
  lst_1_size=llist_size(lst_1);
  lst_2_size=llist_size(lst_2);
```

```
      if(lst_1->head->data==1000)
      lst_1_size=-1;
      printf("\t\t %d\t%d\t%d\n",lst_1_size,lst_2_size,p);

     }
  }
  return num_steps;
}

int main(){
    int D,N;
    scanf("%d%d",&D,&N);
    LList*peg_1=llist_new();
    LList*peg_2=llist_new();
    LList*peg_3=llist_new();
    int i=1;
    while(i<=D){
      peg_1=llist_append(peg_1,i);
      i++;
    }
    int num_of_steps = 0;

    num_of_steps = play_TOH(peg_1,peg_2,peg_3);
    printf(" No of steps= %d\n",num_of_steps);
    return 0;
}

//////////////////////////////////////////////////////////////
LList.c  // List.c starts from here
/*
 * Author:Abhishek Yadav
 * Linked list data structure (Implementation)
 */

#include "LList.h"
#include <stdlib.h>
#include <stdio.h>

// Create a new node with data and next element (can be NULL)
Node* node_new( int data, Node* next )
{
  Node*n = (Node*) malloc( sizeof( Node ) );
  n->data = data;
  n->next = next;
   return n;
}
// Create an empty list (head shall be NULL)
LList* llist_new()
{
  LList* l = (LList*) malloc( sizeof( LList ) );
  l->head = NULL;
  return l;
```

```
}
// Traverse the linked list and return its size
int llist_size( LList* lst )
{
  int i=0;
  Node* n=lst->head;
  while(n!=NULL)
  {
     n=n->next;
     i++;
  }
  return i;
}

// Traverse the linked list and print each element
void llist_print( LList* lst )
{
  Node* n;
  for( n = lst->head; n != NULL; n=n->next )
  {
    printf( "%d ", n->data );
  }
  printf( "\n" );
}

int llist_get( LList* lst, int idx )
{
  int i = 0;
  Node* n=lst->head;
  if(idx<=0 || idx>llist_size(lst))
    return -1;
  while(i<idx){
    n=n->next;
    i++;
    }
  return n->data;
}

// Add a new element at the end of the list
LList* llist_append( LList* lst, int data )
{
    Node*p=lst->head;
    Node*q=(Node*)malloc(sizeof(Node));
    q->data=data;
    q->next=NULL;
    if(p==NULL)
    lst->head=q;
    else{
       while(p->next!=NULL)
          p=p->next;
        p->next=q;
      return lst;
```

```
    }
}

// Add a new element at the beginning of the list
LList* llist_prepend( LList* lst, int data )
{
  Node* n = node_new( data, lst->head );
  lst->head = n;
  return lst;
}
// Add a new element at the @idx index
LList* llist_insert( LList* lst, int idx, int data )
{
  if( idx == 0 )
  {
    Node* p = node_new( data, NULL );
    p->next = lst->head;
    lst->head = p;
  }
  else
  {
    int i = 1;
    Node* n=lst->head;
    Node* p = node_new( data, NULL );
    while(i<idx)
    {
      n=n->next;
      i++;
    }
    p->next=n->next;
    n->next =p;
  }

  return lst;
}

// Remove an element from the end of the list
LList* llist_remove_last( LList* lst )
{
  Node* n;
  if(lst->head==NULL)
  return lst;
  else if( lst->head->next == NULL )
  {
    free( lst->head );
    lst->head = NULL;
  }
  else
  {
    n=lst->head;
    while(n->next->next!=NULL)
      n=n->next;
```

```
    free( n->next );
     n->next = NULL;
  }
  return lst;
}

// Remove an element from the beginning of the list
LList* llist_remove_first( LList* lst )
{
  Node* n = lst->head;
  if(n==NULL)
  return lst;
  lst->head=n->next;
  free(n);
  return lst;
}

// Remove an element from an arbitrary position in the list
LList* llist_remove( LList* lst, int idx )
{
  Node* n=lst->head;
  Node*temp=(Node*)malloc(sizeof(Node));
  if( idx == 0 )
    free( n );
    else if(llist_size(lst)<=idx)
    return lst;
  else
  {
    int i = 1;
    while(i<idx){
      n=n->next;
      i++;}
    temp=n->next;
    n->next=temp->next;
    free(temp);
  }

  return lst;
}
////////////////////////////////////////////////////////
//LList.h starts here

/*
 * Author:Abhishek Yadav
 * Linked list data structure
 */

#ifndef LLIST_H
#define LLIST_H

// Node for the link list
typedef struct Node_ Node;
```

```
struct Node_ {
    int data;
    Node* next;
};

// Create a new node with data and next element (can be NULL)
Node* node_new( int data, Node* next );

// Link list structure
typedef struct LList_ {
    Node* head;
} LList;

// Create an empty list (head shall be NULL)
LList* llist_new();

// Traverse the linked list and return its size
int llist_size( LList* lst );

// Traverse the linked list and print each element
void llist_print( LList* lst );

int llist_get( LList* lst, int idx );
// Add a new element at the end of the list
LList* llist_append( LList* lst, int data );
// Add a new element at the beginning of the list
LList* llist_prepend( LList* lst, int data );
// Add a new element at the @idx index
LList* llist_insert( LList* lst, int idx, int data );

// Remove an element from the end of the list
LList* llist_remove_last( LList* lst );
// Remove an element from the beginning of the list
LList* llist_remove_first( LList* lst );
// Remove an element from an arbitrary position in the list
LList* llist_remove( LList* lst, int idx );

#endif // LLIST_H
```