```c
/*
 * Author : Abhishek Yadav
 * Roll NO.: CS12B032
 *An example maze is as below

        # # # # #
        # E U . #
        # . # . #
        # # . R #
        # # # # #

        'U' is You (sms lingo ;))
        'E' is Enemy
        'R' is reward.
        '#' is wall/pit.
        '.' is free path.

        So, your first step would be to identify your position and the enemy position and find both
your distances to the reward.
        Note, only '#' obstructs the path. 'E' and 'U' do not.

        Input: Size of square matrix, N, followed by the NxN matrix of characters.
        Output: Shortest distance between U and R and in next line, shortest distance between E and
R.

        Example:
        Input:
        5
        # # # # #
        # E U . #
        # . # . #
        # # . R #
        # # # # #

        Output:
        3
        4
   **/
#include<stdlib.h>
#include<stdio.h>
#include "MazeSolver.h"
int shortestDistance( char**maze, int N, int start_x, int start_y, char ch, int *arr, Queue*q,int
distance,int level,int z)
{
  int*flag_arr=malloc(sizeof(int)*4) ;
  int i=0,top,found=0,k,j;
   if(maze[start_y][start_x]==ch)
    return distance;
   else if(start_x<N-1){
      if(maze[start_y][start_x+1]!='#')
       flag_arr[0]=(start_x+1)*(start_x+1)+start_y;
      }
```

```
   if(start_y<N-1){
   if (maze[start_y+1][start_x]!='#')
      flag_arr[1]=start_x*start_x+(start_y+1);
    }
  if(start_x>0)
   {
        if(maze[start_y][start_x-1]!='#')
         flag_arr[2]=(start_x-1)*(start_x-1)+start_y;
   }
   if(start_y>0){
   if (maze[start_y-1][start_x]!='#')
      flag_arr[3]=start_x*start_x+(start_y-1);
    }
    while(i<4)
    {
    int j=0;
         while(j<=z)
         {
             if(flag_arr[i]==arr[j])
             break;
             j++;
         }
         if(j>z && flag_arr[i]>0 && flag_arr[i]<(N*N+N))
         {
             arr[++z]=flag_arr[i];
             q=queue_push(q,flag_arr[i]);
         }
         i++;
    }
    q=queue_pop(q);
     if(level>0)
     level--;
   if(level==0){
   level=queue_size(q);
   distance++;
   }
   top=queue_top(q);
   if(top==-1)
   return -1;
   else {
   for(i=0;i<N;i++)
   {
         for(k=0;k<N;k++){
         if(k*k+i==top)
         {
             found++;
             break;
         }
     }
     if (found==1)
     break;
}
```

```
        distance=shortestDistance(maze, N, k, i, ch, arr, q,distance,level,z);
    }
         return distance;
}
Queue* queue_new()
{
   Queue* node=(Queue*)malloc(sizeof(Queue));
   node=NULL;
   return node;
}
// Deletes the queue, frees memory.
Queue* queue_delete( Queue* st)
{
     Queue*p=st;                //local variable to point to the same address as the passed
      while(p!=NULL)
       p=queue_pop(p);
       return p;
}

// Inserts @val to the back of the queue
Queue* queue_push( Queue* st, int val )
{
      Queue*p=st;
      Queue*q=(Queue*)malloc(sizeof(Queue));
      q->data=val;
      q->link=NULL;
      if(p==NULL)
         st=q;
        else{
              while(p->link!=NULL)
                  p=p->link;

            p->link=q;
      }
      return st;
}

// Remove the element at the front of the queue - also frees memory
Queue* queue_pop( Queue* st )
{
    Queue*p=st;
    Queue*q=(Queue*)malloc(sizeof(Queue));
    q=p;
    st=q->link;
    free(q);
    return st;
}

// Returns the element currently at the top of the queue. If the queue is empty,
// @error should be set to 1, else 0.
int queue_top( Queue* st )
```

```
{
   if(st==NULL)

        return -1;
       else
        return st->data;

}

// Returns the number of elements in the queue
int queue_size( Queue* st )
{
   Queue*p=st;
    int count=0;
    while(p!=NULL)
    {
       p=p->link;
       count++;
    }
    return count;
}


// Prints the elements currently in the queue
void queue_print( Queue* st )
{
     Queue*p=st;
   while(p!=NULL){
   printf("%d  ",p->data);
   p=p->link;
   }
   printf("\n");
}
```