#include <stdio.h>

#include <stdlib.h>

#include <malloc.h>

#define MaxVex 20

typedef char VertexType ; // 顶点的类型

typedef int  EdgeType ;   // 边的类型

typedef struct EdgeNode  // 边表结点

{

    int adjvex ;   // 该弧指向的顶点的位置

    struct EdgeNode \* nextarc ;  // 指向下一条弧的指针

} EdgeNode ;

typedef struct VertexNode    // 顶点表结点

{

    VertexType data ;      // 顶点信息

    EdgeNode \* firstarc ;  // 指向第一条依附该顶点的弧的指针

} VertexNode , AdjList[MaxVex] ;

typedef struct ALGraph  // 图的类型的定义

{

    AdjList vertices ;

    int vexnum ;  // 顶点数

    int arcnum ;  // 边数目

} ALGraph ;

typedef struct Queue  // 定义队列

{

    int data[MaxVex] ;

    int rear ;

    int front ;

}Queue ;

typedef struct Stack  // 定义栈

{

    int data[MaxVex] ;

    int top ;

    int base ;

} Stack ;

bool visited[MaxVex] ;   // 主要针对的是图递归的算法

void InitStack( Stack \* S )

{

    S->base = 0 ;

    S->top = 0 ;

}

bool is\_stack\_empty( Stack \* S )  // 判断栈是否为空

{

    if( S->base == S->top )

    {

        return true ;

    }

    return false ;

}

bool is\_stack\_full( Stack \* S )  // 判断栈是否已满

{

    if( S->top - S->base + 1 == MaxVex )

    {

        return true ;

    }

    return false ;

}

void Push( Stack \* S , int elem )  // 入栈

{

    if( !is\_stack\_full( S ) )

    {

        S->data[S->top++] = elem ;

    }

}

void Pop( Stack \* S , int \* elem ) // 出栈

{

    if( !is\_stack\_empty( S ) )

    {

        \*elem = S->data[--S->top] ;

    }

}

void InitQueue( Queue \* Q )  // 初始化队列

{

    Q->rear = 0 ;

    Q->front = 0 ;

}

bool is\_empty( Queue \* Q )  // 判断队列是否为空

{

    if( Q->front == Q->rear )

    {

        return true ;

    }

    return false ;

}

bool is\_full( Queue \* Q )  // 判断队列是否已满

{

    if( ( Q->rear + 1 ) % MaxVex == Q->front )

    {

        return true ;

    }

    return false ;

}

void EnQueue( Queue \* Q , int elem )  // 入队

{

    if( !is\_full( Q ) )

    {

        Q->data[Q->rear] = elem ;

        Q->rear = ( Q->rear + 1 ) % MaxVex ;

    }

}

void DeQueue( Queue \* Q , int \* elem )  // 出队

{

    if( !is\_empty( Q ) )

    {

        \*elem = Q->data[Q->front] ;

        Q->front = ( Q->front + 1 ) % MaxVex ;

    }

}

// 初始化图

void InitGraph( ALGraph \* G )

{

    G->vexnum = 6 ;

    G->arcnum = 7 ;

}

// 创建图

void CreateGraph( ALGraph \* G )

{

    InitGraph( G ) ;  // 初始化图

    printf("请输入结点信息:") ;

    for( int i = 0 ; i < G->vexnum ; i++ )

    {

        scanf("%c" , &G->vertices[i].data ) ;

        G->vertices[i].firstarc = NULL ;

    }

    printf("请输入弧信息:") ;

    int index ;               // 该弧的其实顶点的下标

    for( int i = 0 ; i < G->arcnum ; i++ )

    {

        EdgeNode \* edge = (EdgeNode\*)malloc(sizeof(EdgeNode)) ;

        scanf("%d %d" , &index , &edge->adjvex ) ;

        edge->nextarc = NULL ;

        EdgeNode \* p = G->vertices[index].firstarc ;

        if( p == NULL )

        {

            G->vertices[index].firstarc = edge ;

        }

        else

        {

            while( p != NULL )

            {

                if( p->nextarc == NULL )

                {

                    break ;

                }

                p = p->nextarc ;

            }

            p->nextarc = edge ;

        }

        // 因为是无向图，所以一条边将使用两次

        EdgeNode \* edge2 = (EdgeNode\*)malloc(sizeof(EdgeNode)) ;

        edge2->adjvex = index ;

        edge2->nextarc = NULL ;

        EdgeNode \* q = G->vertices[edge->adjvex].firstarc ;

        if( q == NULL )

        {

            G->vertices[edge->adjvex].firstarc = edge2 ;

        }

        else

        {

            while( q !=NULL )

            {

                if( q->nextarc == NULL )

                {

                    break ;

                }

                q = q->nextarc ;

            }

            q->nextarc = edge2 ;

        }

    }

}

// 宽度优先遍历

void BFS( ALGraph \* G )

{

    bool visited[MaxVex] ;  // 标示是否已被访问

    int i = 0 ;

    int j = 0 ;

    for( i = 0 ; i < G->vexnum ; i++ )  // 初始化标示都未被访问

    {

        visited[i] = false ;

    }

    for( i = 0 ; i < G->vexnum ; i++ )  // 进入主循环

    {

        if( visited[i] == false )

        {

            printf("%c " , G->vertices[i].data ) ;

            visited[i] = true ;

        }

        EdgeNode \* edge = G->vertices[i].firstarc ;

        while( edge != NULL )

        {

            if( !visited[edge->adjvex] )

            {

                printf("%c " , G->vertices[edge->adjvex].data ) ;  // 访问当前结点

                visited[edge->adjvex] = true ;

            }

            edge = edge->nextarc ;

        } ;

    }

}

// 利用队列实现邻接表的深度优先遍历

void BFS\_Queue( ALGraph \* G )

{

    bool visited[MaxVex] ;  // 标记是否已经访问过

    int i = 0 ;

    int j = 0 ;

    Queue Q ;

    InitQueue( &Q ) ;   // 初始化队列

    for( i = 0 ; i < G->vexnum ; i++ )   // 初始化，表明目前所有顶点都未被访问

    {

        visited[i] = false ;

    }

    for( i = 0 ; i < G->vexnum ; i++ )   // 主循环

    {

        if( !visited[i] )

        {

            EnQueue( &Q , i ) ;

            while( !is\_empty( &Q ) )

            {

                DeQueue( &Q , &i ) ;

                if( !visited[i] )

                {

                    printf("%c " , G->vertices[i].data ) ;   // 访问出队列的顶点

                    visited[i] = true ;  // 标记为已访问的结点

                }

                EdgeNode \* edge = G->vertices[i].firstarc ;

                while( edge != NULL )

                {

                    if( !visited[edge->adjvex] )

                    {

                        EnQueue( &Q , edge->adjvex ) ;

                    }

                    edge = edge->nextarc ;

                }

            }

        }

    }

}

// 利用栈实现图的邻接表结构的深度优先遍历  非递归

void DFS( ALGraph \* G )

{

    bool visited[MaxVex] ;

    int i = 0 ;

    int k = 0 ;

    Stack S ;

    InitStack( &S ) ;

    for( i = 0 ; i < G->vexnum ; i++ )

    {

        visited[i] = false ;

    }

    for( i = 0 ; i < G->vexnum ; i++ )

    {

        if( !visited[i] )

        {

            Push( &S , i ) ;

            while( !is\_stack\_empty( &S ) )

            {

                Pop( &S , &k ) ;

                if( !visited[k] )

                {

                    printf("%c " , G->vertices[k].data ) ;

                    visited[k] = true ;

                }

                EdgeNode \* edge = G->vertices[k].firstarc ;

                while( edge != NULL )

                {

                    if( !visited[edge->adjvex] )

                    {

                        Push( &S , edge->adjvex ) ;

                    }

                    edge = edge->nextarc ;

                }

            }

        }

    }

}

// 初始化全局变量visited

void InitVisit( ALGraph \* G , bool \* visited )

{

    for( int i = 0 ; i < G->vexnum ; i++ )

    {

        visited[i] = false ;

    }

}

// 存储结构为邻接表的图的深度优先遍历的递归实现

// 从下标为v的结点开始遍历

void DFS\_Recursion( ALGraph \* G , int v )

{

    if( !visited[v] )  // 当前结点未被访问

    {

        printf("%c " , G->vertices[v].data ) ;

        visited[v] = true ;

    }

    EdgeNode \* edge = G->vertices[v].firstarc ;

    while( edge != NULL )

    {

        // 从结点v的邻接顶点开始遍历

        if( !visited[edge->adjvex] )  // 当前结点未被访问

        {

            DFS\_Recursion( G , edge->adjvex ) ;

        }

        edge = edge->nextarc ;

    }

}

// 图的递归遍历

// 为防止图为非连通图，必须要有此函数

void DFS\_Traverse\_Recursion( ALGraph \* G )

{

    for( int i = 0 ; i < G->vexnum ; i++ )

    {

        DFS\_Recursion( G , i ) ;

    }

}

// test

int main()

{

    ALGraph G ;

    CreateGraph( &G ) ;  // 创建图

    BFS( &G ) ;  // 宽度优先遍历

    printf("\n") ;

    BFS\_Queue( &G ) ;

    printf("\n") ;

    DFS( &G ) ;

    printf("\n") ;

    InitVisit( &G , visited ) ;

    DFS\_Traverse\_Recursion( &G ) ;

    printf("\n") ;

    return 0 ;

}