

湖南大学



题 目： 图的实现

学生姓名 吴多智

学生学号 201626010520

专业班级 软件 1605

完成日期 2017-12-12

一、需求分析

1. 问题描述

使用邻接表实现无向图。

可以实现以下基本操作：

- 1) 使用邻接表存储无向图，并创建图。
- 2) 使用广度优先搜索遍历、深度优先搜索遍历有向图。
- 3) 打印图的邻接表

2. 输入数据

输入图的顶点个数和边的条数，均为正整数。其中顶点为字符型数据。依次输入各个顶点数据，边数据。

3. 输出数据

- 1) 输出图的基本信息：图顶点数和顶点信息，图类型，边信息
- 2) 输出图的邻接表
- 3) 输出广度遍历和深度遍历后的结果。

4. 测试样例设计

- 1) 顶点数：3

边 数：2

顶点信息：

A B C

边信息：

A B

A C

邻接表为：

A-B-C

B-A

C-A

DFS: A-B-C

BFS: A-B-C

测试目的：此为正则

- 2) 顶点数：0

边数：0

顶点信息：

边信息：

邻接表：

测试目的：当无向图中没有顶点时

- 3) 顶点数：1

边数：0

顶点信息：

A

边信息:

邻接表:

A

测试目的: 当有向图中只有一个顶点时。

4) 顶点数: 4

边数: 5

顶点信息:

A B C D

边信息:

A B

A C

A D

B C

C D

邻接表:

A-B-C-D

B-A-C

C-A-D

D-A-C

DFS:A-B-C-D

BFS:A-B-C-D

测试目的: 有向图为一个单链表时

5) 顶点数: 6

边数: 7

顶点信息:

A B C D E F

边信息:

A-B

A-D

B-C

B-D

B-E

C-F

E-F

邻接表:

A-B-D

B-A-C-D-E

C-B-F

D-A-B

E-B-F

F-C-E

DFS:A-B-C-F-E-D

BFS:A-B-D-C-E-F

二、概要设计

1. 抽象数据类型

1) 数据对象 D:

图是由一个顶点集 V 和一个弧集 E 构成的 数据结构。 $Graph = (V, E)$ 顶点的数据域和边的权可用于存储数据元素

2) 数据关系 R:

$VR = \{ \mid v, w \in V \text{ 且 } P(v, w) \in E \}$ 表示从 v 到 w 的一条弧, 并称 v 为弧头, w 为弧尾。谓词 $P(v, w)$ 定义了弧 的意义或信息。

3) 基本操作:

结构构造/销毁型操作; 获取图顶点、边、查找、遍历、插入, 删除图顶点或边。

2. 算法的基本思想

1) 创建图

首先, 获取图的顶点数和顶点, 获取图的边数和边信息 (两个顶点, 权值)。先构建有向边指向的顶点, 再构建由什么顶点发出该边。

2) 遍历图

广度优先遍历:

首先将图中每个顶点的访问标志设计为未被访问。从图中的某个顶点 V_0 出发, 并在访问此顶点之后依次访问 V_0 的所有未被访问过的 邻接点, 之后按这些顶点被访问的先后次序依次访问它们的邻接点, 直至图中所有 和 V_0 有路径相通的顶点都被访问到。

深度优先遍历:

首先将图中每个顶点的访问标志设计为未被访问。从图中某个顶点 V_0 出发, 访问此顶点, 然后依次从 V_0 的各个未被访问的邻接 点出发深度优先搜索遍历图, 直至图中所 有和 V_0 有路径相通的顶点都被访问到。

3) 打印邻接表

依次打印以每个顶点为头结点组成的单链表。

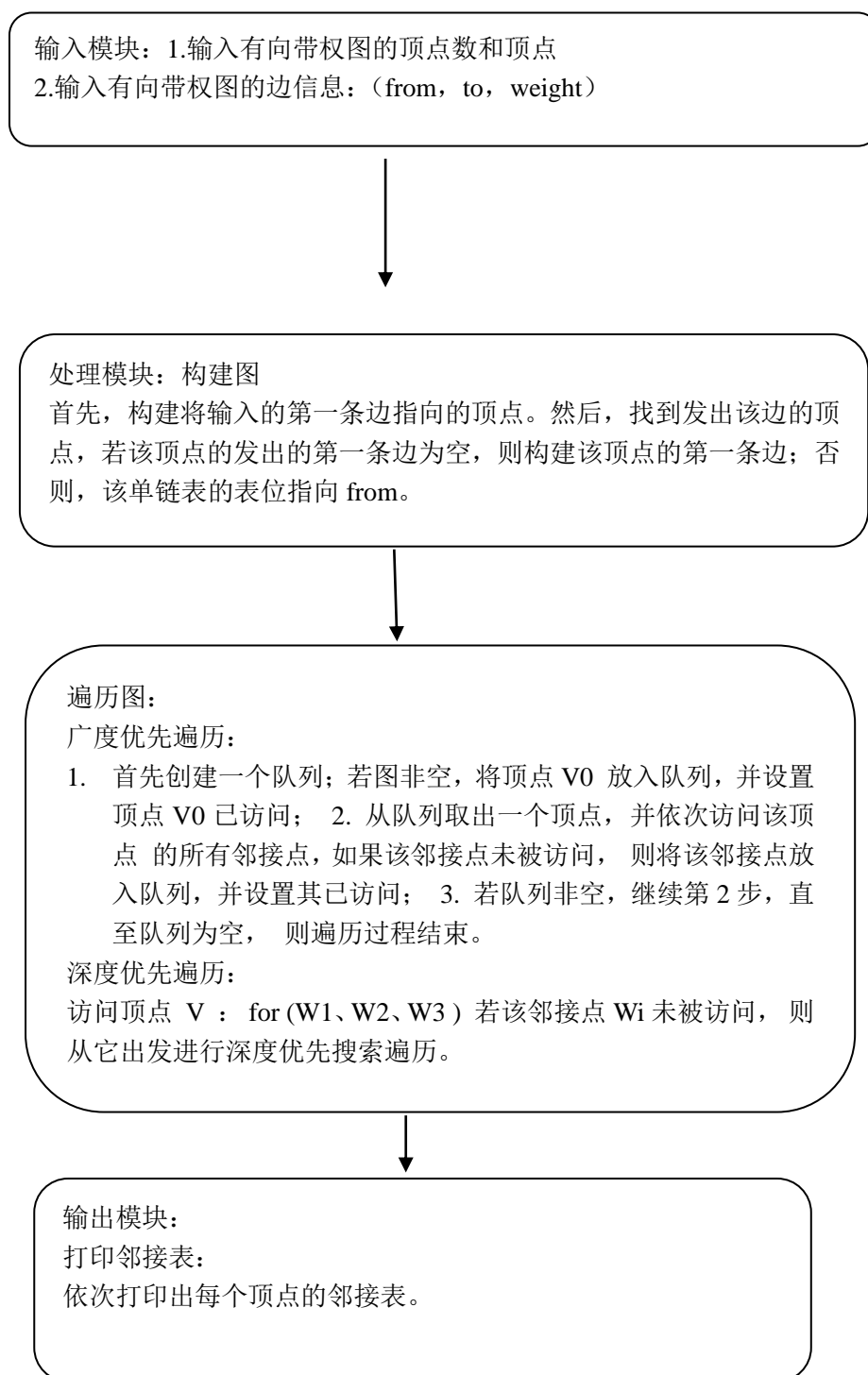
4) 查找某顶点的所有边

查找所有顶点组成的单链表的结点, 若以该顶点构成的单链表中含有所查顶点, 则输出该单链表, 直到查找完所有的单链表。

3. 程序的流程

设计并阐述程序的模块组成, 简单描述每个模块的功能, 整体描述各个模块之间的关系。

输入模块:



三. 详细设计

1. 物理数据类型

首先，需要一个图 ADT。图的 ADT 里应该有顶点和边，顶点操作、边操作，有向图，还是无向图；可以获得顶点邻居的函数，访问边或顶点做标记的函数。

其次，以邻接表的形式存储图。邻接表是将图以多条单链表的形式存储起来，因此，需要一个单链表 ADT。图有多少个顶点就有多少条单链表。

最后，还需要有构成单链表的结点 ADT，结点 ADT 中有数据域和指向下一邻接点的

指针。

图的 ADT:

```
class ListUDG
{
    private: // 内部类
        // 邻接表中表对应的链表的顶点
        class ENode
        {
            public:
                int ivex;           // 该边所指向的顶点的位置
                ENode *nextEdge;    // 指向下一条弧的指针
        };

        // 邻接表中表的顶点
        class VNode
        {
            public:
                char data;          // 顶点信息
                ENode *firstEdge;   // 指向第一条依附该顶点的弧
        };

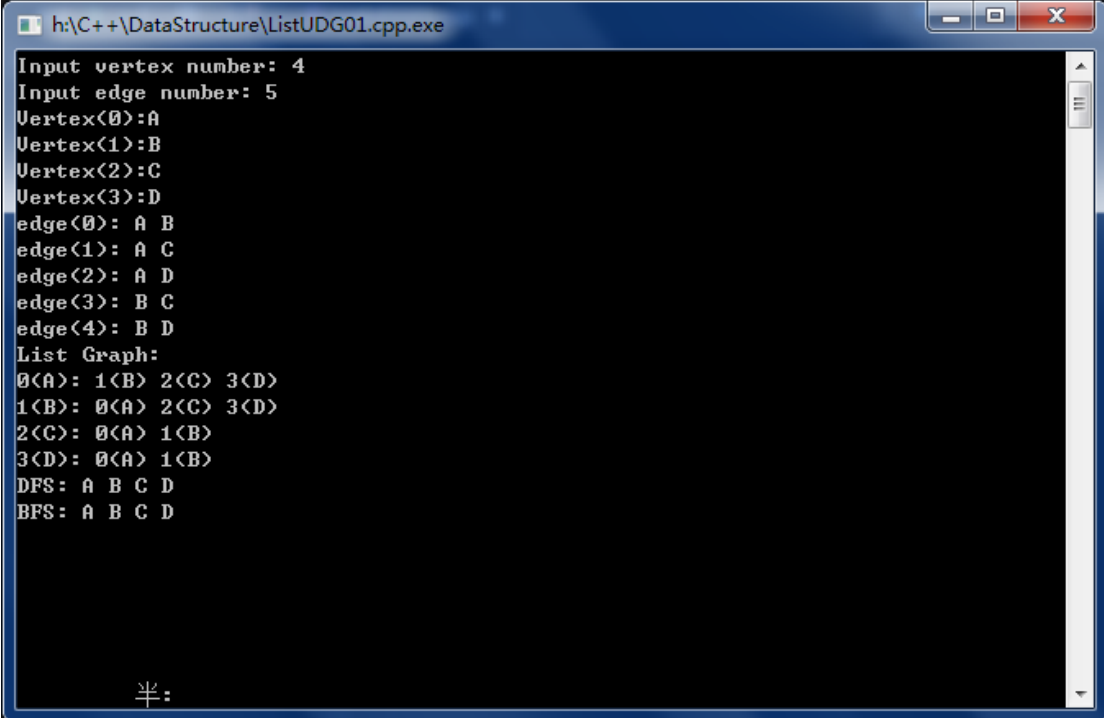
    private: // 私有成员
        int mVexNum;               // 图的顶点的数目
        int mEdgNum;               // 图的边的数目
        VNode mVexs[MAX];

    public:
        // 创建邻接表对应的图(自己输入)
        ListUDG();
        // 创建邻接表对应的图(用已提供的数据)
        ListUDG(char vexs[], int vlen, char edges[][2], int elen);
        ~ListUDG();

        // 深度优先搜索遍历图
        void DFS();
        // 广度优先搜索（类似于树的层次遍历）
        void BFS();
        // 打印邻接表图
        void print();
};
```

```
private:
    // 读取一个输入字符
    char readChar();
    // 返回 ch 的位置
    int getPosition(char ch);
    // 深度优先搜索遍历图的递归实现
    void DFS(int i, int *visited);
    // 将 node 节点链接到 list 的最后
    void linkLast(ENode *list, ENode *node);
};
```

2. 输入和输出的格式



```
h:\C++\DataStructure\ListUDG01.cpp.exe
Input vertex number: 4
Input edge number: 5
Vertex<0>:A
Vertex<1>:B
Vertex<2>:C
Vertex<3>:D
edge<0>: A B
edge<1>: A C
edge<2>: A D
edge<3>: B C
edge<4>: B D
List Graph:
0<A>: 1<B> 2<C> 3<D>
1<B>: 0<A> 2<C> 3<D>
2<C>: 0<A> 1<B>
3<D>: 0<A> 1<B>
DFS: A B C D
BFS: A B C D

半:
```

3. 算法的具体步骤

针对每一个模块，设计并阐述算法的具体步骤，要用文字的形式描述步骤，也可以用流程图的形式描述步骤，要给出伪代码。如果一个模块算法复杂，可以采用分为更小功能模块的方式来分别阐述。

1. 输入/处理模块:

```
ListUDG::ListUDG() {
    char c1, c2;
    int v, e;
```

```

int i, p1, p2;
ENode *node1, *node2;

//输入"顶点数" 和 "边数"
cout<<"Input vertex number: ";
cin>>mVexNum;
cout<<"Input edge number: ";
cin>>mEdgNum;

if ( mVexNum < 1 || mEdgNum < 1 || (mEdgNum > (mVexNum *
(mVexNum-1)))) {
    cout << "Input error: invalid parameters!" << endl;
    return ;
}

//初始化"邻接表"的顶点
for(i=0;i<mVexNum;i++) {
    cout<<"Vertex("<<i<<"): ";
    mVexs[i].data = readChar();
    mVexs[i].firstEdge = NULL;
}

//初始化邻接表的边
for(i=0;i<mEdgNum;i++) {
    // 读取边的起始顶点和结束顶点
    cout << "edge(" << i << "): ";
    c1 = readChar();
    c2 = readChar();
    p1 = getPosition(c1);
    p2 = getPosition(c2);
    //初始化 node1
    node1 = new ENode();
    node1->ivex = p2;
    if(mVexs[p1].firstEdge == NULL) {
        mVexs[p1].firstEdge = node1;
    }else{
        linkLast(mVexs[p1].firstEdge, node1);    //添加到末尾
    }
}

```



```

        //初始化 node2
        node2 = new ENode();
        node2->ivex = p1;
        // 将 node2 链接到“p2 所在链表的末尾”
        if(mVexs[p2].firstEdge == NULL){
            mVexs[p2].firstEdge = node2;
        }else{
            linkLast(mVexs[p2].firstEdge, node2);
        }
    }
}

```

2、深度遍历图

```

/*
 * 深度优先搜索遍历图
 */
void ListUDG::DFS() {
    int i;
    int visited[MAX];        // 顶点访问标记

    // 初始化所有顶点都没有被访问
    for (i = 0; i < mVexNum; i++)
        visited[i] = 0;

    cout << "DFS: ";
    for (i = 0; i < mVexNum; i++){
        if (!visited[i])
            DFS(i, visited);
    }
    cout << endl;
}

```

3、广度遍历图

```

/*
 * 广度优先搜索（类似于树的层次遍历）
 */
void ListUDG::BFS() {
    int head = 0;
    int rear = 0;
    int queue[MAX];

```

```

int visisted[MAX];
int i, j, k;
ENode * node;

for(i=0; i<mVexNum; i++) {
    visisted[i] = 0;
}

cout<<"BFS: ";

for(i=0; i<mVexNum; i++) {
    if(!visisted[i]) {
        visisted[i] = 1;
        cout<<mVexs[i].data<<" ";
        queue[rear++] = i; //入队列
    }
    while(head!=rear) {
        j = queue[head++];
        node = mVexs[j].firstEdge;
        while(node!=NULL) {
            k = node->ivex;
            if(!visisted[k]) {
                visisted[k] = 1;
                cout<<mVexs[k].data<<" ";
                queue[rear++] = k;
            }
            node = node->nextEdge;
        }
    }
}

cout<<endl;
}

```

4、输出模块:

```

/*
 * 打印邻接表图
 */
void ListUDG::print() {
    int i, j;

```

```

    ENode *node;

    cout << "List Graph:" << endl;
    for (i = 0; i < mVexNum; i++) {
        cout << i << "(" << mVexs[i].data << "): ";
        node = mVexs[i].firstEdge;
        while (node != NULL) {
            cout << node->ivex << "(" << mVexs[node->ivex].data << " )";
            node = node->nextEdge;
        }
        cout << endl;
    }
}

```

2. 算法的时空分析

构建图:

邻接表的空间代价与图中边的数目和顶点数目都有关系。每个顶点都要一个数组元素的位置（即使该顶点没有边），而每条边必须出现在其中某个顶点的链表中，所以邻接表的空间代价为 $\Theta(|V|+|E|)$

基于邻接表的图算法，须查看某个顶点和与其相邻的实际的边，其总时间代价为 $\Theta(|V|+|E|)$ 。

遍历图:

深度遍历：在有向图中，DFS 对每一条边处理一次。每个顶点一定会访问到，而且只能访问一次，所以总代价为 $(|V| + |E|)$ 。

广度遍历：在有向图中，BFS 对每一条边处理一次。在无向图中，BFS 对每一条边都从两个方向处理一次。每个顶点一定会访问到，而且只能访问一次，所以总代价为 $(|V| + |E|)$

查找某一顶点所在的链表:

是基于邻接表的图算法，须查看某个顶点和与其相邻的实际的边，其总时间代价为 $\Theta(|V|+|E|)$ 。

输入，输出:

时间代价为 $\Theta(|V|+|E|)$ ，空间代价为 $\Theta(|V|+|E|)$ 。

综上，该程序的时间代价为 $\Theta(|V|+|E|)$ ，空间代价为 $\Theta(|V|+|E|)$ 。

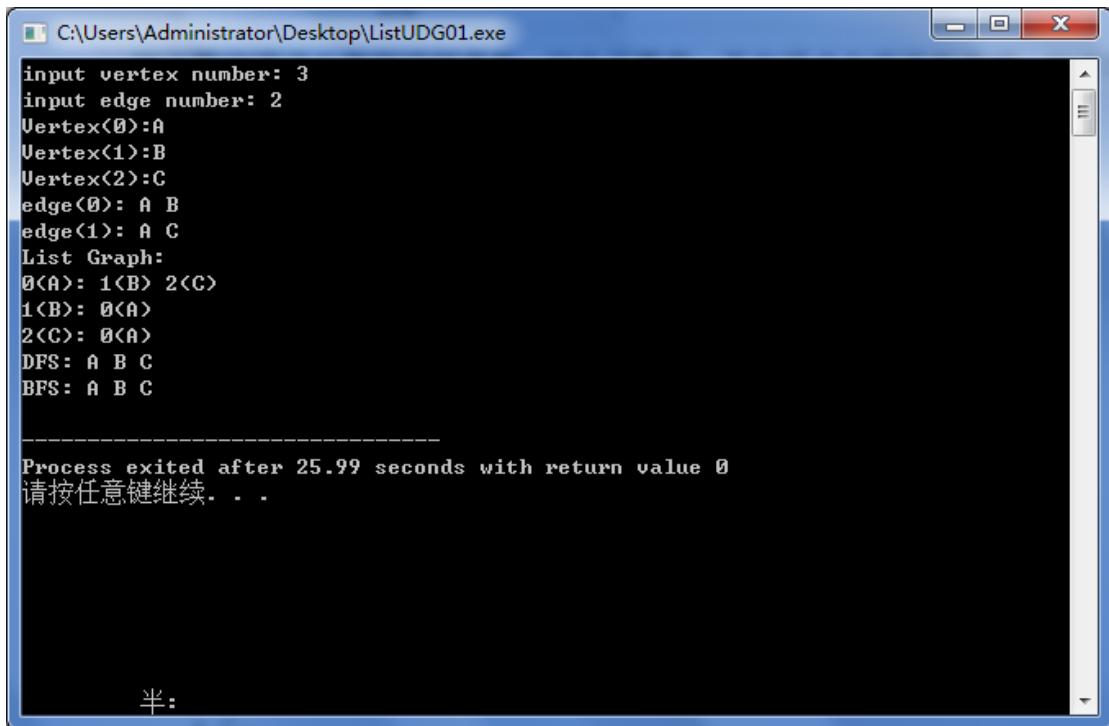
四、调试分析

1. 调试方案设计

邻接表涉及到链表的操作，需要调试查看程序的运行情况；遍历图的时候，也需要调试查看程序是否按照图的情况来遍历节点。

五、测试结果

1)

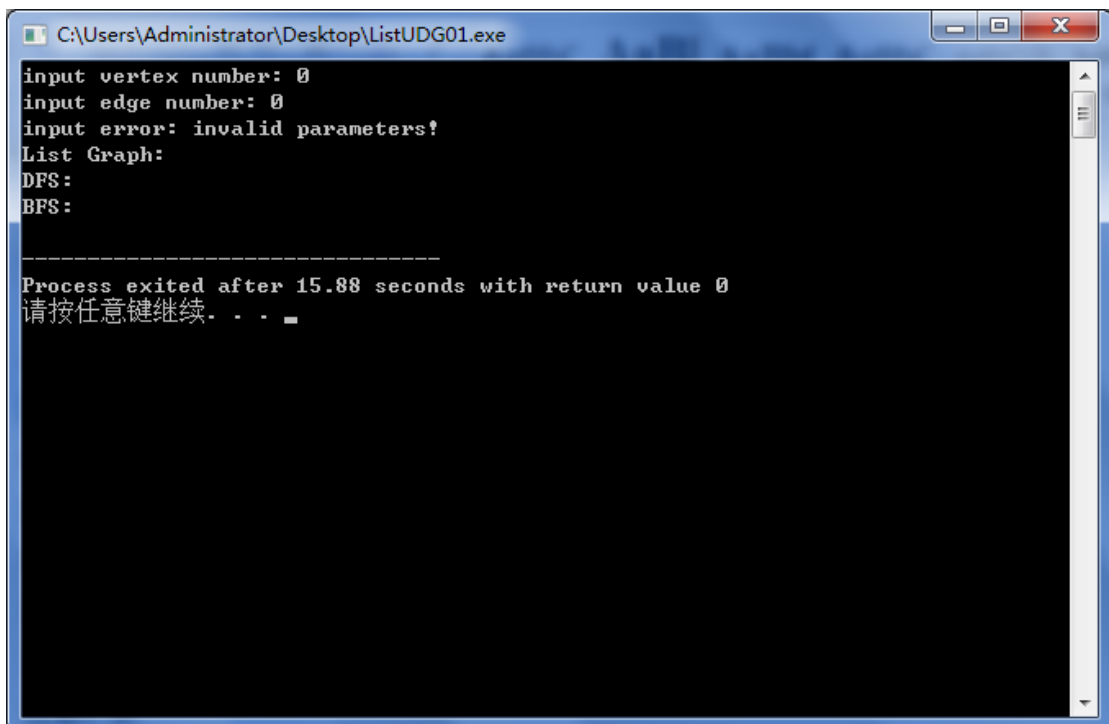


```

C:\Users\Administrator\Desktop>ListUDG01.exe
input vertex number: 3
input edge number: 2
Vertex<0>:A
Vertex<1>:B
Vertex<2>:C
edge<0>: A B
edge<1>: A C
List Graph:
0<A>: 1<B> 2<C>
1<B>: 0<A>
2<C>: 0<A>
DFS: A B C
BFS: A B C

-----
Process exited after 25.99 seconds with return value 0
请按任意键继续. . .
半:
    
```

2)



```

C:\Users\Administrator\Desktop>ListUDG01.exe
input vertex number: 0
input edge number: 0
input error: invalid parameters!
List Graph:
DFS:
BFS:

-----
Process exited after 15.88 seconds with return value 0
请按任意键继续. . .
    
```

3)

```

C:\Users\Administrator\Desktop>ListUDG01.exe

input vertex number: 1
input edge number: 0
Vertex<0>:A
List Graph:
0<A>:
DFS: A
BFS: A

-----
Process exited after 5.654 seconds with return value 0
请按任意键继续. . .

半:
    
```

4)

```

C:\Users\Administrator\Desktop>ListUDG01.exe

input vertex number: 4
input edge number: 5
Vertex<0>:A
Vertex<1>:B
Vertex<2>:C
Vertex<3>:D
edge<0>: A B
edge<1>: A C
edge<2>: A D
edge<3>: B C
edge<4>: B D
List Graph:
0<A>: 1<B> 2<C> 3<D>
1<B>: 0<A> 2<C> 3<D>
2<C>: 0<A> 1<B>
3<D>: 0<A> 1<B>
DFS: A B C D
BFS: A B C D

-----
Process exited after 50.18 seconds with return value 0
请按任意键继续. . .

半:
    
```

5)

```

C:\Users\Administrator\Desktop>ListUDG01.exe
Vertex<2>:C
Vertex<3>:D
Vertex<4>:E
Vertex<5>:F
edge<0>: A B
edge<1>: A D
edge<2>: B C
edge<3>: B D
edge<4>: B E
edge<5>: C F
edge<6>: E F
List Graph:
0<A>: 1<B> 3<D>
1<B>: 0<A> 2<C> 3<D> 4<E>
2<C>: 1<B> 5<F>
3<D>: 0<A> 1<B>
4<E>: 1<B> 5<F>
5<F>: 2<C> 4<E>
DFS: A B C F E D
BFS: A B D C E F

-----
Process exited after 48.13 seconds with return value 0
请按任意键继续. . .
半:

```

六、实验心得

课本中把图定义为高级数据结构,最初不理解为什么图是高级的数据结构,通过实验后,终于理解了。图中的知识用到的前面的基本数据结构,用邻接表存储图用到了前面学到的链表,图的遍历用到了栈及队列的知识,几乎用到了前面学的大部分知识,不得不说是高级数据结构啊。在对图遍历时,用到栈与队列,确实方便与高效许多,对栈与队列的特点有了更深刻的理解。