数据结构课程设计

项目说明文档

电网建造造价模拟系统

作者姓名: 汪明杰

学　　号: 1851055

指导教师: 张　颖

学院专业: 软件学院 软件工程

同济大学

Tongji University

目录

1 项目分析

1.1 项目背景

图（Graph）结构是一种非线性的数据结构，图在实际生活中有很多例子，比如交通运输网，地铁网络，社交网络，计算机中的状态执行（自动机）等等都可以抽象成图结构。图结构是一种比树结构更加复杂的非线性结构。

电网是电力系统中各种电压的变电及输配电线路组成的整体，在一个国家中电网的构建极为重要。但与此同时，如何合理的构造一个电网系统也是需要被解决的一个问题。为了使得构建电网的总工程造价最低，要设计一个能够满足要求的造价方案通常需要借助最小生成树来完成。

1.2 项目需求分析

针对于电网建造造价模拟系统，本项目在实现的过程中，考虑并且满足了以下的需求：

- ✓ **功能完善**

  系统所构造的用来解决电网构建问题的最小生成树应当正确无误。

- ✓ **执行效率高**

  针对数据量比较大的情况，本系统也应该具有在较短时间内求解出正确答案的能力。

- ✓ **代码可读性强**

  本项目在实现过程中，将代码根据功能的不同划分为了不同的代码块，同时进行了合理封装。

- ✓ **健壮性**

  当用户输入的数据不合理时，系统应当给予相应的提示而非直接报错。

- ✓ **可视化**

  该系统通过打印相应操作，使得各种操作可以直观的在使用时被用户感知。

1.3 项目要求

1.3.1 功能要求

假设一个城市有 n 个小区，要实现 n 个小区之间的电网都能够相互接通，构造这个城市 n 个小区之间的电网，使总工程造价最低。请设计一个能够满足要求的造价方案。

在每个小区之间都可以设置一条电网线路，都要付出相应的经济代价。n 个小区之间最多可以有$\frac{n(n-1)}{2}$条线路，选择其中的 n-1 条使总的耗费最少。

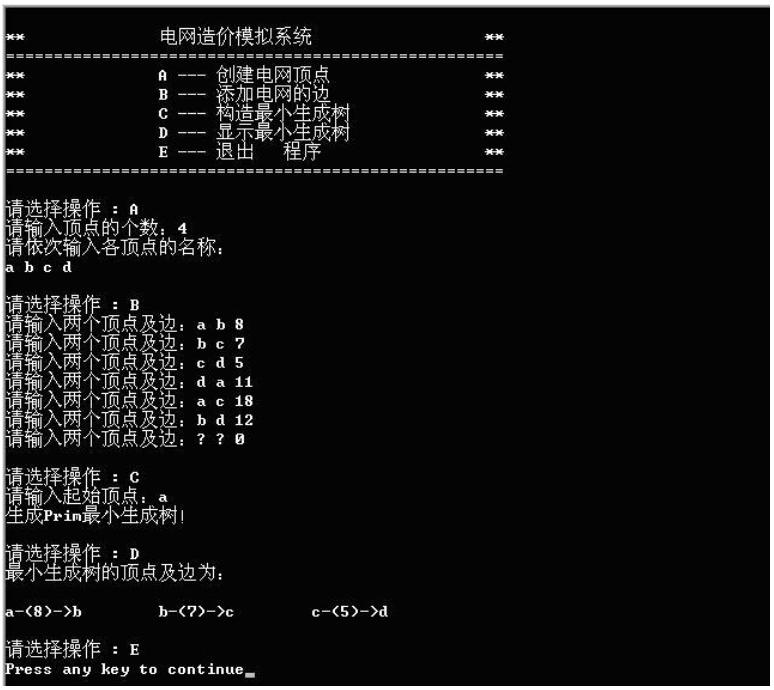
1.3.2 输入格式

输入相应的操作。

1.3.3 输出格式

执行相应的操作，绘制最小生成树。

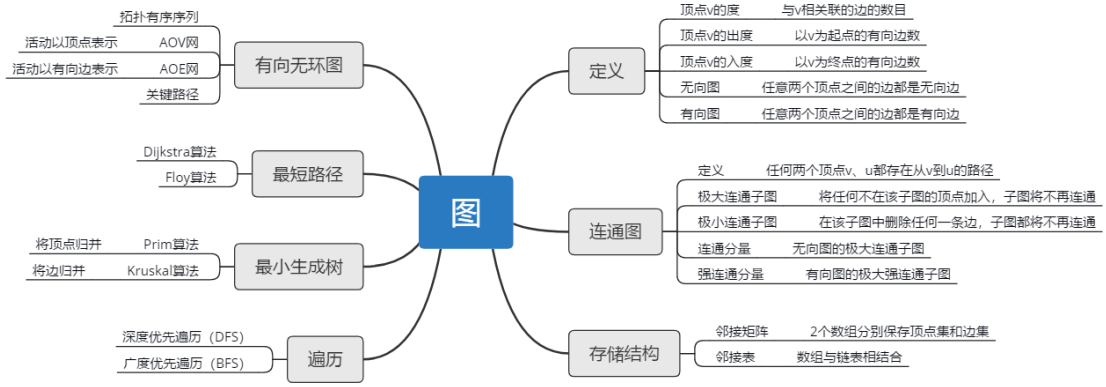
1.3.4 项目示例



2 项目设计

2.1 数据结构设计

图是由若干给定的点及连接两点的线所构成的图形，这种图形通常用来描述某些事物之间的某种特定关系，用点代表事物，用连接两点的线表示相应两个事物间具有这种关系。



如上图所示，本项目中设计了一个**图**（Graph）类。由于本项目中所述电网之间任意两个顶点不存在方向关系，因此属于无向图，即任意两个顶点之间的边都是无向边。

此外，图的存储结构有两种：一种是邻接矩阵法，另一种则是邻接表法。这两种存储结构在执行下列操作时的时间复杂度如下所示：

	邻接矩阵	邻接表
遍历	O (n^2)	O（n+e)
判断某一边	O（1）	O (n)
Prim 算法	O（n^2）	O（n+e)

为了降低时间复杂度，本项目中采用了邻接表法存储图。为了实现邻接表，本项目中使用了向量类（Vector）和双向链表类（List）。

2.2 类设计

2.2.1 向量类（Vector）

向量（Vector）是一个封装了动态大小数组的顺序容器（Sequence Container）。跟任意其它类型容器一样，它能够存放各种类型的对象。可以简单的认为，向量是一个能够存放任意类型的动态数组。需要注意到的是，向量具有以下两个特性：

（1） 顺序序列：

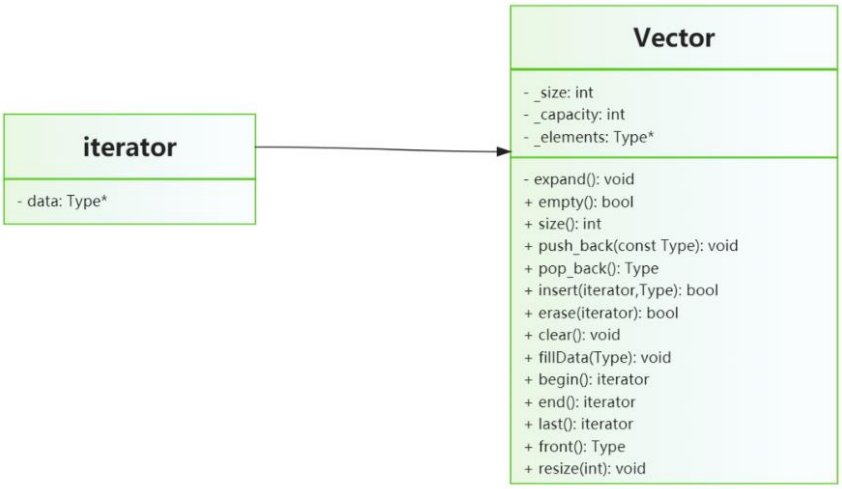
顺序容器中的元素按照严格的线性顺序排序，可以通过元素在序列中的位置访问对应的元素。本类中即通过重载取下标运算符[]实现了此功能。

（2） 动态数组：

支持对序列中的任意元素进行快速直接访问，甚至可以通过指针算术进行该操作，提供了在序列末尾相对快速地添加或者删除元素的操作。

本项目为了实现向量类，在内部定义了一个动态指针以及当前数组的大小。同时，封装了较多的函数以便使用。此外，为了便于对向量进行遍历、插入、删除、查找等操作，增加了一个 iterator 类。iterator 类内部存储一个数组元素指针，可以直接对向量的每个元素进行操作。同时，通过运算符重载相应的自增、自减、判等等操作。

该类和其内部的 iterator 类的 UML 图如下所示：



本类中的主要函数如下所示：

◆ `inline int size()const`

返回向量中元素的个数，也即_size 的大小。

◆ `inline bool empty()const`

判断向量是否为空，也即_size 是否为 0。

◆ `void push_back(const Type data)`

在向量尾端加入一个元素。

◆ `Type pop_back()`

删除向量最后一个元素，并且加以返回。

◆ `bool insert(const Vector<Type>::iterator place, Type item)`

在指定迭代器的位置插入元素，返回是否插入成功。

◆ `bool erase(const Vector<Type>::iterator place)`

删除指定迭代器位置的元素，返回是否删除成功。

◆ `void clear()`

清空向量中所有元素。

◆ `void fillData(const Type data)`

将向量中的元素统一赋值为同一个值。

◆ `iterator begin()`

返回向量首元素的迭代器。

◆ `iterator end()`

返回向量尾元素的下一个位置的迭代器。

◆ `iterator last()`

返回向量尾元素的迭代器。

◆ `Type& front()const`

返回向量的首个元素值。

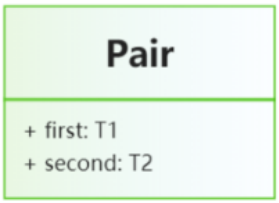
◆ `void resize(int sz)`

将向量重新设定大小，如果变小则删除多余元素。

2.2.2 Pair 类（Pair）

为了方便对于存在一定关系的数据进行操作，Pair 类能够同时存储两种不同类型的数据结构。因此，在函数需要同时返回两个参数的时候，该类就能发挥比较便利的作用。

该类的 UML 图如下所示：

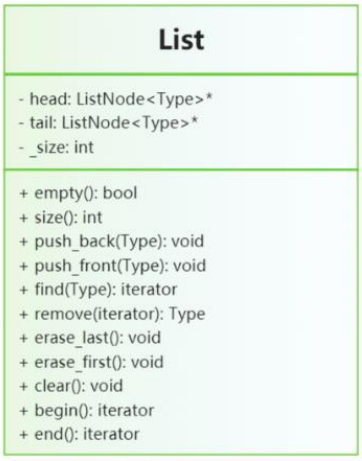


因此，在使用 Pair 类的时候，即可同时存储两种数据类型。

2.2.3 双向链表（List）

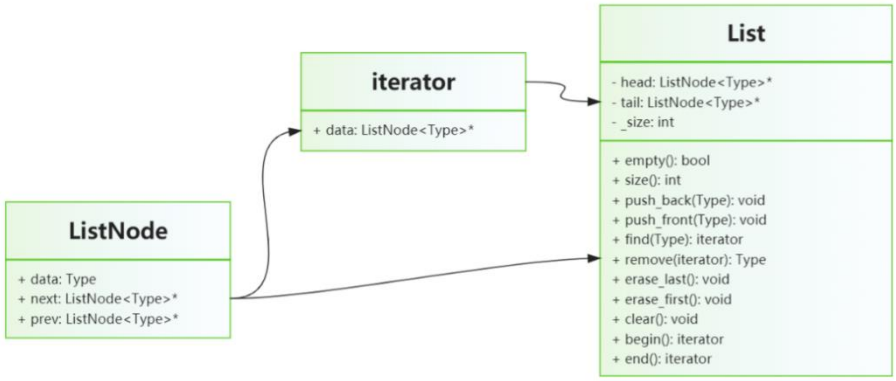
链表的实现原理大同小异，不同之处在于：是否带头结点、是否带尾结点、每一个结点是否带前驱结点等。为了使得链表中各种操作的时间复杂度都尽可能低，因此这里选择了带头结点和尾结点的双向链表来实现链表中的各种操作。也即：选择了牺牲空间来达到降低时间复杂度的效果。

链表的 UML 图如下所示：



为了便于对链表进行遍历、插入、删除、查找等操作，增加了一个 iterator 类。iterator 类内部存储一个链表节点指针，同时，通过运算符重载相应的自增、自减、判等等操作。

iterator 类、ListNode 类和 List 类的关系如下图所示：



其中，List 类中的主要函数如下所示：

◆ `inline int size()const`

返回链表中结点的个数，不包括头结点。

◆ `inline bool empty()const`

判断链表是否为空，也即链表中结点的个数是否为 0。

◆ `void push_back(Type data)`

在链表尾部插入新的数据，也即新增一个结点并且加入到链表末端。

◆ `void push_front(Type data)`

在链表头部插入新的数据，也即新增一个结点并且加入到链表的头部。

◆ `iterator find(const Type& data)const`

在链表中查找值为 data 的元素是否存在，返回该位置的迭代器，若查找失败返回空指针对应的迭代器。

◆ `Type remove(iterator index)`

移除迭代器所处位置的元素，返回移除位置元素的值。

◆ `void erase_last()`

移除链表末端的元素，即最后的结点。

◆ `void erase_first()`

移除链表首端的元素，即第一个结点。

◆ `void clear()`

清空链表，即删除链表中所有的结点。

◆ `iterator begin()`

返回链表第一个元素所在位置的迭代器。

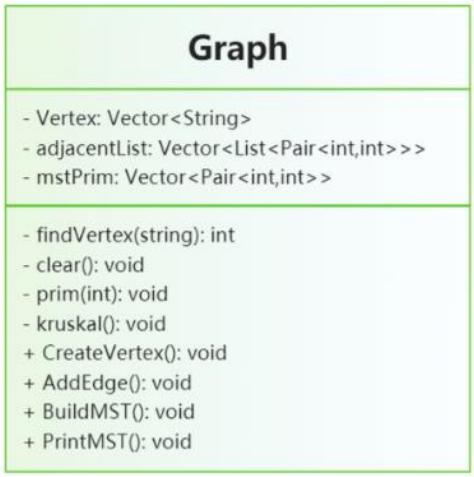
◆ `iterator end()`

返回链表尾结点后的空结点的迭代器。

2.2.4 图类（Graph）

一个图的内部主要需要存储结点和表。针对于结点，项目采用定义一个 Vertex 来进行存储；而针对边，则采用 adjacentList 来存储。此外，为了满足项目中的要求，类内部还定义了用于产生最小生成树的函数、用来展示最小生成树的函数。

图的 UML 图如下所示：



类中主要函数如下所示：

◆ `void CreateVertex()`

该函数用于给图中初始化顶点。

◆ `void AddEdge()`

该函数用于为图中增加边。

- `void BuildMST()`

该函数用于生成 Prim 最小生成树。

- `void PrintMST()`

该函数用于打印 Prim 最小生成树。

- `int findVertex(const string& vetex)`

根据顶点的名称查找到它所对应的索引值。

- `void clear()`

该函数用于清空类内数据。

- `void prim(int index)`

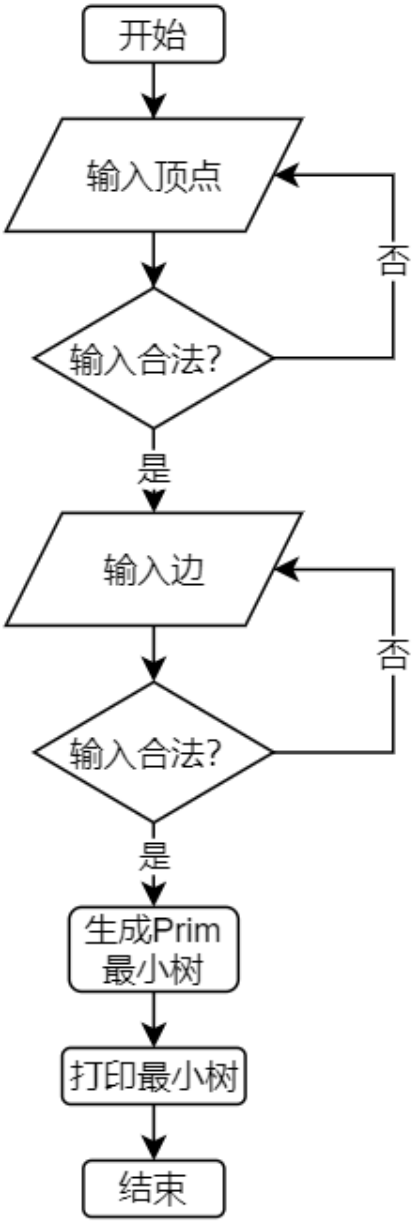
该函数用来产生最小生成树。

通过上述函数操作，即完成了定义了一个图类，各函数的详细实现方法见第 3 部分。

3 项目实现

3.1 项目主体功能

3.1.1 项目主体功能流程图

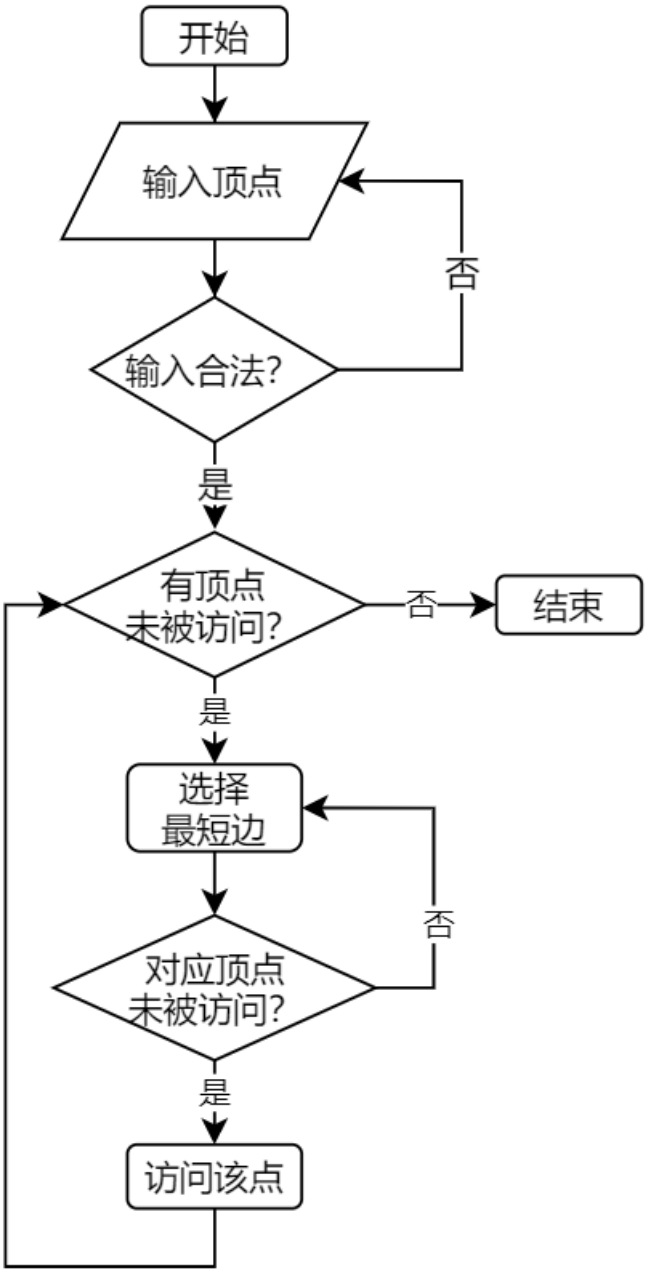


3.1.2 项目主体功能代码

```
int main()
{
    cout << "**\t\t 电网造价模拟系统\t\t**" << endl;
    cout << "=========================================" << endl;
    cout << "**\t\tA --- 创建电网顶点\t\t**" << endl;
    cout << "**\t\tB --- 添加电网的边\t\t**" << endl;
    cout << "**\t\tC --- 构建最小生成树\t\t**" << endl;
    cout << "**\t\tD --- 显示最小生成树\t\t**" << endl;
    cout << "**\t\tE --- 退出    程序\t\t**" << endl;
    cout << "=========================================" << endl;
    Graph g;bool quit = false;char op;
    while (!quit)
    {
        cout << "\n 请选择操作: ";
        cin >> op;
        switch (op)
    {
        case 'a':    case 'A':
            g.CreateVertex();
            break;
        case 'b':    case 'B':
            g.AddEdge();
            break;
        case 'c':    case 'C':
            g.BuildMST();
            break;
        case 'd':    case 'D':
            g.PrintMST();
            break;
        case 'e':    case 'E':
            quit = true;
            break;
        default:
            cout << "输入有误, 请重新输入! " << endl;
            cin.clear();
            cin.ignore(1024, '\n');
            break;
        }
    }
    return 0;
}
```

3.2 Prim 最小生成树

3.2.1 Prim 最小生成树流程图



3.2.2 Prim 最小生成树算法

```
void Graph::prim(int index)
{
    //删除之前生成的最小生成树
    mstPrim.clear();

    //标记是否访问过
    Vector<bool> visited(Vertex.size());
    visited.fillData(false);
    visited[index] = true;

    Vector<int> visitedPrim;
    visitedPrim.push_back(index);

    while (visitedPrim.size() < Vertex.size())
    {
        int start = -1, end = -1, min = MAX_LENGTH;
        for (auto i = visitedPrim.begin(); i != visitedPrim.end(); ++i)
        {
            for (auto j = adjacentList[(*i)].begin(); j != adjacentList
[(*i)].end(); ++j)
            {
                //尚未加入
                if (!visited[(*j).first] && (*j).second < min)
                {
                    start = *i;
                    end = (*j).first;
                    min = (*j).second;
                }
            }
        }
        //加入
        visited[end] = true;
        visitedPrim.push_back(end);
        mstPrim.push_back(Pair<int, int>(start, end));
    }
```

3.3 添加边

采用邻接表法存取的图，在添加表的时候不像用邻接矩阵的图一样可以直接更改邻接矩阵对应的值，而是要加入到对应的链表尾部。代码如下：

```
void Graph::AddEdge()
{
    string vertexA, vertexB;
    int edgeLength;
    while (true)
    {
        cout << "请输入两个顶点及边：";
        cin >> vertexA >> vertexB >> edgeLength;
        if (edgeLength <= 0)
            break;
        int indexA = findVertex(vertexA);
        int indexB = findVertex(vertexB);
        if (indexA == -1)
        {
            cout << "顶点 " << vertexA << " 不存在" << endl;
            continue;
        }
        if (indexB == -1)
        {
            cout << "顶点 " << vertexB << " 不存在" << endl;
            continue;
        }
        adjacentList[indexA].push_back(Pair<int, int>(indexB, edgeLengt
h));
        adjacentList[indexB].push_back(Pair<int, int>(indexA, edgeLengt
h));
    }
}
```

4 项目测试

4.1 创建顶点测试

测试用例：

A

4

a b c d

预期结果：

添加顶点

实验结果：

4.2 添加边测试

测试用例：

B

a b 8

b c 7

c d 5

d a 11

a c 18

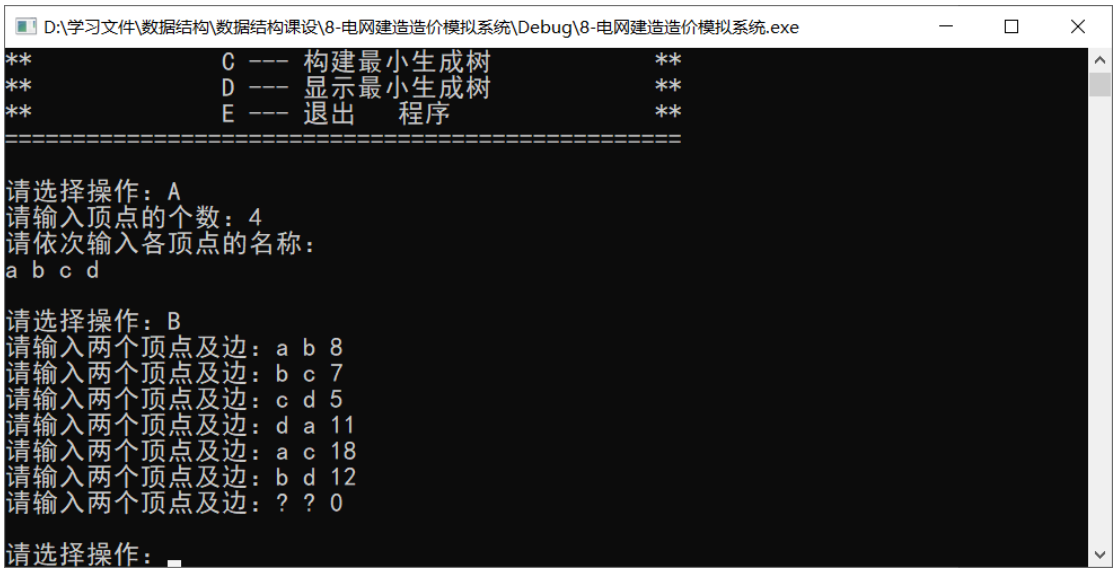
b d 12

? ? 0

预期结果：

添加边

实验结果：



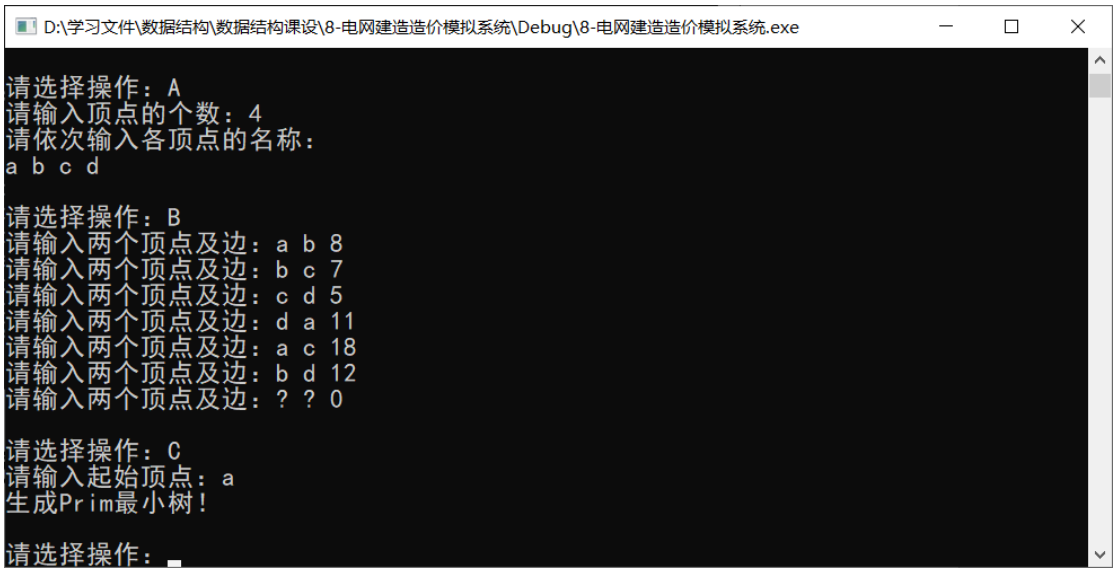
4.3 生成最小生成树测试

测试用例：

C a

预期结果：生成以 a 为起点的 Prim 最小生成树

实验结果：



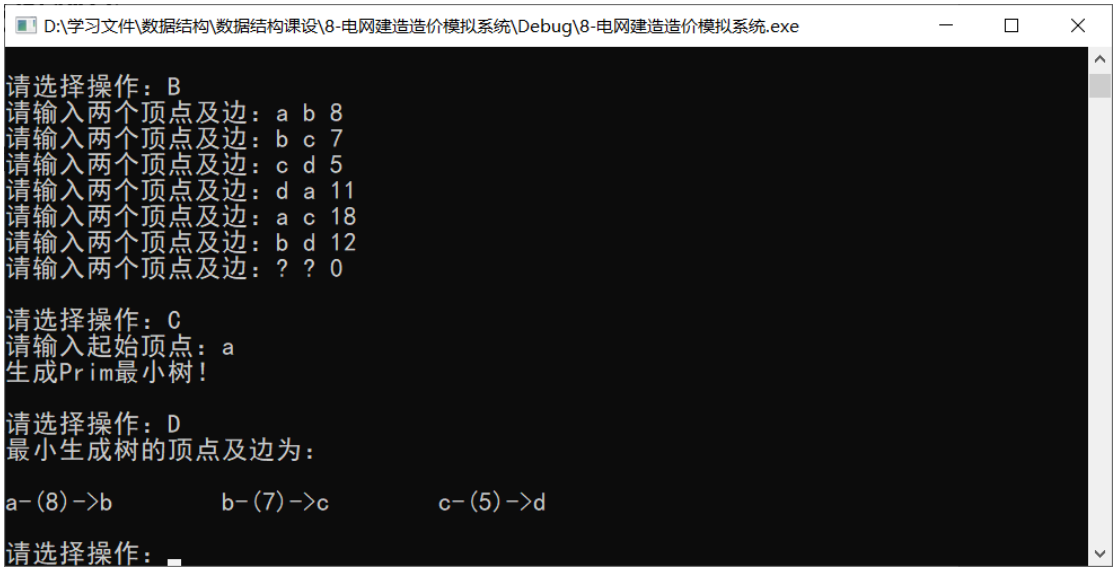
4.4 打印最小生成树测试

测试用例：

D

预期结果：打印之前以 a 为起点生成的 Prim 最小生成树

实验结果：

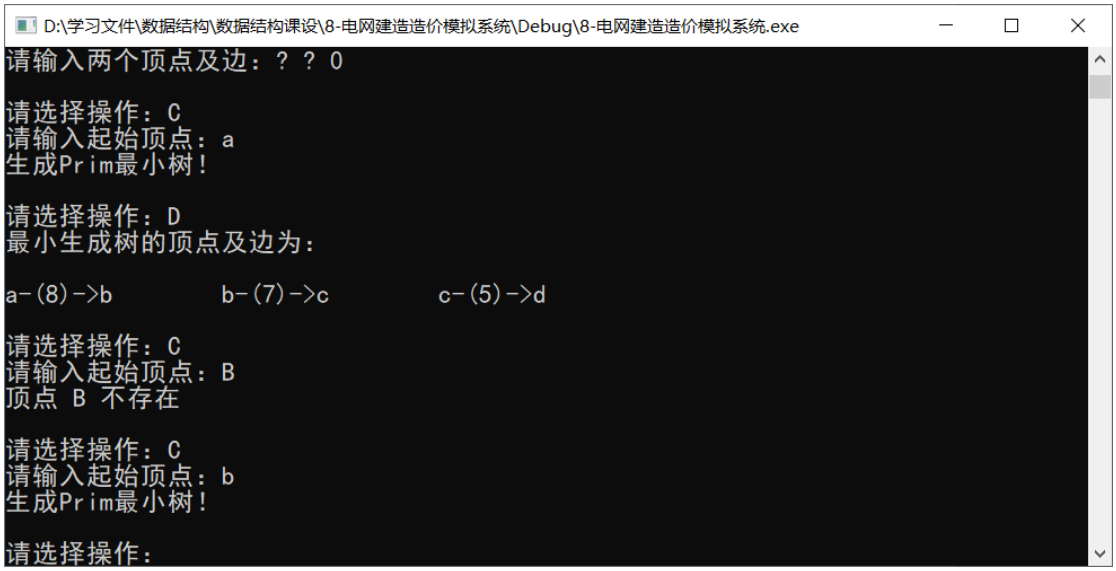


4.5 二次生成最小生成树测试

测试用例：C b

预期结果：生成以 b 为起点的 Prim 最小生成树

实验结果：

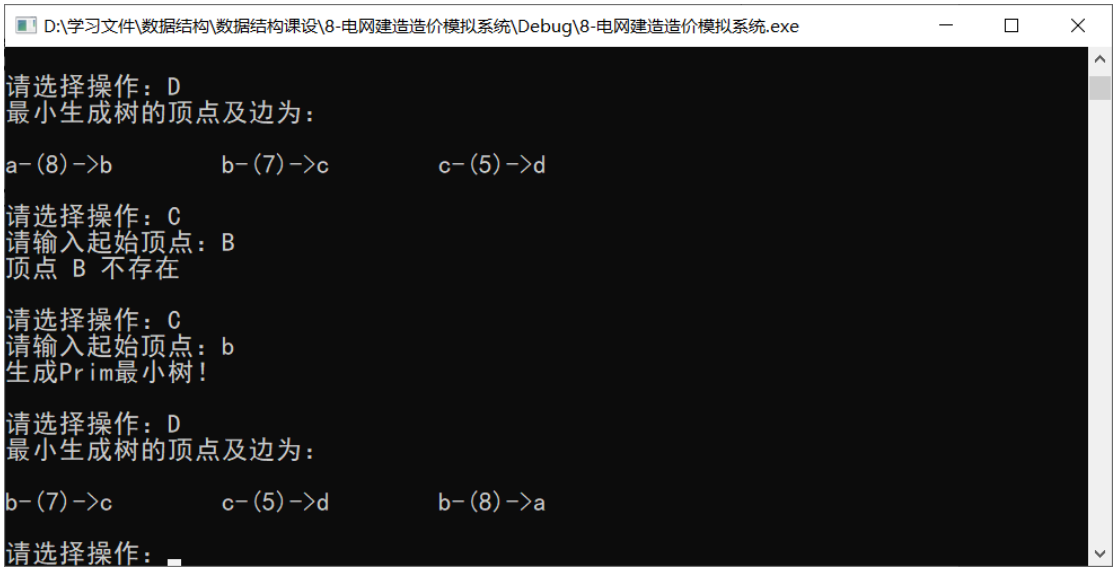


4.6 打印二次生成的最小生成树测试

测试用例：D

预期结果：打印之前以 b 为起点生成的 Prim 最小生成树

实验结果：



4.7 边界测试

4.7.1 只有两个顶点的最小生成树测试

测试用例：

A

2 a b

B

a b 10

? ? 0

C

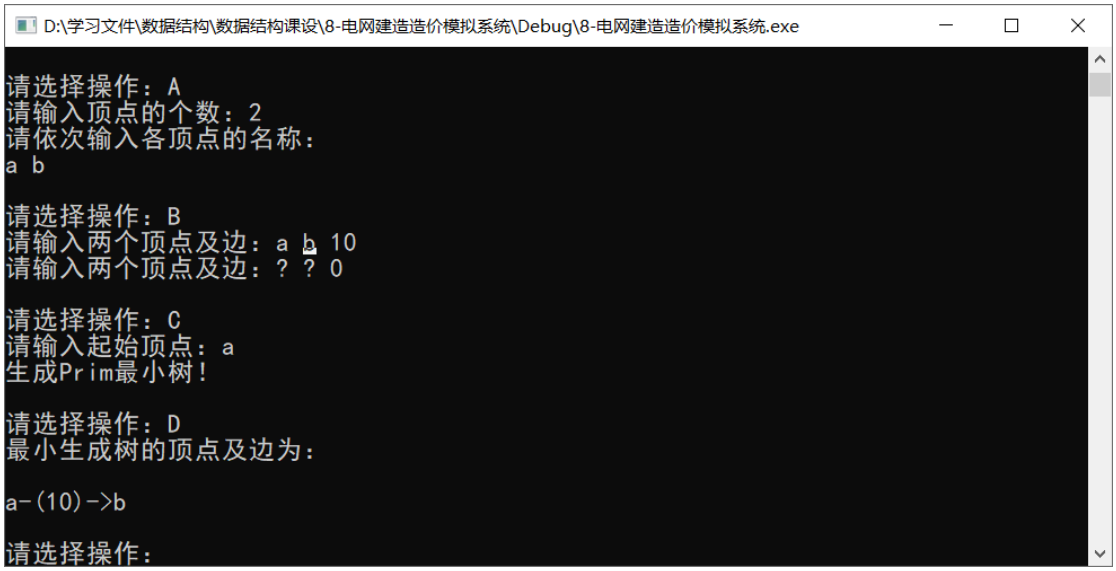
a

D

预期结果：

打印只含两个顶点的图中以 a 为起点生成的 Prim 最小生成树

实验结果：



4.7.2 只有一个顶点的最小生成树测试

测试用例：

A

1 a

C

预期结果：

打印只含一个顶点的图中以 a 为起点生成的 Prim 最小生成树

实验结果：