武汉大学计算机学院

实验报告

课程名称 《数据结构》

专业年级 2020

姓　　名 王俊儒

学　　号 2020301052272

实验学期 2022-2023 学年 第一 学期

课堂时数 32 课外时数 24

填写时间 2023 年 1 月 29 日

实验概述

【实验项目名称】：

武汉大学地图路径规划系统的 C 语言 AVL 树实现

【实验目的】：

考察学生综合使用所学的各种数据结构，并进行相应算法设计与实现的能力。

本实验的第一和第二部分要求实现并应用一个集合容器。集合容器需要提供高效的查询能力，因此在这里考察使用平衡二叉树实现容器类的能力。考虑到授课专业的实际情况，建议使用 AVL-tree 结构。在实现过程中，应灵活使用栈，队列，顺序表，链表等结构。

在此基础上，进一步锻炼在图结构上的算法设计和实现能力，重点在使用邻接表表示图，图上的最短路径算法和最小生成树算法的应用。为加深算法设计体验，对 prim 算法引入优先队列进行优化。

【实验环境】（使用的软件）：

Visual Studio 2022 Community

【参考资料】：

[1] Prim 算法的优化：邻接表、优先级队列（堆）优化(C 语言实现)
https://blog.csdn.net/wangeil007/article/details/107509143
[2] Prim 算法的 C 实现
https://zhuanlan.zhihu.com/p/72896700
[3] [最短路径问题]—Dijkstra 算法最详解
https://zhuanlan.zhihu.com/p/129373740
[4] 最小生成树 Prim 算法 +索引优先队列 - 掘金 (juejin.cn)
https://juejin.cn/post/7084218159586082852
[5] 基于 AVL 树表示的集合 ADT 实现与应用-数据结构课程设计报告
https://jz.docin.com/p-1492337118.html
[6] 迪杰斯特拉(Dijkstra)算法_小 C 哈哈哈的博客-CSDN 博客_缔结特斯拉算法
https://blog.csdn.net/xiaoxi_hahaha/article/details/110257368
[7] 图的存储结构之边集数组_酸菜。的博客-CSDN 博客_边集数组
https://blog.csdn.net/qq_38158479/article/details/104394341
[8] Prim 算法的优化：邻接表、优先级队列（堆）优化(C 语言实现)_wangeil007 的博客-CSDN 博客
https://blog.csdn.net/wangeil007/article/details/107509143

实验内容

【实验方案设计】：

一、实验思路

实验方案分为三个层次：（1）以二叉链表为存储结构，设计并实现 AVL-Tree 集合的插入、删除、查找、取长等功能，查找操作的时间复杂度为 O(log(n))，n 为集合中元素个数。（2）将地图信息以边集数组形式保存至集合容器中，提供路径查询功能，实现最短路径规划。（3）将地图信息以邻接表形式保存，用优先队列（堆）优化后的 prim 算法输出图的最小生成树，操作的时间复杂度为 O(nlog(n))，n 为顶点个数。

二、实验数据

实验地图选取武汉大学文理学部 30 个站点，38 条路径，20 多个地标建筑。为完成后续规划，需要将图像处理成数据存入结构体。根据题目要求，将校园地图抽象成加权无向网，采用边集数组和邻接表数组两种方法定义 graphE 和 graphL 两种结构，graphE 中的点集数组和边集数组都用 set 封装，graphL 用 adjlist 指针构建，同时还需要定义 numV 和 numE 两个常量记录图的定点数和边数，相关预输入数据在【实验结果】中呈现。

三、题目预览与分析

文中具体的概念与操作见【结构与函数定义】，此处仅作文字分析。

第一题，构造一个集合容器 Set，支持对集合的插入、删除、查找、取长、判断空集和清空集合功能，并且要求查找操作的时间复杂度 O(log(n))，n 为集合中元素的个数。

为了让集合更通用，需要定义模板参数 T 来满足不同类型的数据封装需求，随之而来的是 T 的运算法则，需要在相应的类型实例化时重载运算符，例如第二题封装图所需的 vertex

和 edge 结构体，在填入 set 之前需要预定义大于、小于、等于符号，才能正常完成集合中节点的排序。

同时，构建二叉搜索树可以满足时间复杂度 O(log(n))的需求，但是序列有序时二叉搜索树退化成单链表，搜索效率退化为 O（N)。为了解决这个问题我们引入了 AVL 树，又称平衡搜索二叉树。实际上，通过树节点的相对次序保留了前一次循环的结果，将普通搜索的 O(n^2)下降到 O(log(n))，但相应地提升了空间复杂度，占用了更多的堆栈空间。

第二题，使用构造的集合容器保存地图中标出的路口结点信息（路口编号）和直接连接两点的路径信息（路径端点和长度）。

为什么不用数组存储？数组优点：查询快，通过索引直接查找；有序添加，添加速度快，允许重复； 缺点：在中间部位添加、删除比较复杂，大小固定，只能存储一种类型的数据；如果应用需要快速访问数据，很少插入和删除元素，就应该用数组。**C 语言中的数组是静态的，预定义长度后不能改变。**显然，为达成实验目标，我们需要对图中节点进行多次插入和删除，需要多次定义新数组，造成性能和空间的浪费。

此处我们用第一题构建的集合分别封装点 vertex 和边 edge，注意这两种 T 的实例化需要**重载运算符**。

第三题，需要基于第二题的存储结构，设计方案提供路径查询功能，针对表中的任意两个地点，计算最短路径并输出。因为方案中给出的地点并没有以点的形式呈现，无法直接定位和求距离，必须通过该地附近的多个点进行模糊定位。

3、 设计方案，提供路径查询功能。编写程序，输入表中的两个地点 A 和 B，计算一条从 A 到 B 的最短的路径，并输出。

行政单位					
1	教一楼	6	计算机学院	11	艺术学院
2	教二楼	7	生命科学院	12	行政大楼
3	教三楼	8	化学学院	13	老图书馆
4	教四楼	9	物理学院	14	新图书馆
5	理学院	10	经管学院	15	卓尔体育馆

本题可以理解为乘公交问题。给定出发地和目的地，以及附近的多个车站，需要求出最短的一条路径，从车站乘坐公交车到达另一车站，有多条路径，需要多种情况并行讨论。以 1（教二楼）到 8（化学学院）为例，教二楼附近有 17，18 两个车站，化学学院附近有 6，7，8，28 四个车站，需要一一配对求出各自的最短距离，并且从中选出一条最短的距离作为结果输出。

此时问题简化为两点之间的最短路径问题，迪杰斯特拉算法使用类似广度优先搜索的方法解决了带权图的单源最短路径问题，符合题目需求。

第四题，现在要在地图中的所有路口安装交通灯和监控系统，系统由光纤连接，光纤沿道路铺设，请设计最短的铺设方案。

实际上是求权重图的最小生成树，即遍历所有节点的路径长度和最短，可用 prim 算法。Prim 算法与 dijkstra 算法相似，都是基于贪心算法，都适用于稠密图，时间复杂度都是 O（V^2），都可以用堆进行优化，其时间复杂度都与边无关。

观察可知，地图中的结点之间由不太多的道路连接，这种图称为稀疏图，这样的图适宜使用邻接表而非邻接矩阵存储。因此我们需要重新定义邻接表来存储地图信息。

此外，prim 算法如果用优先队列优化，时间复杂度为 O(n(log(n))),n 为边数。优先队列可用最小堆构建，堆顶元素最小，每次取用堆顶元素加入最短路径的点集，因此还需要实现堆的一系列定义与操作。

三、代码库导入

```
#include<stdio.h>
#include<stdlib.h>
#include<algorithm>
#include<string>
#include<vector>
#pragma warning(disable:4996)
using namespace std;
```

四、结构与函数定义

本方案一共实现了四种数据结构，并且定义了满足题目需求的结构体函数。以下**基于代码顺序介绍**并引入概念，所有代码均可直接运行。

第一题，基于 AVL 树实现的集合容器 setnode 和 set。

模板参数 T

Setnode<T>和 set<T>中的 T 是一种模板参数。模板是 C++支持参数化多态的工具，使用模板可以使用户为类或者函数声明一种一般模式，使得类中的某些数据成员或者成员函数的参数、返回值取得任意类型。使用模板的目的就是能够让程序员编写与类型无关的代码。比如编写了一个交换两个整型 int 类型的 swap 函数，这个函数就只能实现 int 型，对 double，字符这些类型无法实现，要实现这些类型的交换就要重新编写另一个 swap 函数。使用模板的目的就是要让这程序的实现与类型无关，比如一个 swap 模板函数，即可以实现 int 型，又可以实现 double 型的交换。此处集合使用模板参数可以实现容器的通用，满足后续实例化多种节点类型的数据结构的需求。

首先实现 AVL 树。

AVL 树是一种平衡二叉树，平衡二叉树递归定义如下：

1.左右子树的高度差小于等于 1。

2.其每一个子树均为平衡二叉树。

为了保证二叉树的平衡， AVL 树引入了所谓监督机制，就是在树的某一部分的不平衡度超过一个阈值后触发相应的平衡操作。保证树的平衡度在可以接受的范围内。AVL 树就是所有结点的平衡因子的绝对值都不超过 1 的二叉树。

然后实现 setnode 和 set.

节点由本身存储的值 value，平衡因子 bf，指向左子树和右子树节点的指针 lchild，rchild，并定义初始化函数 setnode()。

```
template<typename T>
class setnode {
public:
     setnode<T>* lchild;
     setnode<T>* rchild;

     T value;
     int bf;

     setnode() {
          this->lchild = nullptr;
          this->rchild = nullptr;
          this->bf = 0;
     }

};//这里 setnode 加入 parent 指针可以避免后续函数内循环，更简洁，可以尝试改进
```

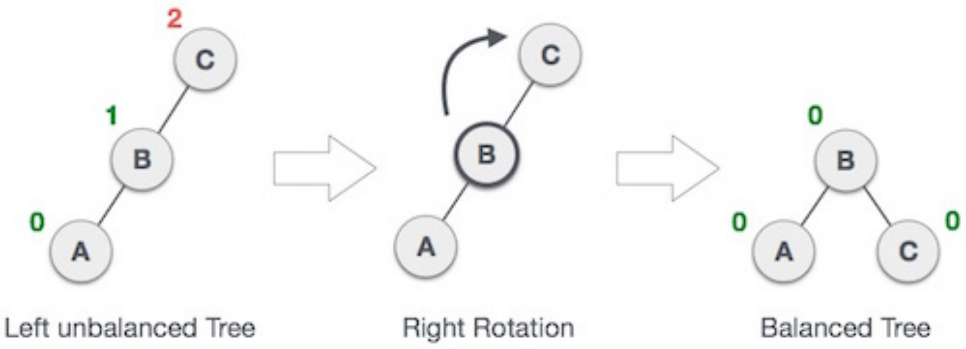
Class set 实现了 setnode 的集合，定义了一个 setnode 类型的头节点 Node，此时的模板参数 T 是 setnode 内数据的类型。结构体内定义函数实现集合的特定功能，定义 treeHeight 用递归方式计算当前节点所处的树高度，balancefactor 计算当前节点的平衡因子。

```
 template<typename T>//加入比较原则
class set {
     typedef setnode<T> Node;

public:

     Node* p = nullptr;

     int treeHeight(Node* p) {//计算当前节点树高度
          if (p == NULL) {
               return 0;
          }
          else if (treeHeight(p->lchild) > treeHeight(p->rchild))
               return treeHeight(p->lchild) + 1;
          else
               return treeHeight(p->rchild) + 1;

     }

     int BalanceFactor(Node* p) {//计算并赋值当前节点平衡因子
          if (p == NULL) {
               return 0;
```

```
        }

        else
        {
            p->bf = treeHeight(p->lchild) - treeHeight(p->rchild);
            return p->bf;
        }
    }
```

AVL 树需要严格所有节点控制平衡因子（bf）的绝对值不超过 1，因此在插入和删除节点时，需要沿插入或删除的途径依次更新节点的 treeHeight。二叉树的平衡化有左旋和右旋两大操作，即逆时针和顺时针旋转。

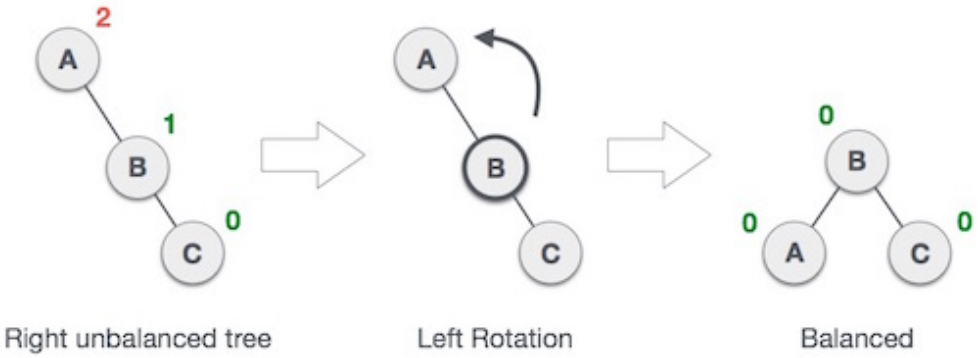
右旋操作



右旋操作就是将图中 B 和 C 节点进行交换。当 B 节点存在 rightchild 时，抛弃 rightchild，和旋转后的 C 节点相连，成为节点左孩子。

```
    Void R_Rotate(Node*& p) {//右旋
        Node* lc = p->lchild;
        p->lchild = lc->rchild;
        lc->rchild = p;

        BalanceFactor(p);
        BalanceFactor(lc);//重新计算节点平衡因子

        p = lc;
        //调整新的头节点
    }
```

左旋操作



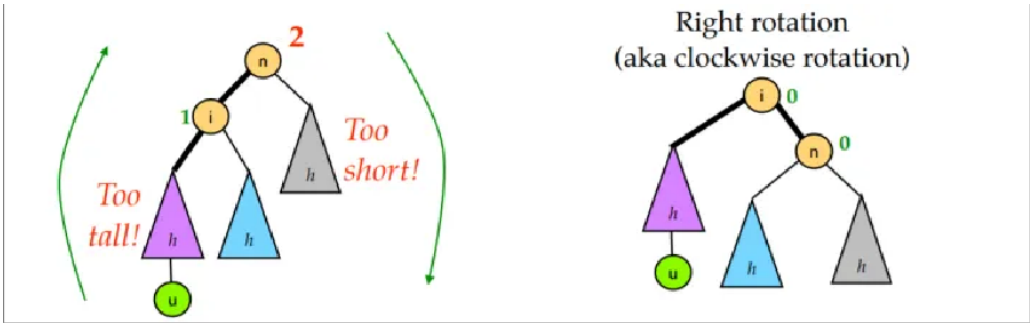
左旋操作与右旋操作对称。

```
void L_Rotate(Node*& p) {//左旋
    Node* rc = p->rchild;
    p->rchild = rc->lchild;
    rc->lchild = p;


    BalanceFactor(p);
    BalanceFactor(rc);//重新计算节点平衡因子


    p = rc;
}
```

平衡操作

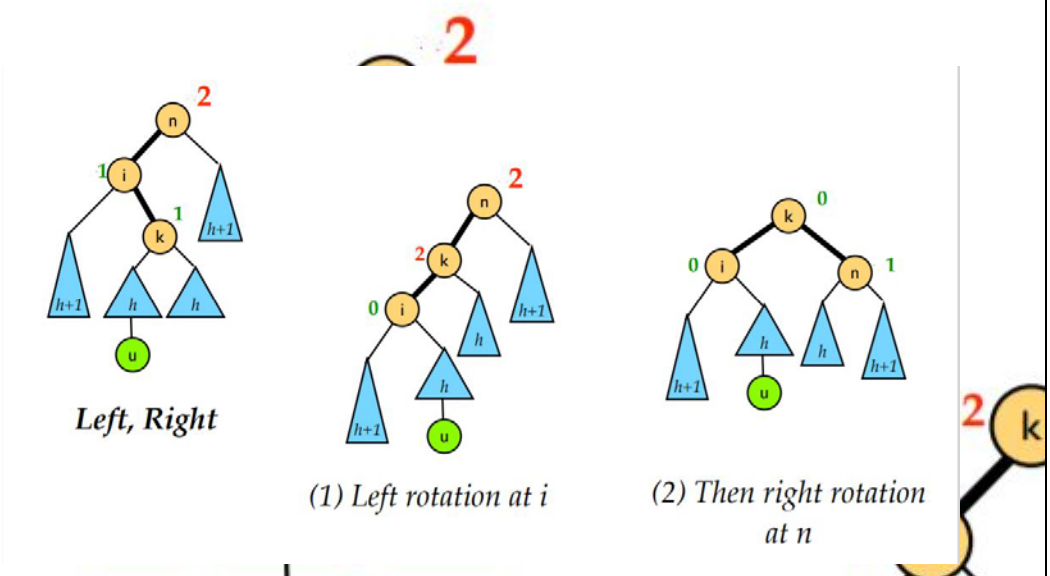
二叉平衡树可能有多种破坏平衡的情况，并有对应的平衡方式。

（1）Left-Left 型



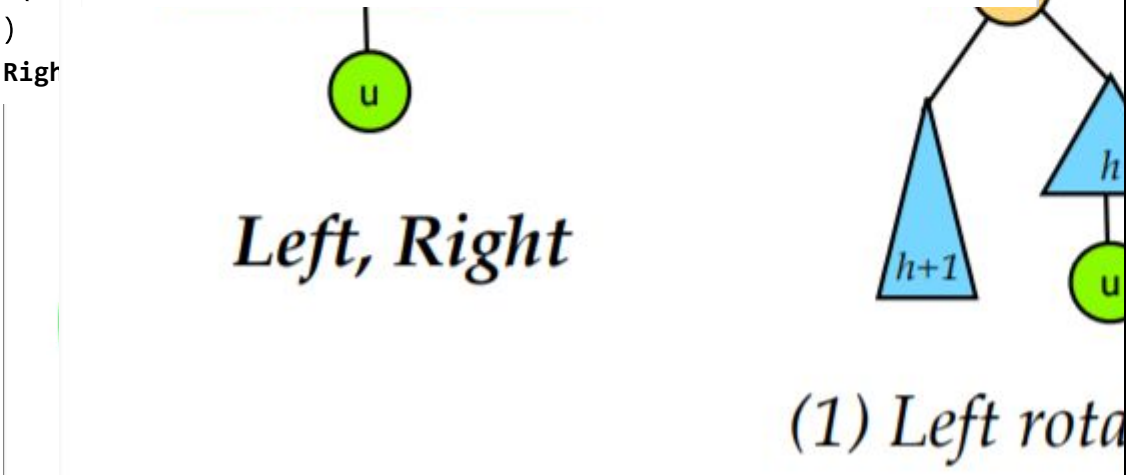
根节点的平衡因子为+2，对节点 n 右旋一次即可。对应代码中的 `if(factor<-1 && BalanceFactor(p->rchild)<=0)`情况。

(2)



(3)

Righ



(1) Left rota

(4) **Right-Left 型**

与 Left-Right 对称，一次右旋一次左旋即可。

```
Node* Rebalance(Node*& p) {//平衡
    int factor = 0;

    if(p!=nullptr)
        factor = p->bf;

    //分四种情况讨论
    if (factor > 1 && BalanceFactor(p->lchild) >= 0)
        //LL,需要 1 次左旋
        L_Rotate(p);
    else if (factor > 1 && BalanceFactor(p->lchild) < 0) {
        //LR,1 次左旋+1 次右旋
```

```
            L_Rotate(p);
            R_Rotate(p);
        }
        else if (factor < -1 && BalanceFactor(p->rchild) <= 0) {
            //RR,1 次右旋
            R_Rotate(p);
        }
        else if (factor < -1 && BalanceFactor(p->rchild)>0) {
            //RL,1 次右旋+1 次左旋
            R_Rotate(p);
            L_Rotate(p);
        }
        //已经平衡，不需要操作
        return p;
    }
    //AVL 树已构成，接下来完成集合构建
```

集合构建

AVL 树已建成，接下来进行集合功能的实现：

（1）向集合中中插入一个元素。将类型 T 的常值参数 v 放入集合，并返回其地址，如失败，返回 NULL.通过对左右节点的遍历找到合适的节点之间，插入一个节点。

```
    Node* emplace(Node*& p, const T& v) {
        Node* root = p;

        if (root == nullptr) {
            Node* newNode = new Node;
            if (newNode == nullptr)
                return nullptr;

            newNode->value = v;

            root = newNode;
            if (p == nullptr) {
                p = newNode;
            }

            return newNode;
        }

        else {

            if (root->value == v) {
                return nullptr;
```

```
            }
            else {
                if (root->value < v)
                    emplace(root->rchild, v);
                else
                    emplace(root->lchild, v);
            }

        }
        Rebalance(p);
        return nullptr;
    }
```

（2）从集合中中删除一个元素。将以参数 value 为关键字的对象从集合中删除，删除成功返回 true，删除失败，返回 false。

```
    bool erase(Node*& p, const T& value) {
        Node* byebye;
        Node* root = p;

        if (root != nullptr) {
            if (root->value==value) {
                if (root->rchild) {
                    erase(root->rchild, value);
                }
                else {
                    byebye = &root;
                    p = byebye->lchild;
                    free(byebye);
                }
            }
            else {
                if (root->value<value) {
                    erase(root->rchild, value);
                }
                else
                    erase(root->lchild, value);
            }
            Rebalance(&root);
            return true;
        }
```

```
        else
            return false;
    }
```

（3）在集合中查找特定值元素。查找集合中对象的 v 等于参数 v 的对象，返 回其地址，如失败，返回 NULL。因为集合的模板参数 T 不确定，在本次实验中有 edge、vertex 类型的实例化，**需要在各自的结构体定义中重载运算符，才能正常运行 find 函数。**

```
    T* find(const T& v) {
        if (p == nullptr)
            return nullptr;

        Node* cur = p;
        while (cur) {
            if (cur->value>v)
                cur = cur->lchild;
            else if (cur->value<v)
                cur = cur->rchild;
            else
                return &(cur->value);
        }
        return nullptr;
    }
```

（4）获得集合中元素个数。返回容器中元素的个数。

```
    int size(Node* p, int count) {
        if (*p != NULL) {
            if ((*p)->lchild == NULL && (*p)->rchild == NULL) {
                count++;
            }
            size(&((*p)->lchild), count);
            size(&((*p)->rchild), count);
        }
        return count;
    }
```

（5）判断集合是否为空。

```
    bool empty() {
        if (size() == 0)
            return true;
        else
            return false;
    }
```

（6）清除集合中所有元素，size()返回 0.

```
    void clear() {
        if (*p == nullptr) {
            free(p);
            p = nullptr;
            return;
        }

        clear((*p)->lchild);
        clear((*p)->rchild);
        free(*p);
        *p = nullptr;
    }
};
```

第二题，基于集合容器保存武汉大学地图中的路口节点信息（路口编号）和直接链接两点的路径信息（路径端点和长度）。

分析可知，边集数组适用于此题，因此构建 vertex 和 edge 两个结构体分别存储点和边，并通过 set<vertex>和 set<edge>构造一个 graphE 结构。

边集数组

边集数组是由两个一维数组构成。一个是存储顶点的信息；另一个是存储边的信息。这个边数组每个数据元素由一条边的起点下标（begin）、终点下标（end）和权（weight）组成。边集数组关注的是边的集合，在边集数组中要查找一个顶点的度需要扫描整个边数组，效率并不高。因此它更适合对边依次进行处理的操作，而不适合对顶点相关的操作。

begin	end	weight

其中 **begin** 是存储起点下标，**end** 是存储终点下标，**weight** 是存储权值。

```
//边集数组构建图（集合），完成第二题的图储存
class graphE {
    struct edge {
        int begin;//起点下标
        int end;//终点下标
        int weight;

        edge() { begin = 0; end = 0; weight = 0; }

        bool operator == (const edge &e) const {
            if ((this->begin == e.begin) &&
                (this->end == e.end)) {
```

```
                return true;
            }
            else
                return false;
        }

        bool operator > (const edge &e) const {
            if (this->begin > e.begin) {
                return true;
            }
            else if ((this->begin == e.begin) &&
                (this->end > e.end)) {
                return true;
            }
            else
                return false;
        }

        bool operator < (const edge& e) const {

            if (this->begin < e.begin) {
                return true;
            }
            else if ((this->begin == e.begin) &&
                (this->end < e.end)) {
                return true;
            }
            else
                return false;
        }

        //需要解决 edge 在 set 中的排序问题，重载 operator
        //就是重新定义比大小的方式
    };

    struct vertex {
        int point;

        vertex() { point = 0; };


        bool operator > (const vertex & p1) const {
            return point > p1.point;
        }
```

```
        bool operator < (const vertex& p1) const {
            return point < p1.point;
        }

        bool operator == (const vertex& p1) const {
            return point == p1.point;
        }
    };

    set<vertex> V;//点集
    set<edge> E;//边集

    int numV;//点数
    int numE;//边数


public:
    graphE() {
        V.p = nullptr;
        E.p = nullptr;
        numV = 0;
        numE = 0;

    }//初始化函数

    void CreateGraph_Edge(graphE* g) {
        int i = 0, j = 0, k = 0;
        edge w{};
        vertex po{};

        printf("请输入顶点数和边数:\n");
        scanf("%d,%d", &g->numV, &g->numE);
        getchar();//获取回车符

        //点数组
        for (i = 1; i <= g->numV; i++) {
            po.point = i;
            g->V.emplace(g->V.p, po);

        }

        //邻边矩阵
        for (k = 1; k <= g->numE; k++) {
```

```
            printf("依次输入第%d 条边<vi,vj>的下标 i,j 和权重 w:\n",k);
            scanf("%d,%d,%d", &w.begin, &w.end, &w.weight);
            getchar();

            g->E.emplace(g->E.p, w);
            //因为无向图，所以一条边要调换头尾再储存一遍
            int temp = w.begin;
            w.begin = w.end;
            w.end = temp;
            g->E.emplace(g->E.p, w);

        }


    }

    //展示边集合专用,在下一个函数调用
    void showedge(setnode<edge>* root) {
        if (root != nullptr) {
            printf("(%d)->(%d)<%d>\n  ", root->value.begin, root->value.end,
root->value.weight);
        }
        else
            return;

        showedge(root->lchild);
        showedge(root->rchild);
    }
    //展示点集合专用,在下一个函数调用
    void showvertex(setnode<vertex>* root) {
        if (root != nullptr) {
            printf("%d ",root->value.point);
        }
        else
            return;
        showvertex(root->lchild);
        showvertex(root->rchild);
    }

    void ShowGraph_Edge(graphE* g) {

        //展示点集合
        setnode<vertex>* rootv = g->V.p;
        printf("点集合:\n[");
```

```
        showvertex(rootv);
        printf("]\n");

        //展示边集合
        setnode<edge>* roote = g->E.p;
        printf("边集合:\n[");
        showedge(roote);
        printf("]\n");
    }
```

GraphE 内嵌套 shortestway 函数用于寻找任意两点的最短路径。实现原理为迪杰斯特拉算法。

Dijkstra 算法

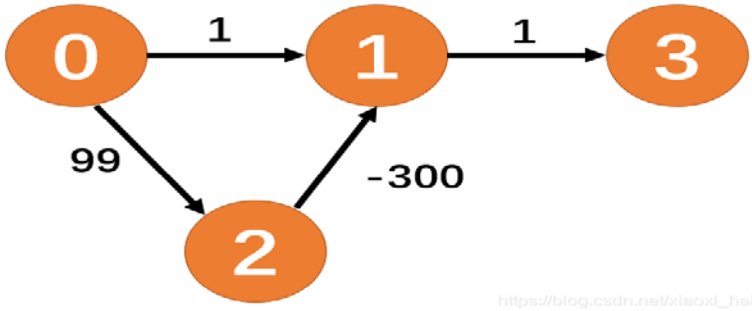
迪杰斯特拉算法是由荷兰计算机科学家在 1956 年发现的算法，此算法使用类似广度优先搜索的方法解决了带权图的单源最短路径问题。它是一个贪心算法。

核心思想：

1. 选定一个点，这个点满足两个条件：1.未被选过，2.距离最短
2. 对于这个点的所有邻近点去尝试松弛

注意事项：

1. 迪杰斯特拉算法无法处理负权边的图。因为再加入集合的过程中贪心算法无法实现最优解。以此图为例，按照 Dijkstra 的思想，首先 0 点加入集合，然后 1 点加入集合，然后 3 点加入集合，然后 2 加入集合，此时 0->2 更新为 99-300，但是此刻已经无法更新 0->3（原来的值为 2）了。



2. 只能解决单源最短路径问题。

伪代码如下：

Algorithm : Dijkstra

Input : Directed graph $G = (V, E, W)$ with weight

Output : All the shortest paths from the source vertex s to every

$1: S \leftarrow \{s\}$
$2: \text{dist}[s, s] \leftarrow 0$
$3:$ **for** $v_i \in V - \{s\}$ **do**
$4:$ $\quad \text{dist}[s, v_i] \leftarrow w(s, v_i)$
$\quad$ (when v_i not found, $\text{dist}[s, v_i] \leftarrow \infty$)
$5:$ **while** $V - S \neq \varnothing$ **do**
$6:$ $\quad$ find $\min_{v_j \in V} \text{dist}[s, v_i]$ from the set $V - S$
$7:$ $\quad S \leftarrow S \cup \{v_j\}$
$8:$ $\quad$ **for** $v_i \in V - S$ **do**
$9:$ $\quad\quad$ **if** $\text{dist}[s, v_j] + w_{j,i} < \text{dist}[s, v_i]$ **then**
$10:$ $\quad\quad\quad \text{dist}[s, v_i] \leftarrow \text{dist}[s, v_j] + w_{j,i}$

```
//两点最短路径
    int shortestway(int start, int des, graphE* g) {
    //计算两端点最短路径(迪杰斯特拉算法)
    //要不断寻找没有被访问过且离源点最近的点

        const int inf = 100000;//不可到达的距离！
        int n = g->numV;//图的顶点个数
        vector<int> dis(n, inf);//源点到其他点的最短距离，inf 表示无法到达
        vector<bool> vis(n, false);//访问记录，false 表示未访问
        vector<int> pre(n, -1);//记录这个点在加入时连接的点（需要重点理解）
        edge e{};
```

函数输入 start，des 分别表示起点和终点，此处以 start 为源点。dis 数组表示源点到其他点的最短距离，inf 为预定义的一个足够大的数，表示无法到达。Vis 数组记录各点的访问情况。Pre 数组记录各点在加入已生成集合时所连接的点，有两种可能，直接与源点连接，或者通过和集合中已有的某点相连，从而间接连接源点。完成遍历后，借助 pre 数组实现对路径的回溯输出。

```
        dis[start] = 0;//源点到源点的距离为 0

        for (int i = 0; i < n - 1; i++) {
            int node = -1;
```

```
            for (int j = 0; j < n; j++) {
                if (!vis[j] && (node == -1 || dis[j] < dis[node])) {
                    node = j;
                }
            }
             // 标记访问
```

第一个 for 循环遍历除源点外 n-1 个点，每次完成对一个点（即代码中的的 dis 整理。第二个 for 循环开始寻找 n-1 个点中没有被访问过且到源点距离最短的点，标记为 node 并在访问数组 vis 中标记为 true，第三个 for 循环考虑引入 node 后各点到源点的距离是否变小，因为存在间接通过 node 到达源点的可能性。其中，if 条件如果判断出（从 j 到选点 node 的距离）+（node 到源点距离）<（j 到源点的距离），则对 dis[j]重新赋值，并标记 j 的连接点为 node，即 pre[j]=node。

```
        for (int j = 0; j < n; j++) {
                e.begin = node+1;
                e.end = j+1; //end 和 begin 都是从 1 开始，函数中的点都是从 0 开始
                if (g->E.find(e) != nullptr) {
                    if (dis[j] > dis[node] + g->E.find(e)->weight) {
                        dis[j] = dis[node] + g->E.find(e)->weight;
                        pre[j] = node;
                    }
                }
            }

            vis[node] = true;
        }
```

此时函数预输入的 start、des 发挥作用，从 des 开始，不断通过 pre[i]回溯，直到 start，打印出一条完整的最短路径。此时需注意数组的[0]项加一等于编号。

```
        //打印最短路径
        printf("从%d 到%d 路线:\n",start+1,des+1);
        /*要反过来展示，从 A 到 B 的路径展示，前面需要以 A 为源点，一步步倒推到 B，
        然后此时通过 pre 进行回溯
        展示出来就是正向的*/
        for (int i = des; i != start; i = pre[i]) {
            printf("%d -> ", i+1);
        }
        printf("%d,总距离%d\n", start+1,dis[des]);

        return dis[des];
    }
};
```

第三题，需要基于第二题的存储结构，设计方案提供路径查询功能，针对表中的任意两个地点，计算最短路径并输出。因为方案中给出的地点并没有以点的形式呈现，无法直接定位和求距离，必须通过该地附近的多个点进行模糊定位。

3、 设计方案，提供路径查询功能。编写程序，输入表中的两个地点 A 和 B，计算一条从 A 到 B 的最短的路径，并输出。

行政单位					
1	教一楼	6	计算机学院	11	艺术学院
2	教二楼	7	生命科学院	12	行政大楼
3	教三楼	8	化学学院	13	老图书馆
4	教四楼	9	物理学院	14	新图书馆
5	理学院	10	经管学院	15	卓尔体育馆

本题可以理解为乘公交问题。给定出发地和目的地，以及附近的多个车站，需要求出最短的一条路径，从车站乘坐公交车到达另一车站，有多条路径，需要多种情况并行讨论。以 1（教二楼）到 8（化学学院）为例，教二楼附近有 17，18 两个车站，化学学院附近有 6，7，8，28 四个车站，需要一一配对求出各自的最短距离，并且从中选出一条最短的距离作为结果输出。

各地车站分布情况如下：

```
/* 1 号：教一楼，10;
       2 号：教二楼，17，18
       3 号：教三楼，27
       4 号：教四楼，5，9
       5 号：理学院，19，20
       6 号：计算机学院，12，13，14
       7 号：生命科学院，1
       8 号：化学学院，6,7,8,28
       9 号：物理学院,25
       10 号：经管学院，22
       11 号：艺术学院，11，19
       12 号：行政大楼，10，11
       13 号：老图书馆，29
       14 号：新图书馆，5
       15 号：卓尔体育馆，7，12*/
```

此处定义 place 结构体，用于存储中文地名和相对应的多个车站编号。

```
//第三题
typedef struct Place {
    char name[10];//储存地理位置的名称
    int station[10];//储存附近站点
    int length;//记录站点数量
```

```
    Place() { length = 0; }
}Place;
```

同时为了实现多条最短路径的规划，还需定义与 place 绑定的路线规划函数 pathpalnning。通过双层 for 循环求出几条最短路径中的最小值并输出。此处用到 place 和 graphE 结构体。

```
//路线规划函数
void pathplanning(Place* place, graphE* ge) {
    int A = 0, B = 0;
    int p1 = 0, p2 = 0;
    printf("请输入起点和终点的序号:\n");
    scanf("%d,%d", &A, &B);
    int shortdis = 100000;
    for (int i = 0; i < place[A-1].length; i++) {
        for (int j = 0; j < place[B-1].length; j++) {
            int d = ge->shortestway(place[A-1].station[i]-1, place[B-1].station[j]-1, ge);
            if (shortdis > d) {
                shortdis = d;
                p1 = place[A - 1].station[i] - 1;
                p2 = place[B - 1].station[j] - 1;

            }
        }
    }
    printf("最短路径如下:\n");
    ge->shortestway(p1, p2, ge);
   }
```

第四题，要求在所有路口安装由道路光纤铺设的交通灯和监控系统，最短的铺设方案实际是求该图的最短生成树.首先用邻接表存储图信息，然后用堆优化的 prim 算法求取最短生成树。

首先构建堆，实现堆的初始化、交换序号、插入、删除和调整功能。

堆(Heap)

堆通常是一个可以被看做一棵完全二叉树的数组对象。

堆满足下列性质：堆中某个节点的值总是不大于或不小于其父节点的值，堆总是一棵完全二叉树。

本方案用数组构建堆，Node 节点存储了点到生成树的距离数组 dis，记录节点编号在堆中位置的数组 pos，index 记录了堆节点编号对应的顶点编号。Size 和 count 分别记录了已经载入的节点数和最大容量。所有的数组下标均从 1 开始，因为下标为 0 的数组无法参与运算，因此数组规模等于最大容量加一。

堆的插入将新的值返回数组的尾部，从下至上的调整堆节点，删除操作将堆顶点输出并

删除，再至上而下修复堆属性。Fix 函数用于调整堆中某一结点的值。我们只需要维持一个小根堆，保证堆的顶点最小。因此只需定义 fix_up 函数，能减小堆节点 dist 的值并重新调整位置。Prim 算法实际上就是不断迭代节点离生成树的距离达到最小值。

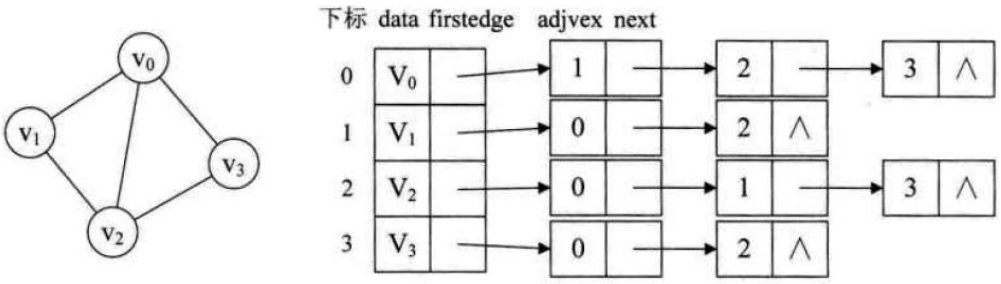
```
/*以下是第四题代码*/
//优化堆结构
class IndexHeap {
    struct Node {
        int* dist;//顶点到生成树的最小距离
        int* pos;//顶点在堆中的位置
    };

public:
    int* index;//堆节点对应的顶点编号
    Node* ass;
    int size;
    int count;

    IndexHeap(int maxsize) {
        int inf = 100000;//无法到达
        ass = new Node;
        count = 0;
        size = maxsize;
        index = new int[maxsize + 1]();
        index[0] = 0;//哨兵
        ass->dist = new int[maxsize+1]();
        ass->pos = new int[maxsize+1]();
        for (int i = 0; i <= size; i++) {
            ass->dist[i] = inf;
        }


    }

    void SwapIndex(IndexHeap* heap, int i) {
        heap->ass->pos[heap->index[i]] = i;
    }//i 是顶点在堆中的位置，index[i]是顶点编号

    //插入顶点编号为 v 的顶点,插入一个 index
    void InsertIndex(IndexHeap* heap, int v) {
        int x = heap->ass->dist[v];
        int child;
        for (child = ++heap->count;
```

```
            x < heap->ass->dist[heap->index[child / 2]] &&child>0;
            child /= 2) {
            heap->index[child] = heap->index[child / 2];
            SwapIndex(heap, child);//child 是 v 顶点在堆中的位置
        }
        heap->index[child] = v;
        SwapIndex(heap, child);
    }

    //删除并返回最小边的索引
    int DeleteIndex(IndexHeap* heap) {
        int v = heap->index[1];
        int x = heap->ass->dist[heap->index[heap->count--]];
        int parent, child ;

        for (parent = 1; parent * 2 <= heap->count ; parent = child) {
            child = parent * 2;
            if (heap->ass->dist[heap->index[child + 1]] <
                heap->ass->dist[heap->index[child]] &&
                child != heap->count) {
                child++;
            }

            if (x > heap->ass->dist[heap->index[child]]) {
                heap->index[parent] = heap->index[child];
                SwapIndex(heap, parent);
            }
            else
                break;
        }
        heap->index[parent] = heap->index[heap->count + 1];
        SwapIndex(heap, parent);
        return v;
    }

    //修改 dist[v]减少至 x，并调整堆
    void Fixup(IndexHeap* heap, int v, int x) {
        heap->ass->dist[v] = x;
        int child;

        //使用 pos 访问堆而不是 v

        for (child = heap->ass->pos[v];
            x < heap->ass->dist[heap->index[child / 2]] &&child>0;
```

```
            child /= 2) {
            heap->index[child] = heap->index[child / 2];
            SwapIndex(heap, child);
        }
        heap->index[child] = v;
        SwapIndex(heap, child);
    }

};
```

邻接表

我们发现，当图中的边数相对于顶点较少时，邻接矩阵是对存储空间的极大浪费。如果用 Enode 数组存储结点，用 adjlist 链表直接存储结点相邻的节点。每个结点数组有一个指向链表的指针 firstedge.



代码完成了 graphL 的初始化、创建和展示，并且建立 prim_optim 完成最小生成树。

```
//邻接表构建图（数组）
class graphL {
    typedef struct Enode {//边表节点
        int adjvex;//邻接点域;
        int weight;//权重
        Enode* next;
    }Enode;

    typedef struct Vnode {//顶点表节点
        int vertex;//顶点域
        Enode* firstedge;//边表头指针
    }Vnode;

public:
    int numV, numE;
    vector<Vnode> adjlist;

    //初始化函数
    graphL(int V,int E) {
        numV = V;
```

```
        numE = E;
        adjlist.resize(numV);
        for (int i = 0; i < numV; i++) {
            adjlist[i].vertex = i+1;
            adjlist[i].firstedge = nullptr;
        }
    }

    void CreateGraph_AL(graphL* gl) {
        Enode* e{};
        int v1, v2, w;;
        for (int k = 0; k < gl->numE; k++) {
            printf("输入边<vi,vj>的顶点序号和权重:\n");
            scanf("%d,%d,%d", &v1, &v2, &w);
            getchar();

            e = (Enode*)malloc(sizeof(Enode));
            if (e == nullptr) {
                exit(0);
            }
            e->adjvex = v2;
            e->weight = w;
            e->next = gl->adjlist[v1 - 1].firstedge;
            gl->adjlist[v1 - 1].firstedge = e;

            //这里一定要用 malloc 重新分配，
            //不然上面一个的地址和下面的地址会连起来
            e = (Enode*)malloc(sizeof(Enode));
            if (e == nullptr) {
                exit(0);
            }
            e->adjvex = v1;
            e->weight = w;
            e->next = gl->adjlist[v2 - 1].firstedge;
            gl->adjlist[v2 - 1].firstedge = e;
        }
    }

    void ShowGraph_AL(graphL* gl) {
        for (int i = 0; i < gl->numV; i++) {
            Enode* e = (Enode*)malloc(sizeof(Enode));
            if (e == nullptr) {
                exit(0);
            }
```

```
            e = gl->adjlist[i].firstedge;
            printf("顶点%d 的邻接表:\n%d->", i+1,i+1);
            while (e->next != nullptr) {
                printf("%d->", e->adjvex);
                e = e->next;
            }
            printf("%d\n", e->adjvex);

        }
    }
```

prim_optim 完成最小生成树。首先遍历 graphL 将数据导入 heap 中。然后与迪杰斯特拉算法一样，第一层循环每次输出一个标记点，内层嵌套第一个 for 循环通过堆 deleteheap 函数，删除堆顶节点并返回，第个 f 循环考虑新标记的节点对堆中的节点距离的影响。这里的 dist 是 prim 到生成树的距离，和迪杰斯特拉求到源点距离的区别。prim 里，只需要比较新的点对已有生成树中某一点的距离，和到生成树的距离。迪杰斯特拉里，需要比较新的点到已有点集合中某一点的距离+某点到源点的距离，和某点直接到源点的距离。
因此第四题的代码改变第三个循环的比较方式，就可以用于迪杰斯特拉算法的堆优化。

```
    //prim+堆（优先序列）
    int prim_optim(graphL* gl, IndexHeap* heap, int start) {

        int sum = 0;
        heap->ass->dist[start] = 0;

        //将 gl 图中的距离导入 heap 中
        Enode* e = new Enode;
        e = gl->adjlist[start-1].firstedge;//找到直接和出发点连接的点

        while (e != nullptr) {
            heap->ass->dist[e->adjvex] = e->weight;
            e = e->next;//先加入所有直接连接的点
        }

        for (int i = 1; i <= gl->numV; i++) {
            heap->InsertIndex(heap, i);
        }



        printf("路径如下:\n");
        printf("%d", start);
        while (heap->count!=1) {//起点也在树里面，下标为 0
```

```
            //此段代码展示了堆的节点，需要时可启用，查看堆节点的变化
            //for (int i = 1; i <= gl->numV; i++) {
                //if(heap->ass->dist[i]!=100000)
                //printf("%d,%d,%d\n", i, heap->ass->pos[i], heap->ass->dist[i]);
            //}

            int minIndex= heap->DeleteIndex(heap);
            int minDist = heap->ass->dist[minIndex];
            heap->ass->dist[minIndex] = 0;//该点加入生成树
            sum += minDist;



            //更新 dist 数组
            Enode* e = new Enode;
            e = gl->adjlist[minIndex-1].firstedge;

            while (e != nullptr) {
                int index = e->adjvex;
                if (heap->ass->dist[index] != 0 && heap->ass->dist[index] > e->weight){
                    heap->Fixup(heap, index, e->weight);
                    //这里的 dist 有 prim 到生成树的距离，和迪杰斯特拉求到源点距离的区别
                    //prim 里，只需要比较新的点对已有生成树中某一点的距离，和到生成树的距离
                    //迪杰斯特拉里，需要比较新的点到已有点集合中某一点的距离+某点到源点的距
离，
                    //和某点直接到源点的距离
                }
                e = e->next;//下一个节点

            }



            printf("->%d(%d)", minIndex, minDist);
        }
        printf("\n 总路径长度%d", sum);
        return sum;//最小生成树的路径和
    }
};
```

四、数据输入

第二题数据以起点，终点，权重形式输入。

```
30,38
1,2,120
14,15,186
30,33,192
12,7,252
```

```
7,6,195
27,28,268
27,17,308
28,8,53
8,6,132
6,1,206
3,4,127
4,5,270
8,9,233
5,9,134
9,17,77
9,26,96
9,30,303
17,18,124
18,29,390
18,19,410
30,10,175
26,10,290
26,25,322
25,24,459
19,11,219
11,24,137
10,11,115
19,20,86
24,23,521
22,23,320
22,21,252
21,20,320
20,16,460
16,15,580
13,28,288
12,13,70
2,4,173
2,3,133
```

第三题数据以地点名，站点形式输入。

```
教一楼
1
10
教二楼
2
17
18
教三楼
1
```

```
27
教四楼
2
5
9
理学院
2
19
20
计算机学院
3
12
13
14
生命科学院
1
1
化学学院
4
6
7
8
28
物理学院
1
25
经管学院
1
22
艺术学院
2
11
19
行政大楼
2
10
11
老图书馆
1
29
新图书馆
1
5
卓尔体育馆
```

```
2
7
12
```

五、函数运行

```
int main() {
    graphE* ge = new graphE;
    ge->CreateGraph_Edge(ge);
    //展示地图
    ge->ShowGraph_Edge(ge);
    Place place[15];
    for (int i = 0; i < 15; i++) {
        printf("请输入%d 号地点:\n",i+1);
        scanf("%s", place[i].name);
        getchar();
        printf("请输入该地点临近的站点个数:\n");
        int m=0;
        scanf("%d", &m);

        for (int j = 0; j < m; j++) {
            printf("请输入站点:\n");
            if (scanf("%d", &place[i].station[j]) == 1) {
                place[i].length++;
                getchar();
            }
            else
                break;
        }
    }

    //step3:寻路
    pathplanning(place, ge);

    //第四题

    graphL* gl=new graphL(30,38);
    gl->CreateGraph_AL(gl);
    gl->ShowGraph_AL(gl);

    IndexHeap* heap = new IndexHeap(30);
    gl->prim_optim(gl, heap, 10);
}
```

【结论】（结果）：

汉字字体使用微软雅黑 light，英文字体使用 consolas，字号为 5 号，段首缩进两个字符，行间距为固定值，16 磅，此处描述实验设计方案预期结果和程序的运行结果

第二题程序运行如下：

```
请输入顶点数和边数:
30,38
依次输入第1条边<vi,vj>的下标i,j和权重w:
1,2,120
依次输入第2条边<vi,vj>的下标i,j和权重w:
14,15,186
依次输入第3条边<vi,vj>的下标i,j和权重w:
13,14,192
依次输入第4条边<vi,vj>的下标i,j和权重w:
12,7,252
依次输入第5条边<vi,vj>的下标i,j和权重w:
7,6,195
依次输入第6条边<vi,vj>的下标i,j和权重w:
27,28,268
依次输入第7条边<vi,vj>的下标i,j和权重w:
27,17,308
依次输入第8条边<vi,vj>的下标i,j和权重w:
28,8,53
依次输入第9条边<vi,vj>的下标i,j和权重w:
8,6,132
依次输入第10条边<vi,vj>的下标i,j和权重w:
6,1,206
依次输入第11条边<vi,vj>的下标i,j和权重w:
3,4,127
依次输入第12条边<vi,vj>的下标i,j和权重w:
4,5,270
依次输入第13条边<vi,vj>的下标i,j和权重w:
8,9,233
依次输入第14条边<vi,vj>的下标i,j和权重w:
5,9,134
依次输入第15条边<vi,vj>的下标i,j和权重w:
9,17,77
依次输入第16条边<vi,vj>的下标i,j和权重w:
9,26,96
依次输入第17条边<vi,vj>的下标i,j和权重w:
9,30,303
依次输入第18条边<vi,vj>的下标i,j和权重w:
17,18,124
依次输入第19条边<vi,vj>的下标i,j和权重w:
18,29,390
依次输入第20条边<vi,vj>的下标i,j和权重w:
18,19,410
依次输入第21条边<vi,vj>的下标i,j和权重w:
30,10,175
依次输入第22条边<vi,vj>的下标i,j和权重w:
26,10,290
依次输入第23条边<vi,vj>的下标i,j和权重w:
26,25,322
依次输入第24条边<vi,vj>的下标i,j和权重w:
25,24,459
依次输入第25条边<vi,vj>的下标i,j和权重w:
19,11,219
```

GraphE 展示如下：

```
点集合:
[1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 ]
边集合:
[(1)->(2)<120>
 (2)->(1)<120>
 (1)->(6)<206>
 (14)->(15)<186>
 (13)->(14)<192>
 (12)->(7)<252>
 (7)->(12)<252>
 (7)->(6)<195>
 (6)->(7)<195>
 (6)->(1)<206>
 (3)->(4)<127>
 (2)->(4)<173>
 (2)->(3)<133>
 (3)->(2)<133>
 (4)->(3)<127>
 (4)->(2)<173>
 (4)->(5)<270>
 (5)->(4)<270>
 (5)->(9)<134>
 (6)->(8)<132>
 (8)->(28)<53>
 (8)->(6)<132>
 (8)->(9)<233>
 (9)->(8)<233>
 (9)->(5)<134>
 (9)->(17)<77>
 (9)->(26)<96>
 (9)->(30)<303>
 (10)->(30)<175>
 (10)->(26)<290>
 (10)->(11)<115>
 (11)->(19)<219>
 (11)->(10)<115>
 (11)->(24)<137>
 (12)->(13)<70>
 (13)->(12)<70>
 (14)->(13)<192>
 (13)->(28)<288>
 (15)->(14)<186>
 (27)->(28)<268>
 (27)->(17)<308>
 (17)->(27)<308>
 (17)->(9)<77>
 (16)->(20)<460>
 (16)->(15)<580>
 (15)->(16)<580>
 (17)->(18)<124>
 (26)->(9)<96>
 (18)->(17)<124>
```

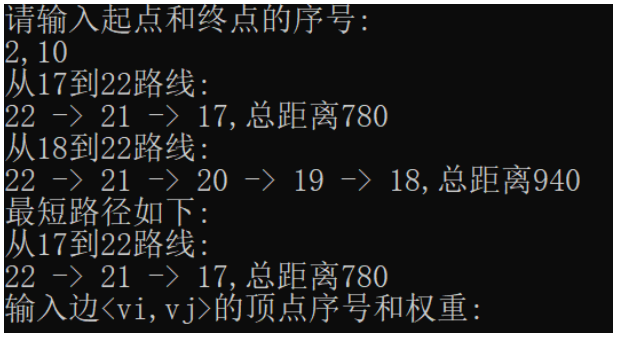
第三题数据输入如下：

```
请输入1号地点:
教一楼
请输入该地点临近的站点个数:
1
请输入站点:
10
请输入2号地点:
教二楼
请输入该地点临近的站点个数:
2
请输入站点:
17
请输入站点:
18
请输入3号地点:
教三楼
请输入该地点临近的站点个数:
1
请输入站点:
27
请输入4号地点:
教四楼
请输入该地点临近的站点个数:
2
请输入站点:
5
请输入站点:
9
请输入5号地点:
理学院
请输入该地点临近的站点个数:
2
请输入站点:
19
请输入站点:
20
请输入6号地点:
计算机学院
请输入该地点临近的站点个数:
3
请输入站点:
12
请输入站点:
13
请输入站点:
14
请输入7号地点:
生命科学院
请输入该地点临近的站点个数:
1
请输入站点:
1
```

结果展示如下：

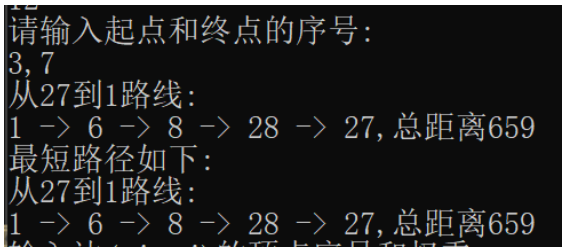
以 2 和 10 为例，输出最短路径

22->21->17，总距离 780



以 3 和 7 为例，输出最短路径

1->6->8->28->27，总距离 659



第四题数据输入如下：

1,2,120

14,15,186

13,14,192

12,7,252

7,6,195

27,28,268

27,17,308

28,8,53

8,6,132

6,1,206

3,4,127

4,5,270

8,9,233

5,9,134

9,17,77

9,26,96

9,30,303

17,18,124

18,29,390

18,19,410

30,10,175

26,10,290

26,25,322

25,24,459

19,11,219

11,24,137

10,11,115

19,20,86

24,23,521

22,23,320

22,21,252

21,20,320

20,16,460

16,15,580

13,28,288

12,13,70

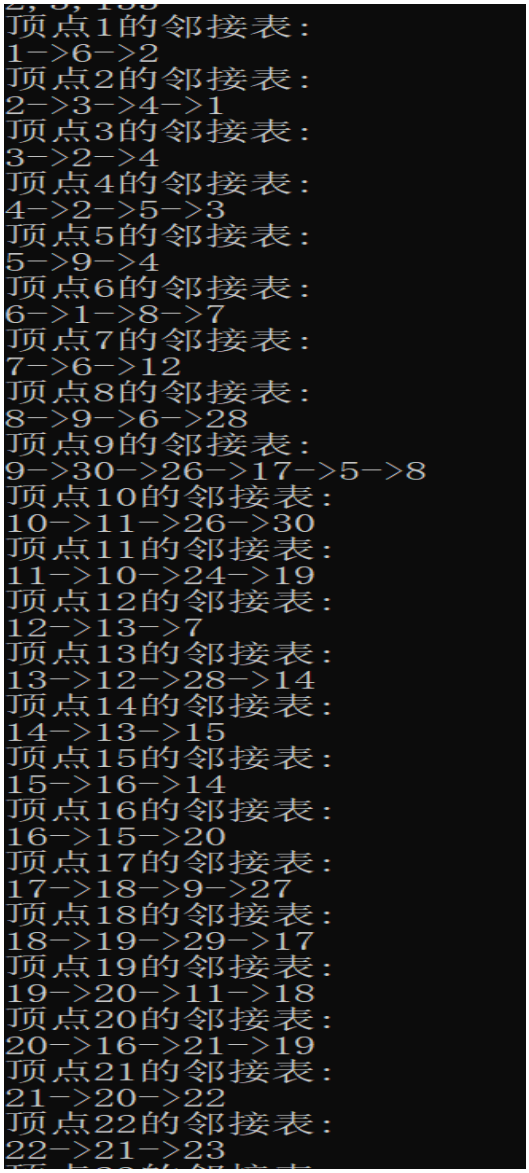
2,4,173

2,3,133

输入情况如下：

```
输入边<vi,vj>的顶点序号和权重:
6,1,206
输入边<vi,vj>的顶点序号和权重:
3,4,127
输入边<vi,vj>的顶点序号和权重:
4,5,270
输入边<vi,vj>的顶点序号和权重:
8,9,233
输入边<vi,vj>的顶点序号和权重:
5,9,134
输入边<vi,vj>的顶点序号和权重:
9,17,77
输入边<vi,vj>的顶点序号和权重:
9,26,96
输入边<vi,vj>的顶点序号和权重:
9,30,303
输入边<vi,vj>的顶点序号和权重:
17,18,124
输入边<vi,vj>的顶点序号和权重:
18,29,390
输入边<vi,vj>的顶点序号和权重:
18,19,410
输入边<vi,vj>的顶点序号和权重:
30,10,175
输入边<vi,vj>的顶点序号和权重:
26,10,290
输入边<vi,vj>的顶点序号和权重:
26,25,322
输入边<vi,vj>的顶点序号和权重:
25,24,459
输入边<vi,vj>的顶点序号和权重:
19,11,219
输入边<vi,vj>的顶点序号和权重:
11,24,137
```

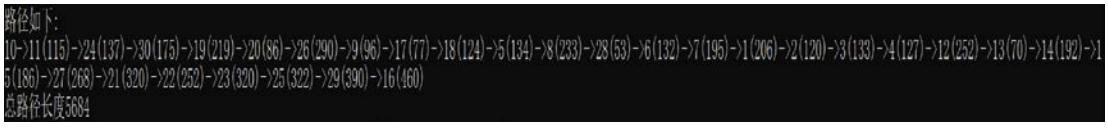
图展示如下：



输出最小生成树如下：

箭头表示生成顺序，不代表路径方向

总距离长度 5684



结果与预期相同，程序运行无误，实验设计目标达成，实验结束。

【小结】：

经过半个多月的努力，终于完成了本次实验报告，通过本次实验完成了基于 AVL 树的集合构建、边集数组的图信息储存和调用，迪杰斯特拉算法和 prim 算法的应用，堆算法的优化，深刻理解了平衡二叉树的构建算法和递归算法、指针、数组和链表等结构的运用，以普利姆和迪杰斯特拉等广度优先算法(DFS) 的实践应用，完成了对武汉大学地图主体部分的路线规划，具有一定的现实作用。从产品的角度，以实用性为第一原则来构建算法和程序，尤其是程序运行的便捷性。

本实验仍有需要改进的地方，首先是集合的模板构建，本实验采用了模板特化时重载运算符的方式来完成集合的通用功能，但不具备通用性，如何做到像数组一样，无需用户自行定义运算法则，自动识别或者有用户友好的识别方式，才能让本实验中的集合得以更广泛的运用。其次是程序的数据输入采用了 printf 和 scanf 的组合，需要用户频繁输入数据，即使给出了预输入的数据格式，但仍需要用于手动输入，并不具有可迁移性，应简化操作步骤，优先考虑使用者体验。然后，作为以实际应用为目的的程序设计方案，应当配备人性化的操作界面和可视化图像，本程序并不能直观理解二叉树和图的结构。最后，prim 算法适用于稠密图，本题是稀疏图，即使采用邻接表存储，用 prim 算法求最小生成树存在一定的效率浪费，应当尝试更合适的算法。

最后，感谢蒋老师在本学期数据结构课程上的倾囊相授和认真答疑，和同学们一起营造了良好学习氛围，让我对数据结构和算法有了一个基础的理解，也让我对计算机实际应用中效率与空间的权衡艺术产生了浓厚的兴趣。

祝本课程的老师和同学们一帆风顺！

2020301052272 王俊儒

指导教师评语及成绩

【评语】：

成绩： **指导教师签名：**

批阅日期：

附件：

实验报告说明

1．实验项目名称：要用最简练的语言反映实验的内容。要求与实验指导书中相一致。

2．实验目的：目的要明确，要抓住重点，符合实验任务书中的要求。

3．实验环境：实验用的软硬件环境（配置）。

4．实验方案设计（思路、步骤和方法等）：这是实验报告极其重要的内容。包括概要设计、详细设计和核心算法说明及分析，系统开发工具等。应同时提交程序或设计电子版。

对于**设计型和综合型实验**，在上述内容基础上还应该画出流程图、设计思路和设计方法，再配以相应的文字说明。

对于**创新型实验**，还应注明其创新点、特色。

5．结论（结果）：即根据实验过程中所见到的现象和测得的数据，做出结论（可以将部分测试结果进行截屏）。

6．小结：对本次实验的心得体会，所遇到的问题及解决方法，其他思考和建议。

7．指导教师评语及成绩：指导教师依据学生的实际报告内容，用简练语言给出本次实验报告的评价和价值。