《数据结构》课程设计报告

学生姓名 徐 智 涵

班　　级 计科 2101

学　　号 211302124

指导老师 陈 宏 建

成　　绩

时　　间 2022.12.22～2023.1.1

序号	项目名称	考核比重	得分
1	方案设计	10%	
2	系统设计	50%	
3	设计验收	20%	
4	设计报告	20%	

2023 年　1 月　3　日

《数据结构课程设计》任务书

课程编号	10130090	学分	1	周数	1 周
学号	211302124	专业	计算机科学与技术		
姓名	徐智涵	班级	计科 2101		

目的与要求：

培养学生综合程序设计的能力，训练学生灵活应用所学数据结构知识，独立完成问题分析、总体设计、详细设计和编程实现等软件开发全过程的综合实践能力。巩固、深化学生的理论知识，提高编程水平，并在此过程中培养他们严谨的科学态度和良好的学习作风。为今后学习其他计算机课程打下基础。

能够根据计算机及应用领域复杂工程问题，选择适当的数据结构及其存储结构分析确定解决问题的算法，并能对其进行有效性和时空性能分析。

通过对计算机及应用领域的复杂工程的综合分析，利用数据结构的有关算法设计出复杂问题系统的解决方案并能对方案进行合理有效分析并得出结论。

主要任务及具体要求：

任务：

设计并实现一个复杂问题的系统（根据学号顺序选两题完成，题目见报告），开发工具自选，并根据要求写出设计报告。

要求：

课程设计为学生提供了一个既动手又动脑，独立实践的机会，将书本上的理论知识和工作、生产实际有机地结合起来，从而锻炼学生分析问题、解决实际问题的能力，提高学生的编程序能力和创新意识。在处理每个题目时，要求从分析题目的需求入手，按设计抽象数据类型、构思算法、通过算法的设计实现抽象数据类型、编制上机程序和上机调试等若干步骤完成题目，最终写出完整的课程设计与程序分析报告。前期准备工作完备与否直接影响到后序上机调试工作的效率。

进程安排：

时间	内容	时间（天）	授课形式
1	课题讲解，方案设计	0.5	讲练结合
2	系统设计	4	讲练结合
3	设计验收	2	讲练结合
4	设计报告	0.5	讲练结合

2022 年 12 月 18 日

《数据结构课程设计》答辩记录表

学号	211302124	专业	计算机科学与技术
姓名	徐智迪	班级	计科2101

答辩记录：

~~问述Dijkstra算法的思想~~

问题一：简述Dijkstra算法的思想

答：将顶点分为两个集合V1、V2，先将源点加入V1，其余点加入V2，并用集合S记录经由V1中顶点到达V2中顶点的单源最短路径，将S中当前最短路径对应的在V2中的顶点加入V1，更新S，直至V2为中，到S中记录了单源最短路径。

问题二：简述稀疏矩阵快速转置的算法思想

答：利用记录每列非零元素个数的number数组和记录每列第一个非零元素在转置后矩阵的三元组表中位置的数组position来快速构建新矩阵的三元组表。

问题三：简述你设计系统过程中遇到的问题及解决办法，并谈谈对本次课程设计的体会

答：由于Kruskal并非课堂教学重点内容，故一开始设计时遇到了较大的阻碍，不清楚如何判断回路。后来，通过搜索网络上的资料，借助查并集的思想解决了回路的判断问题，并完成了设计。通过此次设计，我深刻了解并加强了对数据结构这门课程的掌握程度，获益良多。

答辩人：徐智迪（手写签名）　　日期：2023.1.2

目录

一、课程设计题目描述……2

1、校园网架设的方案与设计问题……2

2、设计一个矩阵运算器……2

二、课题数据结构与算法……2

1、校园网架设的方案与设计问题……2

（1）数据结构……2

（2）算法……3

2、设计一个矩阵运算器……4

（1）数据结构……4

（2）算法……4

三、测试数据及运行结果……5

1、校园网架设的方案与设计问题……5

2、设计一个矩阵运算器……6

四、体会与总结……10

五、源程序……11

1、校园网架设的方案与设计问题……11

2、设计一个矩阵运算器 ……21

一、课程设计题目描述

1、校园网架设的方案与设计问题

【问题描述】

若要在扬州大学的七个校区（江阳路南校区、江阳路北校区、瘦西湖校区、农学院校区、工学院校区、水利学院校区、医学院校区）之间架设校园网，如何以最低的经济代价架设这个校园网。并求出江阳路南校区到其他各校区之间的最短距离。

【基本要求】

（1）利用二种方法（Prim 算法和克鲁斯卡尔（Kruskual）算法生成校园网的架设方案

（2）利用迪杰斯特拉算法求出江阳路校区到其他各校区之间的最短距离。

（3）将七个校区设计为一个 DAG 图，求出一组拓扑排序序列和关键路径。

【测试数据】

对每种方法设定一组模拟测试数据进行测试，验证程序的正确性。

2、设计一个矩阵运算器

【问题描述】

设计一个矩阵运算器，对矩阵进行乘方(^)、加(+)、减(-)、乘(*)、转置等运算；

【基本要求】

（1） 求含有乘方(^)、加(+)、减(-)、乘(*)运算；。

【测试数据】

分别选定一组测试数据进行测试，验证程序的正确性。

二、课题数据结构与算法

1、校园网架设的方案与设计问题

（1）数据结构

```
struct MGraph//邻接矩阵
{
    char* vexs[MAX];
    int** arcs;
    int vexnum;
};

struct CloseEdge//记录最小权值边的信息
{
    int distance;
    int row;
    int col;
};

struct TArry//三元组表
{
    int distance;
    int vertx01;
```

```
    int vertx02;
};

struct ArcNode//节点
{
    int adjvex;
    struct ArcNode* nextarc;
    int distance;
};

struct VNode//首节点
{
    char* vertx;
    int indegree;
    struct ArcNode* firstarc;
};

struct ALGraph//邻接表
{
    struct VNode* vertices;
    int vexnum;
};
```

（2）算法

```
struct MGraph* MGraphCreat()//创建邻接矩阵

void Prime(struct MGraph* Graph)//Prime 算法

void copy(struct TArry* tArry01, struct TArry* tArry02)//将 tArry02 的信息复制给 tArry01
void QuickSort(struct TArry* tArry, int top, int bottom)//快速排序
bool judgement(struct TArry* tArry, int* unionFind, int i)//判断两个顶点是否属于同一集合
void Kruskal(struct MGraph* Graph)//Kruskal 算法

void Dijkstra(struct MGraph* Graph)//Dijkstra 算法

struct ALGraph* ALGraphCreat()//创建邻接表

void TopologicalSort(struct ALGraph* Graph)//拓扑排序

void DFS(struct VNode* NodeList, struct ArcNode* Node, int sum, int top, int* temp, int* stack, int*
path)//深度优先搜索
void CriticalPath(struct ALGraph* Graph)//关键路径

void SolutionForOneAndTwo()//基于邻接矩阵创建的图所得的方案集合
void SolutionForThree()//基于邻接表法创建的图所得的方案集合
```

2、设计一个矩阵运算器

（1）数据结构

```
struct Triple//三元组表
{
	int row;
	int col;
	int data;
};

struct TSMatrix//矩阵
{
	struct Triple Elem[MAXSIZE];
	int position[MAXSIZE];
	int length_row, length_col, length_Elem;
};
```

（2）算法

```
struct TSMatrix* MatrixInit()//创建参与运算的矩阵
int** matrixInit(struct TSMatrix* TsMatrix)  //创建用于临时记录的矩阵
struct TsMatrix* Copy(struct TSMatrix* TsMatrix, int** matrix)  //复制矩阵
void print(struct TSMatrix* TsMatrix)  //打印矩阵信息

struct TsMatrix* AddMatrix(struct TSMatrix* A, struct TSMatrix* B)  //加法运算
struct TsMatrix* SubMatrix(struct TSMatrix* A, struct TSMatrix* B)  //减法运算
struct TSMatrix* MultMatrix(struct TSMatrix* A, struct TSMatrix* B)  //乘法运算
struct TSMatrix* FastTransMatrix(struct TSMatrix* A)  //转置运算

void Involution()//调用乘法运算进行矩阵的乘方
void Addition()//调用加法运算进行矩阵的加法
void Subtraction()//调用减法运算进行矩阵的减法
void Multiplication()//调用乘法运算进行矩阵的乘法
void Transpose()//调用转置运算进行矩阵的转置

void menu()//菜单函数
```

三、测试数据及运行结果

1、校园网架设的方案与设计问题

测试数据①：

	江阳路南校区	江阳路北校区	瘦西湖校区	农学院校区	工学院校区	水利学院校区	医学院校区
江阳路南校区	0	15	27	31	14	19	22
江阳路北校区	15	0	13	29	11	16	18
瘦西湖校区	27	13	0	26	33	23	18
农学院校区	31	29	26	0	17	11	19
工学院校区	14	11	33	17	0	12	21
水利学院校区	19	16	23	11	12	0	17
医学院校区	22	18	18	19	21	17	0

Prime 算法和 Kruskal 算法求校园网架设方案：

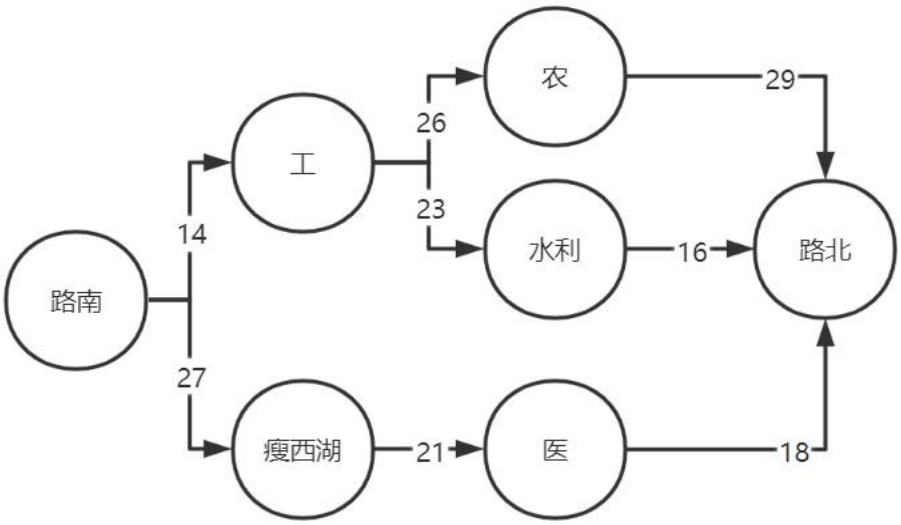
```
Result of Prime:
(江阳路南校区,工学院校区)
(工学院校区,江阳路北校区)
(工学院校区,水利学院校区)
(水利学院校区,农学院校区)
(江阳路北校区,瘦西湖校区)
(水利学院校区,医学院校区)
Distance: 78
```

```
Result of Kruskal:
(江阳路北校区,工学院校区)
(农学院校区,水利学院校区)
(工学院校区,水利学院校区)
(江阳路北校区,瘦西湖校区)
(江阳路南校区,工学院校区)
(水利学院校区,医学院校区)
Distance: 78
```

Dijkstra 算法求江阳路南校区到其他校区的最短距离：

```
Result of Dijkstra:
From 江阳路南校区 to 工学院校区: 14
From 江阳路南校区 to 江阳路北校区: 15
From 江阳路南校区 to 水利学院校区: 19
From 江阳路南校区 to 医学院校区: 22
From 江阳路南校区 to 瘦西湖校区: 27
From 江阳路南校区 to 农学院校区: 30
```

测试数据②：



将七个校区设计为一个 DAG 图，求出一组拓扑排序序列和关键路径：

```
Result of CriticalPath:
江阳路南校区->瘦西湖校区->医学院校区->江阳路北校区->End

Result of TopologicalSort:
江阳路南校区->工学院校区->水利学院校区->农学院校区->瘦西湖校区->医学院校区->江阳路北校区->End

Press any key to continue . . .
```

2、设计一个矩阵运算器

测试数据①：

矩阵一：					矩阵二：			
0	5	5	7	|	8	4	7	1
1	4	8	6	|	4	3	9	9
9	0	0	3	|	3	0	8	0
3	2	0	4	|	6	7	0	0
5	0	5	1	|	5	4	4	3

加法：

```
Please enter the number of rows and columns of the matrix.
row: 5
col: 4

row 1: 0 5 5 7
row 2: 1 4 8 6
row 3: 9 0 0 3
row 4: 3 2 0 4
row 5: 5 0 5 1

Please enter the number of rows and columns of the matrix.
row: 5
col: 4

row 1: 8 4 7 1
row 2: 4 3 9 9
row 3: 3 0 8 0
row 4: 6 7 0 0
row 5: 5 4 4 3

The result is:
   8   9  12   8
   5   7  17  15
  12   0   8   3
   9   9   0   4
  10   4   9   4
```

row	col	data
1	1	8
1	2	9
1	3	12
1	4	8
2	1	5
2	2	7
2	3	17
2	4	15
3	1	12
3	3	8
3	4	3
4	1	9
4	2	9
4	4	4
5	1	10
5	2	4
5	3	9
5	4	4

减法：

```
Please enter the number of rows and columns of the matrix.
row: 5
col: 4

row 1: 0 5 5 7
row 2: 1 4 8 6
row 3: 9 0 0 3
row 4: 3 2 0 4
row 5: 5 0 5 1

Please enter the number of rows and columns of the matrix.
row: 5
col: 4

row 1: 8 4 7 1
row 2: 4 3 9 9
row 3: 3 0 8 0
row 4: 6 7 0 0
row 5: 5 4 4 3

The result is:
  -8   1  -2   6
  -3   1  -1  -3
   6   0  -8   3
  -3  -5   0   4
   0  -4   1  -2
```

row	col	data
1	1	-8
1	2	1
1	3	-2
1	4	6
2	1	-3
2	2	1
2	3	-1
2	4	-3
3	1	6
3	3	-8
3	4	3
4	1	-3
4	2	-5
4	4	4
5	2	-4
5	3	1
5	4	-2

测试数据②：

矩阵一：				矩阵二：		
0	5	5	7	8	4	7
1	4	8	6	4	3	9
9	0	0	3	3	0	8
				6	7	0

乘法：

```
Please enter the number of rows and columns of the matrix.
row: 3
col: 4

row 1: 0 5 5 7
row 2: 1 4 8 6
row 3: 9 0 0 3

Please enter the number of rows and columns of the matrix.
row: 4
col: 3

row 1: 8 4 7
row 2: 4 3 9
row 3: 3 0 8
row 4: 6 7 0

The result is:
  77  64  85
  84  58 107
  90  57  63
```

row	col	data
1	1	77
1	2	64
1	3	85
2	1	84
2	2	58
2	3	107
3	1	90
3	2	57
3	3	63

转置：

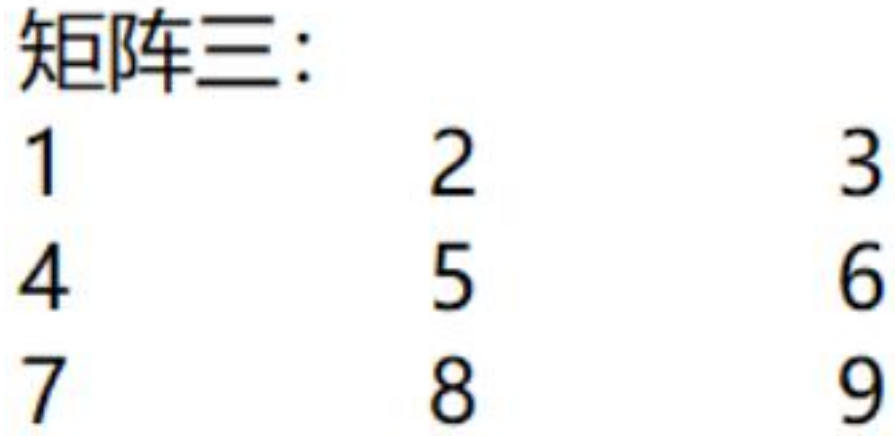
```
Before transpose:
   0    5    5    7
   1    4    8    6
   9    0    0    3
```

row	col	data
1	2	5
1	3	5
1	4	7
2	1	1
2	2	4
2	3	8
2	4	6
3	1	9
3	4	3

```
After transpose:
   0    1    9
   5    4    0
   5    8    0
   7    6    3
```

row	col	data
1	2	1
1	3	9
2	1	5
2	2	4
3	1	5
3	2	8
4	1	7
4	2	6
4	3	3

测试数据③：



乘方：

```
Please enter the
row: 3
col: 3

row 1: 1 2 3
row 2: 4 5 6
row 3: 7 8 9

The result is:
  30  36  42
  66  81  96
 102 126 150
```

row	col	data
1	1	30
1	2	36
1	3	42
2	1	66
2	2	81
2	3	96
3	1	102
3	2	126
3	3	150

四、体会与总结

本次课程设计共有两个题目。其中，题目一为利用图的知识建设一个包含七个校区的校园网络，利用二种方法（Prime 算法和 Kruskual 算法）生成校园网的架设方案。Prime 算法是先找到一个顶点添加到一个集合中，再将与集合内某点构成的边的权值最小的集合外的一个顶点添加到集合内，重复这一过程直至所有顶点都被添加到集合内,此时生成了一个包含了 n 个顶点和 n－1 条边(本例中 n = 7）的最小生成树，即校园网的架设方案；Kruskual 算法需要先将所有边根据权值由小到大进行排序，再借助并查集将所有顶点并入同一集合且不形成回路；利用 **Dijkstra** 算法求单源最短路径的思想与 **Prime** 算法类似，只需将 **Prime** 算法中集合内外各一点构成的单条边的权值，用从源点出发的路径的总权值替换，以作为新考量对象即可；要进行拓扑排序，需遍历所有顶点并将入度为 **0** 的顶点添加到一个集合中，而所有以其为前驱的顶点入度减一，重复这一过程直至所有点被添加到集合中，因为在过程中可能同时有多个顶点入度为 **0**，故最终生成的序列不唯一；利用 **DAG** 图求关键路径，可以借助子集树的思想求出从源点出发的权值最大的路径，即关键路径。题目二是设计一个矩阵运算器实现矩阵的加法、减法、乘法及乘方运算，先根据输入矩阵信息创建一个对应的三元组表，再根据矩阵运算的性质进行相应的计算。

通过本次课程设计，遇到了一些预期中的问题，也遇到了不少意料之外的问题。通过解决这些问题，我进一步理解了图和矩阵的基本操作，加深了对于各种算法的思想及其具体实现方法的掌握，这对我在数据结构这门课程的学习上助益良多。

五、源程序

1、校园网架设的方案与设计问题

```
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>
#include<string.h>
#include<windows.h>
#define MAX 16
#define INFINITY 10000

struct MGraph//邻接矩阵
{
	char* vexs[MAX];
	int** arcs;
	int vexnum;
};

struct MGraph* MGraphCreat()//创建邻接矩阵
{
	FILE* fp;
	if (!(fp = fopen("data.txt", "r")))
	{
		printf("File opened failed!\n");
		exit(0);
	}
	struct MGraph* Graph = (struct MGraph*)malloc(sizeof(struct MGraph));
	fscanf(fp, "%d", &Graph->vexnum);
	if (Graph->vexnum > MAX)
	{
		printf("Array out of bounds!\n");
		exit(0);
	}
	for (int i = 0; i < Graph->vexnum; i++)
	{
		char* str = (char*)malloc(sizeof(char) * 100);
		fscanf(fp, "%s", str);
		Graph->vexs[i] = str;
	}
	int** matrix = (int**)malloc(sizeof(int*) * Graph->vexnum);
	for (int i = 0; i < Graph->vexnum; i++)
	{
		matrix[i] = (int*)malloc(sizeof(int) * Graph->vexnum);
		for (int j = 0; j < Graph->vexnum; j++)
```

```
            {
                fscanf(fp, "%d", &matrix[i][j]);
                if (!matrix[i][j])matrix[i][j] = INFINITY;
            }
        }
        fclose(fp);
        Graph->arcs = matrix;
        return Graph;
}

struct CloseEdge//记录最小权值边的信息
{
        int distance;
        int row;
        int col;
};

void Prime(struct MGraph* Graph)//Prime算法
{
        int vexnum = Graph->vexnum;
        int** matrix = Graph->arcs;
        bool* visited = (bool*)malloc(sizeof(bool) * vexnum);
        struct CloseEdge* closeEdge = (struct CloseEdge*)malloc(sizeof(struct CloseEdge) * vexnum);
        for (int i = 0; i < vexnum; i++)
        {
                visited[i] = false;
                closeEdge[i].distance = INFINITY;
                closeEdge[i].row = i;
                closeEdge[i].col = i;
        }
        int mark = 0, count = vexnum - 1, sum = 0;
        printf("Result of Prime: \n");
        while (count--)
        {
                visited[mark] = true;
                int i = mark, min = INFINITY;
                for (int j = 0; j < vexnum; j++)
                {
                        if (!visited[j] && matrix[i][j] < closeEdge[j].distance)
                        {
                                closeEdge[j].distance = matrix[i][j];
                                closeEdge[j].row = i;
                                closeEdge[j].col = j;
                        }
```

```
                if (closeEdge[j].distance < min)
                {
                    min = closeEdge[j].distance;
                    mark = j;
                }
            }
            sum = sum + closeEdge[mark].distance;
            closeEdge[mark].distance = INFINITY;
            printf("(%s,%s)\n", Graph->vexs[closeEdge[mark].row], Graph->vexs[closeEdge[mark].col]);
        }
        printf("Distance: %d\n\n", sum);
}

struct TArry//三元组表
{
        int distance;
        int vertx01;
        int vertx02;
};

void copy(struct TArry* tArry01, struct TArry* tArry02)//将tArry02的信息复制给tArry01
{
        tArry01->distance = tArry02->distance;
        tArry01->vertx01 = tArry02->vertx01;
        tArry01->vertx02 = tArry02->vertx02;
}

void QuickSort(struct TArry* tArry, int top, int bottom)//快速排序
{
        if (top < bottom)
        {
            int i = top, j = bottom;
            struct TArry* flag = (struct TArry*)malloc(sizeof(struct TArry));
            struct TArry* temp = (struct TArry*)malloc(sizeof(struct TArry));
            copy(flag, &tArry[i]);
            while (i < j)
            {
                while (tArry[j].distance >= flag->distance)j--;
                if (i < j)
                {
                    copy(temp, &tArry[i]);
                    copy(&tArry[i++], &tArry[j]);
                    copy(&tArry[j], temp);
                }
```

```
            while (tArry[i].distance < flag->distance)i++;
            if (i < j)
            {
                copy(temp, &tArry[j]);
                copy(&tArry[j--], &tArry[i]);
                copy(&tArry[i], temp);
            }
        }
        copy(&tArry[i], flag);
        QuickSort(tArry, top, i - 1);
        QuickSort(tArry, i + 1, bottom);
    }
}

bool judgement(struct TArry* tArry, int* unionFind, int i)//判断两个顶点是否属于同一集合
{
    int vertx01 = tArry[i].vertx01, vertx02 = tArry[i].vertx02;
    int temp01 = unionFind[vertx01], temp02 = unionFind[vertx02];
    while (temp01 != unionFind[temp01])
    {
        temp01 = unionFind[temp01];
    }
    while (temp02 != unionFind[temp02])
    {
        temp02 = unionFind[temp02];
    }
    if (temp01 == temp02)return false;
    else
    {
        unionFind[temp01] = unionFind[temp02];
        return true;
    }
}

void Kruskal(struct MGraph* Graph)//Kruskal算法
{
    int vexnum = Graph->vexnum;
    int** matrix = Graph->arcs;
    int arcnum = vexnum * (vexnum - 1) / 2;
    struct TArry* tArry = (struct TArry*)malloc(sizeof(struct TArry) * arcnum);
    int* unionFind = (int*)malloc(sizeof(int) * vexnum);
    int index = 0;
    for (int i = 0; i < vexnum; i++)
    {
```

```
            unionFind[i] = i;
            for (int j = i + 1; j < vexnum; j++)
            {
                tArry[index].distance = matrix[i][j];
                tArry[index].vertx01 = i;
                tArry[index].vertx02 = j;
                index++;
            }
        }
        QuickSort(tArry, 0, arcnum - 1);
        int count = 1, sum = 0;
        printf("Result of Kruskal: \n");
        for (int i = 0; i < arcnum && count < vexnum; i++)
        {
            if (judgement(tArry, unionFind, i))
            {
                printf("(%s,%s)\n", Graph->vexs[tArry[i].vertx01], Graph->vexs[tArry[i].vertx02]);
                count++;
                sum = sum + tArry[i].distance;
            }
        }
        printf("Distance: %d\n\n", sum);
}

void Dijkstra(struct MGraph* Graph)//Dijkstra算法
{
        int vexnum = Graph->vexnum;
        int** matrix = Graph->arcs;
        bool* visited = (bool*)malloc(sizeof(bool) * vexnum);
        int* bestDistance = (int*)malloc(sizeof(int) * vexnum);
        int* prior = (int*)malloc(sizeof(int) * vexnum);
        int count = vexnum - 1, best = 0, origin = 0;
        for (int i = 0; i < vexnum; i++)
        {
            visited[i] = false;
            bestDistance[i] = matrix[origin][i];
            prior[i] = origin;
        }
        prior[origin] = -1;
        int index = origin;
        printf("Result of Dijkstra: \n");
        while (count--)
        {
            visited[index] = true;
```

```
            int i = index, min = INFINITY;
            for (int j = 0; j < vexnum; j++)
            {
                if (!visited[j])
                {
                    if (matrix[i][j] + best < bestDistance[j])
                    {
                        bestDistance[j] = matrix[i][j] + best;
                        prior[j] = i;
                    }
                    if (bestDistance[j] < min)
                    {
                        min = bestDistance[j];
                        index = j;
                    }
                }
            }
            best = bestDistance[index];
            printf("From %s to %s: %d\n", Graph->vexs[origin], Graph->vexs[index], best);
        }
        printf("\n");
}

struct ArcNode//节点
{
        int adjvex;
        struct ArcNode* nextarc;
        int distance;
};

struct VNode//首节点
{
        char* vertx;
        int indegree;
        struct ArcNode* firstarc;
};

struct ALGraph//邻接表
{
        struct VNode* vertices;
        int vexnum;
};

struct ALGraph* ALGraphCreat()//创建邻接表
```

```
{
    FILE* fp;
    fp = fopen("data.txt", "rb");
    if (!fp)
    {
        printf("File open failed!\n");
        exit(0);
    }
    struct ALGraph* Graph = (struct ALGraph*)malloc(sizeof(struct ALGraph));
    fscanf(fp, "%d", &Graph->vexnum);
    struct VNode* NodeList = (struct VNode*)malloc(sizeof(struct VNode) * Graph->vexnum);
    for (int i = 0; i < Graph->vexnum; i++)
    {
        char* str = (char*)malloc(sizeof(char) * 100);
        fscanf(fp, "%s", str);
        NodeList[i].vertx = str;
        NodeList[i].indegree = 0;
        NodeList[i].firstarc = NULL;
    }
    int** matrix = (int**)malloc(sizeof(int*) * Graph->vexnum);
    for (int i = 0; i < Graph->vexnum; i++)
    {
        matrix[i] = (int*)malloc(sizeof(int) * Graph->vexnum);
        for (int j = 0; j < Graph->vexnum; j++)
        {
            fscanf(fp, "%d", &matrix[i][j]);
            if (!matrix[i][j])matrix[i][j] = INFINITY;
        }
    }
    int vex01, vex02;
    fscanf(fp, "%d%d", &vex01, &vex02);
    while (vex01 != vex02)
    {
        struct ArcNode* NewNode = (struct ArcNode*)malloc(sizeof(struct ArcNode));
        NewNode->adjvex = vex02;
        NewNode->nextarc = NodeList[vex01].firstarc;
        NewNode->distance = matrix[vex01][vex02];
        NodeList[vex01].firstarc = NewNode;
        NodeList[vex02].indegree++;
        fscanf(fp, "%d%d", &vex01, &vex02);
    }
    Graph->vertices = NodeList;
    fclose(fp);
    return Graph;
```

```
}

void TopologicalSort(struct ALGraph* Graph)//拓扑排序
{
	int vexnum = Graph->vexnum, top = 0;
	int* stack = (int*)malloc(sizeof(int) * MAX);
	struct VNode* NodeList = Graph->vertices;
	for (int i = 0; i < vexnum; i++)
	{
		if (NodeList[i].indegree == 0)
		{
			stack[top++] = i;
			NodeList[i].indegree--;
		}
	}
	printf("Result of TopologicalSort: \n");
	while (top--)
	{
		int index = stack[top];
		printf("%s->", NodeList[index].vertx);
		struct ArcNode* Node = NodeList[index].firstarc;
		while (Node != NULL)
		{
			index = Node->adjvex;
			NodeList[index].indegree--;
			Node = Node->nextarc;
		}
		for (int i = 0; i < vexnum; i++)
		{
			if (NodeList[i].indegree == 0)
			{
				stack[top++] = i;
				NodeList[i].indegree--;
			}
		}
	}
	printf("End\n\n");
}

void DFS(struct VNode* NodeList, struct ArcNode* Node, int sum, int top, int* temp, int* stack, int*
path)//深度优先搜索
{
	if (Node == NULL)
	{
```

```
            if (sum > temp[0])
            {
                temp[0] = sum;
                for (int i = 0; i < top; i++)
                {
                    path[i] = stack[i];
                    temp[2] = temp[1];
                }
                path[top] = '\0';
            }
            return;
        }
        while (Node != NULL)
        {
            int index = Node->adjvex;
            stack[top++] = index;
            sum = sum + Node->distance;
            DFS(NodeList, NodeList[index].firstarc, sum, top, temp, stack, path);
            sum = sum - Node->distance;
            Node = Node->nextarc;
            top--;
        }

}

void CriticalPath(struct ALGraph* Graph)//关键路径
{
        int vexnum = Graph->vexnum, top = 0;
        int* stack = (int*)malloc(sizeof(int) * MAX);
        int* path = (int*)malloc(sizeof(int) * MAX);
        int* temp = (int*)malloc(sizeof(int) * 3);
        struct VNode* NodeList = Graph->vertices;
        for (int i = 0; i < 3; i++)
            temp[i] = 0;
        for (int i = 0; i < vexnum; i++)
        {
            if (NodeList[i].indegree == 0)
            {
                temp[1] = i;
                struct ArcNode* Node = NodeList[i].firstarc;
                DFS(NodeList, Node, 0, top, temp, stack, path);
            }
        }
        printf("Result of CriticalPath: \n");
```

```
        printf("%s->", NodeList[temp[2]].vertx);
        while (path[top] != '\0')
        {
                int index = path[top];
                printf("%s->", NodeList[index].vertx);
                top++;
        }
        printf("End\n\n");
}

void SolutionForOneAndTwo()//基于邻接矩阵创建的图所得的方案集合
{
        struct MGraph* Graph = MGraphCreat();
        Prime(Graph);
        Kruskal(Graph);
        Dijkstra(Graph);
}

void SolutionForThree()//基于邻接表法创建的图所得的方案集合
{
        struct ALGraph* Graph = ALGraphCreat();
        CriticalPath(Graph);
        TopologicalSort(Graph);
}

int main()
{
        SetConsoleOutputCP(65001);
        SolutionForOneAndTwo();
        SolutionForThree();
        system("pause");
        return 0;
}
```

2、设计一个矩阵运算器

```
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>
#define MAXSIZE 128

struct Triple
{
	int row;
	int col;
	int data;
};

struct TSMatrix
{
	struct Triple Elem[MAXSIZE];
	int position[MAXSIZE];
	int length_row, length_col, length_Elem;
};

struct TSMatrix* MatrixInit()
{
	struct TSMatrix* TsMatrix = (struct TSMatrix*)malloc(sizeof(struct TSMatrix));
	printf("Please enter the number of rows and columns of the matrix.\n");
	printf("row: ");
	scanf("%d", &TsMatrix->length_row);
	printf("col: ");
	scanf("%d", &TsMatrix->length_col);
	TsMatrix->length_Elem = 0;
	printf("\n");
	int data_in, count = 0;
	for (int i = 0; i < TsMatrix->length_row; i++)
	{
		printf("row %d: ", i + 1);
		TsMatrix->position[count++] = TsMatrix->length_Elem;
		for (int j = 0; j < TsMatrix->length_col; j++)
		{
			scanf("%d", &data_in);
			if (data_in)
			{
				TsMatrix->Elem[TsMatrix->length_Elem].row = i;
				TsMatrix->Elem[TsMatrix->length_Elem].col = j;
				TsMatrix->Elem[TsMatrix->length_Elem].data = data_in;
				TsMatrix->length_Elem++;
```

```
                }
            }
        }
        printf("\n");
        return TsMatrix;
}

int** matrixInit(struct TSMatrix* TsMatrix)
{
        int** matrix = (int**)malloc(sizeof(int) * TsMatrix->length_row);
        for (int i = 0; i < TsMatrix->length_row; i++)
        {
            matrix[i] = (int*)malloc(sizeof(int) * TsMatrix->length_col);
            for (int j = 0; j < TsMatrix->length_col; j++)
                matrix[i][j] = 0;
        }
        return matrix;
}

struct TsMatrix* Copy(struct TSMatrix* TsMatrix, int** matrix)
{
        for (int i = 0; i < TsMatrix->length_row; i++)
        {
            for (int j = 0; j < TsMatrix->length_col; j++)
            {
                if (matrix[i][j])
                {
                    TsMatrix->Elem[TsMatrix->length_Elem].row = i;
                    TsMatrix->Elem[TsMatrix->length_Elem].col = j;
                    TsMatrix->Elem[TsMatrix->length_Elem].data = matrix[i][j];
                    TsMatrix->length_Elem++;
                }
            }
        }
        return TsMatrix;
}

void print(struct TSMatrix* TsMatrix)
{
        int count = 0;
        for (int i = 0; i < TsMatrix->length_row; i++)
        {
            for (int j = 0; j < TsMatrix->length_col; j++)
            {
```

```
                if (TsMatrix->Elem[count].row == i && TsMatrix->Elem[count].col == j)
                {
                    printf("%4d", TsMatrix->Elem[count].data);
                    count++;
                }
                else printf("%4d", 0);
            }
            printf("\n");
        }
        count = 0;
        printf("┌──────┬──────┬──────┐\n");
        printf("│  row │  col │ data │\n");
        for (int i = 0; i < TsMatrix->length_row; i++)
        {
            for (int j = 0; j < TsMatrix->length_col; j++)
                if (TsMatrix->Elem[count].row == i && TsMatrix->Elem[count].col == j)
                {
                    printf("├──────┼──────┼──────┤\n");
                    printf("│  %3d │  %3d │  %3d │\n", TsMatrix->Elem[count].row + 1,
TsMatrix->Elem[count].col + 1, TsMatrix->Elem[count].data);
                    count++;
                }
        }
        printf("└──────┴──────┴──────┘\n");
}

struct TsMatrix* AddMatrix(struct TSMatrix* A, struct TSMatrix* B)
{
    if (A->length_row != B->length_row || A->length_col != B->length_col)return NULL;
    struct TSMatrix* C = (struct TSMatrix*)malloc(sizeof(struct TSMatrix));
    C->length_row = A->length_row;
    C->length_col = A->length_col;
    C->length_Elem = 0;
    int** matrix = matrixInit(C);
    int row, col;
    for (int i = 0; i < A->length_Elem; i++)
    {
        row = A->Elem[i].row;
        col = A->Elem[i].col;
        matrix[row][col] = matrix[row][col] + A->Elem[i].data;
    }
    for (int i = 0; i < B->length_Elem; i++)
    {
        row = B->Elem[i].row;
```

```
            col = B->Elem[i].col;
            matrix[row][col] = matrix[row][col] + B->Elem[i].data;
        }
        return Copy(C, matrix);
}

struct TsMatrix* SubMatrix(struct TSMatrix* A, struct TSMatrix* B)
{
        if (A->length_row != B->length_row || A->length_col != B->length_col)return NULL;
        struct TSMatrix* C = (struct TSMatrix*)malloc(sizeof(struct TSMatrix));
        C->length_row = A->length_row;
        C->length_col = A->length_col;
        C->length_Elem = 0;
        int** matrix = matrixInit(C);
        int row, col;
        for (int i = 0; i < A->length_Elem; i++)
        {
            row = A->Elem[i].row;
            col = A->Elem[i].col;
            matrix[row][col] = matrix[row][col] + A->Elem[i].data;
        }
        for (int i = 0; i < B->length_Elem; i++)
        {
            row = B->Elem[i].row;
            col = B->Elem[i].col;
            matrix[row][col] = matrix[row][col] - B->Elem[i].data;
        }
        return Copy(C, matrix);
}

struct TSMatrix* MultMatrix(struct TSMatrix* A, struct TSMatrix* B)
{
        if (A->length_col != B->length_row)return NULL;
        struct TSMatrix* C = (struct TSMatrix*)malloc(sizeof(struct TSMatrix));
        C->length_row = A->length_row;
        C->length_col = B->length_col;
        C->length_Elem = 0;
        if (A->length_Elem * B->length_Elem != 0)
        {
            for (int A_row = 0; A_row < A->length_row; A_row++)
            {
                int* sum = (int*)malloc(sizeof(int) * C->length_col);
                for (int C_col = 0; C_col < C->length_col; C_col++)
                    sum[C_col] = 0;
```

```
                C->position[A_row] = C->length_Elem;
                int end_A;
                if (A_row + 1 < A->length_row)end_A = A->position[A_row + 1];
                else end_A = A->length_Elem;
                for (int p = A->position[A_row]; p < end_A; p++)
                {
                        int B_row = A->Elem[p].col;
                        int end_B;
                        if (B_row + 1 < B->length_row)end_B = B->position[B_row + 1];
                        else end_B = B->length_Elem;
                        for (int q = B->position[B_row]; q < end_B; q++)
                        {
                                int C_col = B->Elem[q].col;
                                sum[C_col] += A->Elem[p].data * B->Elem[q].data;
                        }
                }
                for (int C_col = 0; C_col < C->length_col; C_col++)
                {
                        if (sum[C_col])
                        {
                                if (C->length_Elem > MAXSIZE) return NULL;
                                C->Elem[C->length_Elem].row = A_row;
                                C->Elem[C->length_Elem].col = C_col;
                                C->Elem[C->length_Elem].data = sum[C_col];
                                C->length_Elem++;
                        }
                }
        }
    }
    return C;
}

struct TSMatrix* FastTransMatrix(struct TSMatrix* A)
{
    int number[MAXSIZE];
    int position[MAXSIZE];
    struct TSMatrix* B = (struct TSMatrix*)malloc(sizeof(struct TSMatrix));
    B->length_row = A->length_col;
    B->length_col = A->length_row;
    B->length_Elem = A->length_Elem;
    if (B->length_Elem)
    {
        for (int j = 0; j < A->length_col; j++)
                number[j] = 0;
```

```
        for (int t = 0; t < A->length_Elem; t++)
            number[A->Elem[t].col]++;
        position[0] = 0;
        for (int j = 1; j < A->length_col; j++)
            position[j] = position[j - 1] + number[j - 1];
        for (int p = 0; p < A->length_Elem; p++)
        {
            int j = A->Elem[p].col;
            int q = position[j];
            B->Elem[q].row = A->Elem[p].col;
            B->Elem[q].col = A->Elem[p].row;
            B->Elem[q].data = A->Elem[p].data;
            position[j]++;
        }
    }
    return B;
}

void Involution()
{
    struct TSMatrix* TsMatrix = MatrixInit();
    TsMatrix = MultMatrix(TsMatrix, TsMatrix);
    if (TsMatrix != NULL)
    {
        printf("The result is: \n");
        print(TsMatrix);
    }
    else printf("The matrix cannot be Involuted!\n");
}

void Addition()
{
    struct TSMatrix* TsMatrix_A = MatrixInit();
    struct TSMatrix* TsMatrix_B = MatrixInit();
    struct TSMatrix* TsMatrix_C = AddMatrix(TsMatrix_A, TsMatrix_B);
    if (TsMatrix_C != NULL)
    {
        printf("The result is: \n");
        print(TsMatrix_C);
    }
    else printf("These two matrices cannot be added!\n");
}

void Subtraction()
```

```
{
	struct TSMatrix* TsMatrix_A = MatrixInit();
	struct TSMatrix* TsMatrix_B = MatrixInit();
	struct TSMatrix* TsMatrix_C = SubMatrix(TsMatrix_A, TsMatrix_B);
	if (TsMatrix_C != NULL)
	{
		printf("The result is: \n");
		print(TsMatrix_C);
	}
	else printf("These two matrices cannot be subtracted!\n");
}

void Multiplication()
{
	struct TSMatrix* TsMatrix_A = MatrixInit();
	struct TSMatrix* TsMatrix_B = MatrixInit();
	struct TSMatrix* TsMatrix_C = MultMatrix(TsMatrix_A, TsMatrix_B);
	if (TsMatrix_C != NULL)
	{
		printf("The result is: \n");
		print(TsMatrix_C);
	}
	else printf("These two matrices cannot be multiplied!\n");
}

void Transpose()
{
	struct TSMatrix* TsMatrix = MatrixInit();
	printf("Before transpose: \n");
	print(TsMatrix);
	TsMatrix = FastTransMatrix(TsMatrix);
	printf("After transpose: \n");
	print(TsMatrix);
}

void menu()
{
	char ch;
	while (true)
	{
		printf("-----functions-----\n");

	printf("[1]Involution\n[2]Addition\n[3]Subtraction\n[4]Multiplication\n[5]Transpose\n[0]Exit\n");
		printf("--------------------\n");
```

```
            printf("Please select a function: ");
            switch (ch = getchar())
            {
            case'1': system("cls"); Involution(); break;
            case'2': system("cls"); Addition(); break;
            case'3': system("cls"); Subtraction(); break;
            case'4': system("cls"); Multiplication(); break;
            case'5': system("cls"); Transpose(); break;
            case'0': return;
            default: printf("Wrong input!\n");
            }
            while (getchar() != '\n');
            system("pause");
            system("cls");
        }
}

int main()
{
        menu();
        return 0;
}
```