

目录

题目 1..... 2

 选择的题目 2

 题目功能描述 2

 问题分析与算法设计思路 2

 数据结构定义 2

 函数功能描述 3

 程序流程图 3

 程序源代码 3

 数据输入/输出截图 7

 算法核心思想 15

 图存储结构（邻接矩阵） 16

 实验心得体会 16

题目 2..... 17

 选择的题目 17

 题目功能描述 17

 问题分析与算法设计思路 17

 数据结构定义 19

 函数功能描述 20

 程序源代码 20

 数据输入/输出截图 28

 栈的存储结构 31

 实验心得体会 32

 注 32

题目 3..... 32

 选择的题目 32

 题目功能描述 32

 问题分析与算法设计思路 33

 数据结构定义 33

 函数功能描述 33

 程序流程图 错误!未定义书签。

程序源代码.....	34
数据输入/输出截图	36
顺序存储结构	37
实验心得体会	37
分工情况	38
小组成员（姓名 学号）	38
分工.....	38
成绩申报.....	38

题目 1

选择的题目

第 19 题

题目功能描述

设计一个交通咨询系统，在系统能让旅客查询从任一个城市顶点到 另一个城市顶点之间的最短路径问题。采用图来构造各个城市之间的联系，图中 顶点表示城市，边表示各个城市之间的交通关系，所带权值包括两个城市间的交 通费用、距离、时间。**（难易程度：中）**

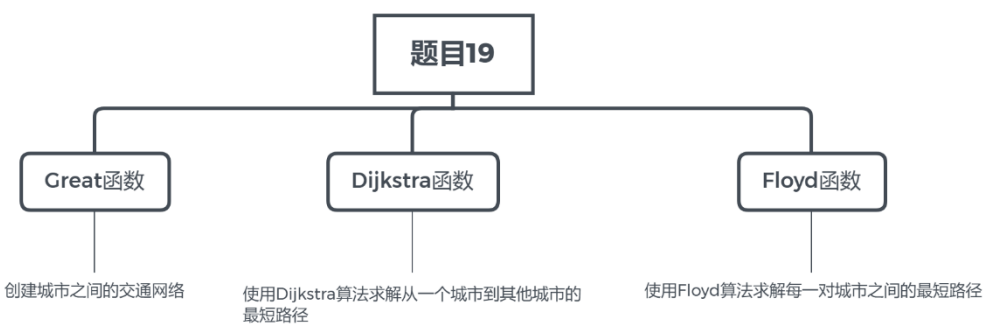
问题分析与算法设计思路

该题本质是在考察图的最短路径算法的实现，其中第 2 问和第 3 问分别是利用 Folyd 算法和 Dijkstra 算法进行求解；同时本问题中的交通网络图采用带权的无向图来进行建模实现，最终实现了从某一城市到其他城市之间的最短路径以及对于两两城市之间最短路径的求解。

数据结构定义

```
struct Graph
{
    int n;                                //本程序中，程序的标号是从一开始
    int m;
    int Matrix[233][233] = {0};
};
```

函数功能描述



函数

void Great(Graph *G)	创建城市之间的交通网络
void Dijkstra(Graph *G)	使用 Dijkstra 算法求解从一个城市到其他城市的最短路径
void Floyd(Graph *G)	使用 Floyd 算法求解每一对城市之间的最短路径

功能描述

Great 函数通过借用 Graph 的 struct 来完成交通网络的创建（包括城市结点的数量，边的数量，边的权重等，这里边的权重可以指里程，花费等，可根据实际情况输入）

Dijkstra 函数利用迪杰特斯拉算法的思想,将图中结点集合分为两组，第一组为已经求出最短路径的顶点集合，另一组为确定最短路径的顶点集合，其中 D[]用于记录初始的城市到其余各个城市的最短路径，p[]集合用于记录最短的路径中顶点 i 生成的情况，S[]集合用于表示最短的路径的生成的情况。

Floyd 函数利用弗洛伊德算法来完成对任意两城市之间最短路径的求解，Path[][]为路径的矩阵，而 D[i][j]数组则存储了从 i 城市到 j 城市的距离。

程序流程图

程序源代码

```

#include <stdio.h>
#include <stdlib.h>
#include <limits.h> //有 INT_MAX 可以直接调用整形之中最大的数
#include <iostream>
#include <string>
struct Graph
{
    int n; //本程序中，程序的标号是从一开始，而不是

```

从 0 开始

```
int m;  
int Matrix[233][233] = {0};  
};
```

```
void Great(Graph *G){  
    printf("请输入城市的数量: ");  
    scanf("%d",&(G->n));  
    printf("请输入城市之间连通路线的数量: ");  
    scanf("%d", &(G->m));  
  
    for(int i = 1;i<=233;i++){  
        for(int j = 1;j<=233;j++){  
            G->Matrix[i][j] = 10000;           //该矩阵默认全部是最大的值  
        }  
    }  
    printf("请输入该城市交通网络的邻接矩阵的形式: ");  
    for(int i = 0;i<G->m;i++){                //将边给输入好  
        int j,k;                             //j 代表横坐标, k 代表纵坐标  
        int Temp;                             //Temp 代表权值的大小  
        scanf("%d %d %d",&j,&k,&Temp);  
        G->Matrix[j][k] = Temp;  
        G->Matrix[k][j] = Temp;  
    }  
}
```

```
void Dijkstra(Graph *G){                      //从一个城市出发到其他城市的最短的路  
径
```

```
    printf("请输入你选择的城市的序号 (从 1 开始): ");  
    int D[233];  
    int p[233];                               //记录最短的路径中顶点 i 的前驱顶点  
    int s[233];                               //用于标识最短的路径的生成的情况,为 1 表示
```

从原点到 i 的最短的路径已经找到

```
    int v;  
    int k;                                    //这个 k 具体是干啥用得呢  
    scanf("%d",&v);  
    int min,max = 10000,pre;  
    for(int i = 1;i<=G->n;i++){  
        D[i] = G->Matrix[v][i];  
        if(D[i] != INT_MAX)                  //如果这里不是空的  
            p[i] = v + 1;  
        else  
            p[i] = 0;  
        s[i] = 0;
```

```

}
s[v] = 1; //将原点送到 U
for(int i = 1; i <= G->n; i++){
    min = 10001; //min > max 让最大值得那个也能加入 U
    for(int j = 1; j <= G->n; j++){
        if((!s[j]) && (D[j] < min)){
            min = D[j];
            k = j;
        }
    }
}
s[k] = 1; //将找到得顶点 k 送入 U
for(int j = 1; j <= G->n; j++){
    if((!s[j]) && (D[j] > D[k] + G->Matrix[k][j]))
    {
        D[j] = D[k] + G->Matrix[k][j];
        p[j] = k+1; //k 是 j 的前驱
    }
}
//所有的顶点已经扩充到了 U 中
}
for(int k = 1; k <= G->n; k++){
    if(k == v && k < 10){
        printf(" 0 %10d", k);
    }else if(k == v && k >= 10){
        printf(" 0 %11d", k);
    }else if(D[k] == 10000 && k < 10){
        printf(" ∞ %10d", k);
    }else if(D[k] == 10000 && k >= 10){
        printf(" ∞ %11d", k);
    }else{
        printf(" %-10d %d", D[k], k);
    }
    pre = p[k];
    if(k != v){
        while((pre != 0) && (pre != v+1)){
            printf("<_%d", pre-1);
            pre = p[pre-1];
        }
        printf("<_%d", v);
    }
    printf("\n");
}
}

```

```

void Floyd(Graph *G){
    printf("接下来输出每一对城市之间的最短路径: \n");
}

```

```

int Path[233][233]; //路径矩阵
int D[233][233];
int i,j,k,pre;
int w,max = 10000;
for(i = 0;i<=G->n;i++){
    for(j = 0;j<=G->n;j++){
        if(G->Matrix[i][j] != max){
            Path[i][j] = i+1; //i 是 j 得前驱
        }else{
            Path[i][j] = 0;
        }
        D[i][j] = G->Matrix[i][j];
    }
}
for(k = 1;k<=G->n;k++){
    for(i = 1;i<=G->n;i++){
        for(j = 1;j<=G->n;j++){
            if(D[i][j] > (D[i][k] + D[k][j])){
                D[i][j] = D[i][k] + D[k][j]; //修改路径的长度
                Path[i][j] = Path[k][j]; //修改路径
            }
        }
    }
}
for(i = 1;i<=G->n;i++){
    printf("第%d 组: \n",i);
    for(j = 1;j<=G->n;j++){
        if(i == j && j < 10){
            printf(" 0 %10d",j);
        }else if(i == j && j >= 10){
            printf(" 0 %11d",j);
        }else if(D[i][j] == 10000 && j < 10){
            printf(" ∞ %10d",j);
        }else if(D[i][j] == 10000 && j >= 10){
            printf(" ∞ %11d",j);
        }else{
            printf(" %-10d %d",D[i][j],j);
        }
        pre = Path[i][j];
        if(i!=j){
            while((pre != 0) && (pre != i+1)){
                printf("<_%d",pre-1);
                pre = Path[i][pre-1];
            }
        }
    }
}

```

```

    }
    printf("<_%d\n",i);
}
}
}

```

```

int main(void){
    Graph G;
    Great(&G);
    Dijkstra(&G);
    Floyd(&G);
    return 0;
}

```

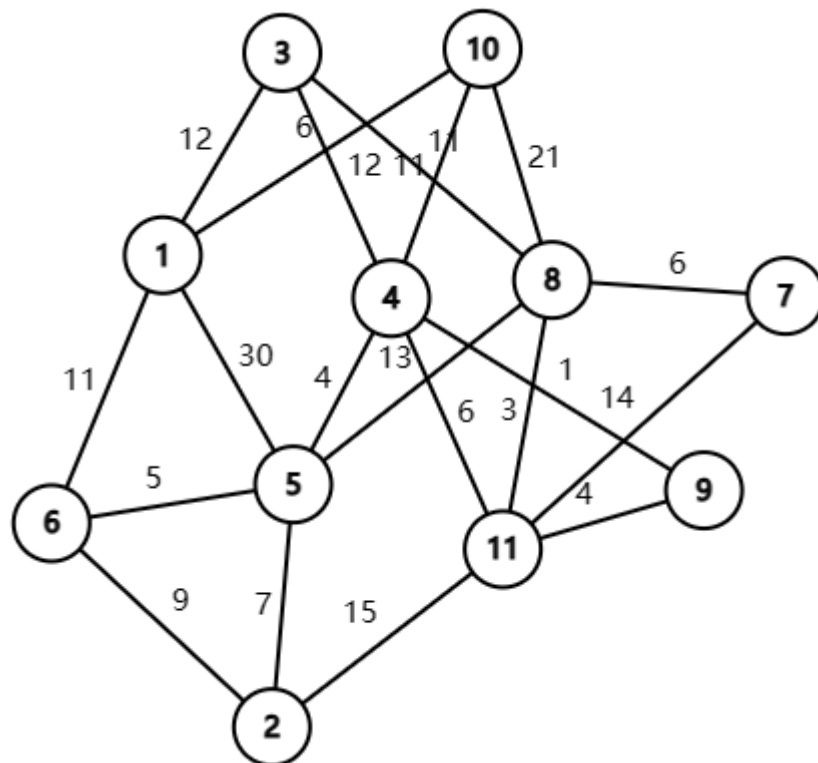
数据输入/输出截图

测试样例 1:

Graph1

该图由 11 个结点和 20 条边，属于交通网络中较复杂的情况

、



输入输出如下：

```
请输入该城市交通网络的邻接矩阵的形式：1 10 6
1 6 11
1 3 12
1 5 30
2 5 7
2 6 9
2 11 15
4 11 6
4 9 1
4 3 12
4 10 11
3 8 11
4 5 4
5 6 5
8 11 3
7 8 6
5 8 13
8 10 21
9 11 4
11 7 14
请输入你选择的城市的序号（从1开始）：2
20      1<_6<_2
0        2
23      3<_4<_5<_2
11      4<_5<_2
7        5<_2
9        6<_2
24      7<_8<_11<_2
18      8<_11<_2
12      9<_4<_5<_2
22      10<_4<_5<_2
15      11<_2
```

由于求任意两城市最短路径的输出太长，故以文本形式给出
接下来输出每一对城市之间的最短路径：

第 1 组：

0	1<_1
20	2<_6<_1
12	3<_1
17	4<_10<_1
16	5<_6<_1
11	6<_1
29	7<_8<_3<_1
23	8<_3<_1
18	9<_4<_10<_1
6	10<_1
22	11<_9<_4<_10<_1

第 2 组：

20	$1 <_6 <_2$
0	$2 <_2$
23	$3 <_4 <_5 <_2$
11	$4 <_5 <_2$
7	$5 <_2$
9	$6 <_2$
24	$7 <_8 <_{11} <_2$
18	$8 <_{11} <_2$
12	$9 <_4 <_5 <_2$
22	$10 <_4 <_5 <_2$
15	$11 <_2$

第 3 组:

12	$1 <_3$
23	$2 <_5 <_4 <_3$
0	$3 <_3$
12	$4 <_3$
16	$5 <_4 <_3$
21	$6 <_5 <_4 <_3$
17	$7 <_8 <_3$
11	$8 <_3$
13	$9 <_4 <_3$
18	$10 <_1 <_3$
14	$11 <_8 <_3$

第 4 组:

17	$1 <_{10} <_4$
11	$2 <_5 <_4$
12	$3 <_4$
0	$4 <_4$
4	$5 <_4$
9	$6 <_5 <_4$
14	$7 <_8 <_{11} <_9 <_4$
8	$8 <_{11} <_9 <_4$
1	$9 <_4$
11	$10 <_4$
5	$11 <_9 <_4$

第 5 组:

16	$1 <_6 <_5$
7	$2 <_5$
16	$3 <_4 <_5$
4	$4 <_5$
0	$5 <_5$
5	$6 <_5$
18	$7 <_8 <_{11} <_9 <_4 <_5$
12	$8 <_{11} <_9 <_4 <_5$

5	$9 <_4 <_5$
15	$10 <_4 <_5$
9	$11 <_9 <_4 <_5$

第 6 组:

11	$1 <_6$
9	$2 <_6$
21	$3 <_4 <_5 <_6$
9	$4 <_5 <_6$
5	$5 <_6$
0	$6 <_6$
23	$7 <_8 <_{11} <_9 <_4 <_5 <_6$
17	$8 <_{11} <_9 <_4 <_5 <_6$
10	$9 <_4 <_5 <_6$
17	$10 <_1 <_6$
14	$11 <_9 <_4 <_5 <_6$

第 7 组:

29	$1 <_3 <_8 <_7$
24	$2 <_{11} <_8 <_7$
17	$3 <_8 <_7$
14	$4 <_9 <_{11} <_8 <_7$
18	$5 <_4 <_9 <_{11} <_8 <_7$
23	$6 <_5 <_4 <_9 <_{11} <_8 <_7$
0	$7 <_7$
6	$8 <_7$
13	$9 <_{11} <_8 <_7$
25	$10 <_4 <_9 <_{11} <_8 <_7$
9	$11 <_8 <_7$

第 8 组:

23	$1 <_3 <_8$
18	$2 <_{11} <_8$
11	$3 <_8$
8	$4 <_9 <_{11} <_8$
12	$5 <_4 <_9 <_{11} <_8$
17	$6 <_5 <_4 <_9 <_{11} <_8$
6	$7 <_8$
0	$8 <_8$
7	$9 <_{11} <_8$
19	$10 <_4 <_9 <_{11} <_8$
3	$11 <_8$

第 9 组:

18	$1 <_{10} <_4 <_9$
12	$2 <_5 <_4 <_9$
13	$3 <_4 <_9$
1	$4 <_9$

5	$5 <_4 <_9$
10	$6 <_5 <_4 <_9$
13	$7 <_8 <_{11} <_9$
7	$8 <_{11} <_9$
0	$9 <_9$
12	$10 <_4 <_9$
4	$11 <_9$
第 10 组:	
6	$1 <_{10}$
22	$2 <_5 <_4 <_{10}$
18	$3 <_1 <_{10}$
11	$4 <_{10}$
15	$5 <_4 <_{10}$
17	$6 <_1 <_{10}$
25	$7 <_8 <_{11} <_9 <_4 <_{10}$
19	$8 <_{11} <_9 <_4 <_{10}$
12	$9 <_4 <_{10}$
0	$10 <_{10}$
16	$11 <_9 <_4 <_{10}$
第 11 组:	
22	$1 <_{10} <_4 <_9 <_{11}$
15	$2 <_{11}$
14	$3 <_8 <_{11}$
5	$4 <_9 <_{11}$
9	$5 <_4 <_9 <_{11}$
14	$6 <_5 <_4 <_9 <_{11}$
9	$7 <_8 <_{11}$
3	$8 <_{11}$
4	$9 <_{11}$
16	$10 <_4 <_9 <_{11}$
0	$11 <_{11}$

测试样例二：

该图有 10 个结点，以及 12 条边

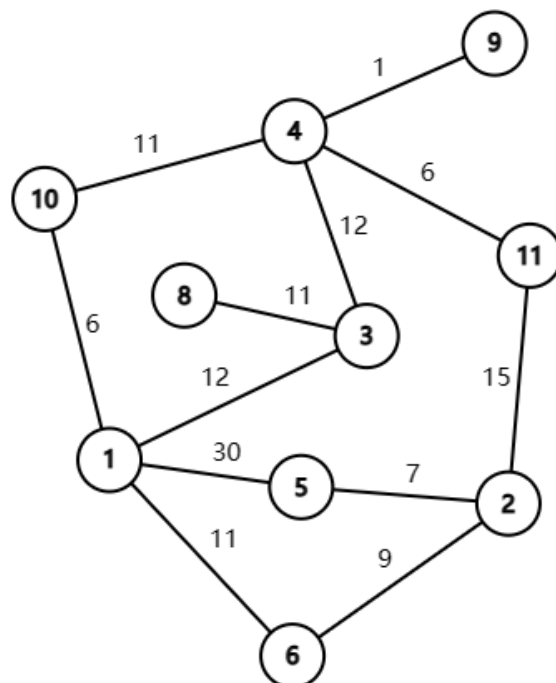
```
请输入城市的数量：10
请输入城市之间连通路线的数量：12
请输入该城市交通网络的邻接矩阵的形式：1 10 6
1 6 11
1 3 12
1 5 30
2 5 7
2 6 9
2 11 15
4 11 6
4 9 1
4 3 12
4 10 11
3 8 11
请输入你选择的城市的序号（从1开始）：3
12      1<_3
32      2<_6<_1<_3
0       3
12      4<_3
39      5<_2<_6<_1<_3
23      6<_1<_3
∞       7<_3
11      8<_3
13      9<_4<_3
18      10<_1<_3
```

输入输出如下：

接下来输出每一对城市之间的最短路径：

第 1 组：

0	1<_1
20	2<_6<_1
12	3<_1
17	4<_10<_1
27	5<_2<_6<_1
11	6<_1
∞	7<_1
23	8<_3<_1



18	$9<_4<_{10}<_1$
6	$10<_1$
第 2 组:	
20	$1<_6<_2$
0	$2<_2$
32	$3<_1<_6<_2$
37	$4<_{10}<_1<_6<_2$
7	$5<_2$
9	$6<_2$
∞	$7<_2$
43	$8<_3<_1<_6<_2$
38	$9<_4<_{10}<_1<_6<_2$
26	$10<_1<_6<_2$
第 3 组:	
12	$1<_3$
32	$2<_6<_1<_3$
0	$3<_3$
12	$4<_3$
39	$5<_2<_6<_1<_3$
23	$6<_1<_3$
∞	$7<_3$
11	$8<_3$
13	$9<_4<_3$
18	$10<_1<_3$
第 4 组:	
17	$1<_{10}<_4$
37	$2<_6<_1<_{10}<_4$
12	$3<_4$
0	$4<_4$
44	$5<_2<_6<_1<_{10}<_4$
28	$6<_1<_{10}<_4$
∞	$7<_4$
23	$8<_3<_4$
1	$9<_4$
11	$10<_4$
第 5 组:	
27	$1<_6<_2<_5$
7	$2<_5$
39	$3<_1<_6<_2<_5$
44	$4<_{10}<_1<_6<_2<_5$
0	$5<_5$
16	$6<_2<_5$
∞	$7<_5$
50	$8<_3<_1<_6<_2<_5$

45 $9 <_4 10 <_1 6 <_2 5$

33 $10 <_1 6 <_2 5$

第 6 组:

11 $1 <_6$

9 $2 <_6$

23 $3 <_1 6$

28 $4 <_{10} 1 <_6$

16 $5 <_2 6$

0 $6 <_6$

∞ $7 <_6$

34 $8 <_3 1 <_6$

29 $9 <_4 10 <_1 6$

17 $10 <_1 6$

第 7 组:

∞ $1 <_7$

∞ $2 <_7$

∞ $3 <_7$

∞ $4 <_7$

∞ $5 <_7$

∞ $6 <_7$

0 $7 <_7$

∞ $8 <_7$

∞ $9 <_7$

∞ $10 <_7$

第 8 组:

23 $1 <_3 8$

43 $2 <_6 1 <_3 8$

11 $3 <_8$

23 $4 <_3 8$

50 $5 <_2 6 <_1 3 <_8$

34 $6 <_1 3 <_8$

∞ $7 <_8$

0 $8 <_8$

24 $9 <_4 3 <_8$

29 $10 <_1 3 <_8$

第 9 组:

18 $1 <_{10} 4 <_9$

38 $2 <_6 1 <_{10} 4 <_9$

13 $3 <_4 9$

1 $4 <_9$

45 $5 <_2 6 <_1 10 <_4 9$

29 $6 <_1 10 <_4 9$

∞ $7 <_9$

24 $8 <_3 4 <_9$

0	9<_9
12	10<_4<_9
第 10 组:	
6	1<_10
26	2<_6<_1<_10
18	3<_1<_10
11	4<_10
33	5<_2<_6<_1<_10
17	6<_1<_10
∞	7<_10
29	8<_3<_1<_10
12	9<_4<_10
0	10<_10

算法核心思想

迪杰斯特拉(Dijkstra)算法

Dijkstra(迪杰斯特拉)算法是典型的单源最短路径算法，用于计算一个节点到其他所有节点的最短路径。主要特点是以起始点为中心向外层层扩展，直到扩展到终点为止。

算法步骤：

- 将所有的顶点分为两部分：已知最短路径的顶点集合 P 和未知最短路径的顶点集合 Q。最开始，已知最短路径的顶点集合 P 中只有源点一个顶点。我们这里用一个 $S[i]$ 数组来记录哪些点在集合 P 中。例如对于某个顶点 i，如果 $S[i]$ 为 1 则表示这个顶点在集合 P 中，如果 $S[i]$ 为 0 则表示这个顶点在集合 Q 中。（注：此处 P 和 Q 均未出现在程序中，其出现是为了更好的阐述算法思想）
- 设置源点 A 到自己的最短路径为 0 即 $D[i]=0$ （i 为 A 的编号）。若存在源点有能直接到达的顶点 i，则把 $D[i]$ 设为 $G \rightarrow Matrix[s][i]$ 。同时把所有其它（源点不能直接到达的）顶点的最短路径设为 ∞ 。
- 在集合 Q 的所有顶点中选择一个离源点 s 最近的顶点 u（即 $dis[u]$ 最小）加入到集合 P。并考察所有以点 u 为起点的边，对每一条边进行松弛操作。例如存在一条从 u 到 v 的边，那么可以通过将边 $u \rightarrow v$ 添加到尾部来拓展一条从 s 到 v 的路径，这条路径的长度是 $D[u] + G \rightarrow Matrix[u][v]$ 。如果这个值比目前已知的 $D[v]$ 的值要小，我们可以用新值来替代当前 $dis[v]$ 中的值。
- 重复第 3 步，如果集合 Q 为空，算法结束。最终 dis 数组中的值就是源点到所有顶点的最短路径。

弗洛伊德(Floyd)算法

Floyd 算法是解决任意两点间的最短路径的一种算法，可以正确处理有向图或负权的最短路径问题，同时也被用于计算有向图的传递闭包。Floyd 算法的时间复杂度为 $O(n^3)$ 。

算法步骤：

- 从任意一条单边路径开始。所有两点之间的距离是边的权，如果两点之间没有边相连，则权为无穷大。
- 对于每一对顶点 u 和 v ，看看是否存在一个顶点 w 使得从 u 到 w 再到 v 比已知的路径更短。如果是更新它。

关于第三问的算法实现分析

第二问中使用的迪杰斯特拉(Dijkstra)算法其实也可以用于第三问中求任意两对顶点之间的最短路径问题，但是一般多源的最短路径还是由弗洛伊德(Floyd)算法去求解

本文在此分析了下原因：

Dijkstra 时间复杂度为 $O(n^2)$ 空间复杂度为 $O(n^2)$

Floyd 时间复杂度为 $O(n^3)$ 空间复杂度为 $O(n^2)$

故在求取单源问题时候一般倾向于使用 Dijkstra，若是将 Dijkstra 中的存取最短路径 $D[]$ 数组拓展为 2 维情况时候，其时间复杂度和空间复杂度都会去乘以 n ，最终导致时间复杂度与 Floyd 相同的情况下，空间复杂度不如 Floyd。

综上，故针对任意两对结点之间的最短路径的求取，Floyd 仍是最主流的算法。

图存储结构（邻接矩阵）

优缺点

容易获得每个边的权值，取得第 i 条边的权值的时间复杂度仅为 $O(1)$ 。

同时能较方便地实现本程序中迪杰斯特拉算法与弗洛伊德算法。

缺点为对于边数相对顶点较少的图，浪费了极大的存储空间，是典型地以空间换时间的存储形式。

采用邻接矩阵的图存储结构的理由

存取速度高效,通过下标来直接存储。

无需为表示结点间的逻辑关系而增加额外的存储空间。

比起其他的存储结构，能更方便地实现解决问题所需的算法。

实验心得体会

深化了我对于求取最短路径的 Floyd 和 Dijkstra 两大算法的理解；

在程序实现的过程中，强化了记忆，提高了自己的编程能力；

此外，在探索第三问其他的求解算法的过程中，我对于两种算法的复杂度的讨论有了更为深刻的理解和认识；

综上，很有幸能参与这次数据结构大作业，感谢这次大作业为我带来的能力上的提升。

题目 2

选择的题目

T4 中缀表达式进行表达式求值（中等难度）

题目功能描述

本程序可对输入的中序表达式求值。

具体功能可分为：

1. 读入表达式
2. 判断表达式是否有误
3. 如无误，则将表达式转化为后缀表达式
4. 对后缀表达式进行求值

问题分析与算法设计思路

问题分析

该问题可分为三部分：

1. 对读入的中缀表达式判断是否有误
2. 将中缀表达式转化为后缀表达式
3. 对后缀表达式进行求值运算

对应的算法设计思路

判断读入的中缀表达式是否有误

表达式的错误可分为四种：

1. 格式与题目给定不符合：未按照"#·····（数学表达式）#"的格式进行输入
2. 缺失操作数：两运算符之间没有数字，操作数缺失：出现了"6+ -3","2*/4", "2/-4"等的情况
3. 非法算符：出现了除数字（0~9），运算符（'+','-','*','/'（'（'）'）以外的字符
4. 括号不配匹配：左括号或右括号多余

本题本别设计种算法，分别对对应的错误进行判断：

1.判断输入字符串的首尾元素是否为'#'：

同时取读入字符串的首尾元素，若不通时等于'#'则不符合输入规范

2.缺失运算数：设置 int lose(char*a)函数：

设置标志变量 key=0，并从字符串首开始逐字遍历字符串：

若为运算符则 key++；

若为数字则 key=0；

若 key 超出 1 则发生运算数缺失

3.非法算符：设置 illegal(char*a)函数：

逐字遍历字符串，若发现非法字符则含有非法字符

4.括号不匹配：设置 kuohao (char*a) 函数：

设置标志变量 key=0，并从字符串首开始逐字遍历字符串：

若为'('则 key++；

若为')'则 key--；

若 key<0 则左括号缺失

遍历完后：

若 key! =0 则右括号缺失

将中缀表达式转化为后缀表达式

逐字遍历字符串，服从以下规则：

1、字符为 运算数：

直接送入后缀表达式（注：需要先分析出完整的运算数）。

2、字符为 左括号：

直接入栈（注：左括号入栈后优先级降至最低）。

3、字符为 右括号：

直接出栈，并将出栈字符依次送入后缀表达式，直到栈顶字符为左括号（左括号也要出栈，但不送入后缀表达式）。

总结：只要满足 栈顶为左括号 即可进行最后一次出栈。

4、字符为 操作符：

若栈空，直接入栈。

若栈非空，判断栈顶操作符，若栈顶操作符优先级低于该操作符，该操作符入栈；否则一直

出栈，并将出栈字符依次送入后缀表达式，直到栈空或栈顶操作符优先级低于该操作符，该操作符再入栈。

对后缀表达式进行求值运算

从左至右依次遍历后缀表达式各个字符，服从以下规则：

1、字符为 运算数：

直接入栈（注：需要先分析出完整的运算数并将其转换为对应的数据类型）

2、字符为 操作符：

连续出栈两次，使用出栈的两个数据进行相应计算，并将计算结果入栈

3、重复以上步骤直至遍历完成后缀表达式，最后栈中的数据就是中缀表达式的计算结果。

数据结构定义

数字栈：

定义两个整数指针 top，base 分别指向栈顶元素与栈底元素的后一元素

当出栈时：top 后移

当入栈时：top 前移

当 top == base 时，栈为空

```
typedef struct {  
    int *base; //用于栈存储的基地址  
    int *top; //指向该基地址的栈顶指针  
    int stackSize; //栈的大小  
}SqStackInt;
```

运算符栈：

定义两个字符指针 top，base 分别指向栈顶元素与栈底元素的后一元素

当出栈时：top 后移

当入栈时：top 前移

当 top == base 时，栈为空

```
typedef struct {  
    char *base; //用于栈存储的基地址  
    char *top; //指向该基地址的栈顶指针  
    int stackSize; //栈的大小  
}SqStackChar;
```

函数功能描述

int InitStack_Int(SqStackInt &S)
建立空栈 (for 数字)
int InitStack_Char(SqStackChar &S)
建立空栈 (for 运算符)
int Push_Int(SqStackInt &S,int e)
数字入数字栈
int Push_Char(SqStackChar &S,char e)
运算符入运算符栈
int Pop_Int(SqStackInt &S,int &e)
数字栈弹出首元素
int Pop_Char(SqStackChar &S,char &e)
字符栈弹出首元素
int StackEmpty_Int(SqStackInt S)
判断数字栈是否为空
int StackEmpty_Char(SqStackChar S)
判断运算符栈是否为空
int ClearStack_Int(SqStackInt S)
清空数字栈剩余的所有元素
int ClearStack_Char(SqStackChar S)
清空运算符栈中的所有元素
int DestroyStack_Int(SqStackInt &S)
运行结束后将数字栈销毁
int DestroyStack_Char(SqStackChar &S)
运行结束后将运算符栈销毁
int isOper(char c)
判断该字符是否为运算符 (还是数字)
char getStackTopPriority(char StackTop,char c)
判断运算符栈顶元素与当前运算符的优先级大小
int operate(int a,char oper,int b)
根据得到的运算符与两数字, 进行计算

程序源代码

```
#include<stdio.h>
#include<stdlib.h>
#include<stdlib.h>
#include<sys/malloc.h>
#include<stdio.h>

#define MAXSIZE 100
```

```

/*定义顺序栈*/
typedef struct {
    int *base; //用于栈存储的基地址
    int *top; //指向该基地址的栈顶指针
    int stackSize; //栈的大小
}SqStackInt;
/*定义顺序栈*/
typedef struct {
    char *base; //用于栈存储的基地址
    char *top; //指向该基地址的栈顶指针
    int stackSize; //栈的大小
}SqStackChar;
/*初始化*/
int InitStack_Int(SqStackInt &S){
    S.base = (int *)malloc(MAXSIZE*sizeof(int)); //给基地址分配一个内存空间
    S.top = S.base; //将栈顶指针指向这个基地址
    S.stackSize = MAXSIZE; //设置栈的大小
    return 0;
}
int InitStack_Char(SqStackChar &S){
    S.base = (char *)malloc(MAXSIZE*sizeof(char)); //给基地址分配一个内存空间
    S.top = S.base; //将栈顶指针指向这个基地址
    S.stackSize = MAXSIZE; //设置栈的大小
    return 0;
}
/*进栈*/
int Push_Int(SqStackInt &S,int e){
    if(S.top-S.base==S.stackSize) return -1;
    *S.top = e; //将输入的值压入栈中
    S.top++; //指针上移一个单位

    return 0;
}
int Push_Char(SqStackChar &S,char e){
    if(S.top-S.base==S.stackSize) return -1;
    *S.top = e; //将输入的值压入栈中
    S.top++; //指针上移一个单位

    return 0;
}
/*出栈*/
int Pop_Int(SqStackInt &S,int &e) {
    if(S.base==S.top) return -1;
    S.top--; //指针下移一个

```

```

        e = *S.top; //将当前指针所指的值赋值给 e

        return 0;
    }
int Pop_Char(SqStackChar &S,char &e) {
    if(S.base==S.top) return -1;

    S.top--; //指针下移一个
    e = *S.top; //将当前指针所指的值赋值给 e

    return 0;
}
/*获取栈的长度*/
int GetLength_Int(SqStackInt S){
    return S.top-S.base;
}
int GetLength_Char(SqStackChar S){
    return S.top-S.base;
}
/*判断栈空*/
int StackEmpty_Int(SqStackInt S) {
    if(S.top==S.base) return 0; //为空返回 0
    return 1; //不为空返回 1
}
int StackEmpty_Char(SqStackChar S) {
    if(S.top==S.base) return 0; //为空返回 0
    return 1; //不为空返回 1
}
/*清空栈*/
int ClearStack_Int(SqStackInt S){
    if(S.base) //栈不为空
        S.base = S.top;
    return 0;
}
int ClearStack_Char(SqStackChar S){
    if(S.base) //栈不为空
        S.base = S.top;
    return 0;
}
/*销毁栈*/
int DestroyStack_Int(SqStackInt &S){
    if(S.base){
        free(S.base);
    }
}

```

```

        S.stackSize = 0;
        S.top = S.base = NULL;
    }
    return 0;
}

int DestroyStack_Char(SqStackChar &S){
    if(S.base){
        free(S.base);
        S.stackSize = 0;
        S.top = S.base = NULL;
    }
    return 0;
}

/*读取栈顶元素*/
int GetTop_Int(SqStackInt S) {
    return *(S.top-1);
}

char GetTop_Char(SqStackChar S) {
    return *(S.top-1);
}

}

int isOper(char c){
    if(c == '#' || c == '+' || c == '-' || c == '*' || c == '/' || c == '(' || c == '){
        return 1; //是操作符
    }else{
        return 0; //不是操作符
    }
}

/*
* c1:栈顶操作符
* c2:扫描操作符
*
*/

char getStackTopPriority(char StackTop,char c){
    //printf("Priority:%c::%c\n",StackTop,c);
    if(StackTop == '#' || (StackTop == '(' && c != ')') || c == '(' || (StackTop == '+' || StackTop == '-' )&&((c == '*' || StackTop == '*')))) {
        return '<'; //栈顶操作符优先级 小于等于 当前扫描操作符则 操作符进栈
    }else if(StackTop == '(' && c == '){
        return '=';
    }else if(c == '*' || c == '/' || c == '){
        return '>';
    }
}

```

```

    }else{
        return '>';
    }

}

/*
*从 S1 栈中弹出两个操作数 a 和 b
*从 S2 栈中弹出一个操作符 oper
* 然后两个操作数和一个操作符进行运算
*/
int operate(int a,char oper,int b){
    if(oper == '+'){
        return a+b;
    }else if(oper == '-'){
        return a-b;
    }else if(oper == '*'){
        return a*b;
    }else {
        return a/b;
    }
}

/*
错误可能有以下几个：
缺失操作数
非法算符
括号不配匹配
*/
int kuohao(char*a){
    int z = 1;
    int key = 0;
    char ch = a[z - 1];
    while (ch!='#'){

        if(ch=='(')
            key++;
        if(ch==')'){
            key--;
        }
        if(key<0){
            printf("括号不匹配（右括号多余）\n");
            return 1;
        }
        ch = a[z++];
    }
}

```



```

        if(key!=0){
            printf("括号不匹配（左括号多余）\n");
            return 1;
        }
        return 0;
    }

int illegal(char*a){
    int z = 1;
    char ch = a[z-1];
    while(ch!='#'){
        if(!(ch<('9'+1)&&ch>('0'-1))||ch=='+'||ch=='-'||ch=='*'||ch=='/'||ch=='('||ch==')){
            printf("非法算符\n");
            return 1;
        }
        ch = a[z++];
    }
    return 0;
}

int lose(char*a){
    int z = 1,key=0;
    char ch = a[z-1];
    while(ch!='#'){
        if(ch=='*'||ch=='/'||ch=='+'||ch=='-'){
            key++;
        }
        if(ch<='9'&&ch>='0')
            key = 0;

        if (key == 2)
        {
            printf("运算数缺失\n");
            return 1;
        }

        ch = a[z++];
    }
    return 0;
}

int wrong(char *a)

```

```

{
    if(kuohao(a)||illegal(a)||lose(a))
        return 1;
    return 0;
}

int main(){
    char arr[100] = {'\0'},k;
    int key = 0;
    char ch;
    k = getchar();
    if(k!='#'){
        printf("格式错误! ");
        return 0;
    }

    while ((ch = getchar()) != '\n')
    {
        arr[key++] = ch;
    }
    if(arr[key-1]!='#'){
        printf("格式错误! ");
        return 0;
    }

    key = 0;

    while (arr[key] != '#')
    {
        printf("%c", arr[key]);
        key++;
    }

    if(wrong(arr)){
        return 0;
    }
    SqStackInt S1; //用来存储操作数的栈 int 类型
    SqStackChar S2;
    //初始化两个栈
    InitStack_Int(S1);
    InitStack_Char(S2);
    //操作符栈 进栈一个 # 号作为结束标志
    Push_Char(S2,'#');
    int i = 0; //用于循环遍历 中缀表达式 arr 数组

```

```

while(arr[i]!='#'||GetTop_Char(S2)!='#'){

    if(!isOper(arr[i])){ //如果不是操作符
        int e = arr[i] - '0';
        Push_Int(S1,e); //进操作数 S1 栈
        i++;
    }else{ //是操作符等待进入 S2 栈
        char e = arr[i];
        //比较操作符 S2 栈 当前栈顶操作符的和当前扫描到的操作符优先级大
        小

        switch (getStackTopPriority(GetTop_Char(S2),e)){

            case '<':{ //栈顶操作符优先级小-->
                Push_Char(S2,e);
                i++;
                break;
            }

            case '=':{
                char x;
                Pop_Char(S2,x);
                i++;
                break;
            }

            case '>':{
                int a,b;char oper;
                Pop_Int(S1,b);Pop_Int(S1,a);
                Pop_Char(S2,oper);
                int e = operate(a,oper,b);
                Push_Int(S1,e);

                break;
            }

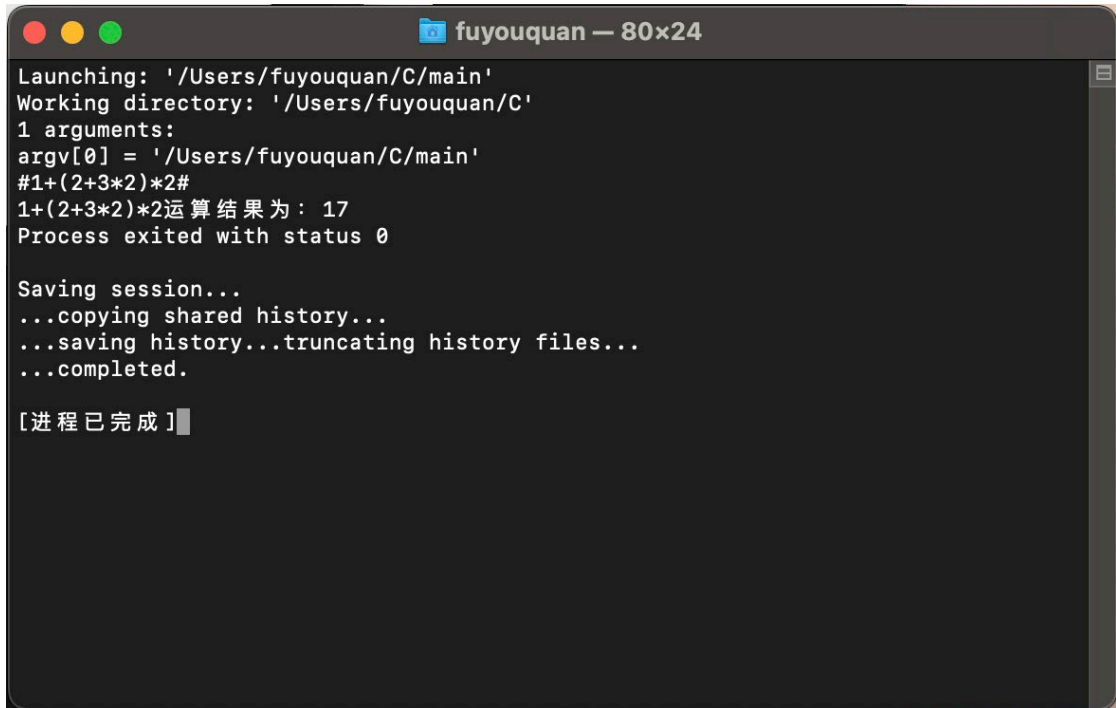
        }
    }
}

printf("运算结果为: %d\n",GetTop_Int(S1));
return 0;
}

```

数据输入/输出截图

正确运行截图：（含有括号，乘除，加减等操作）



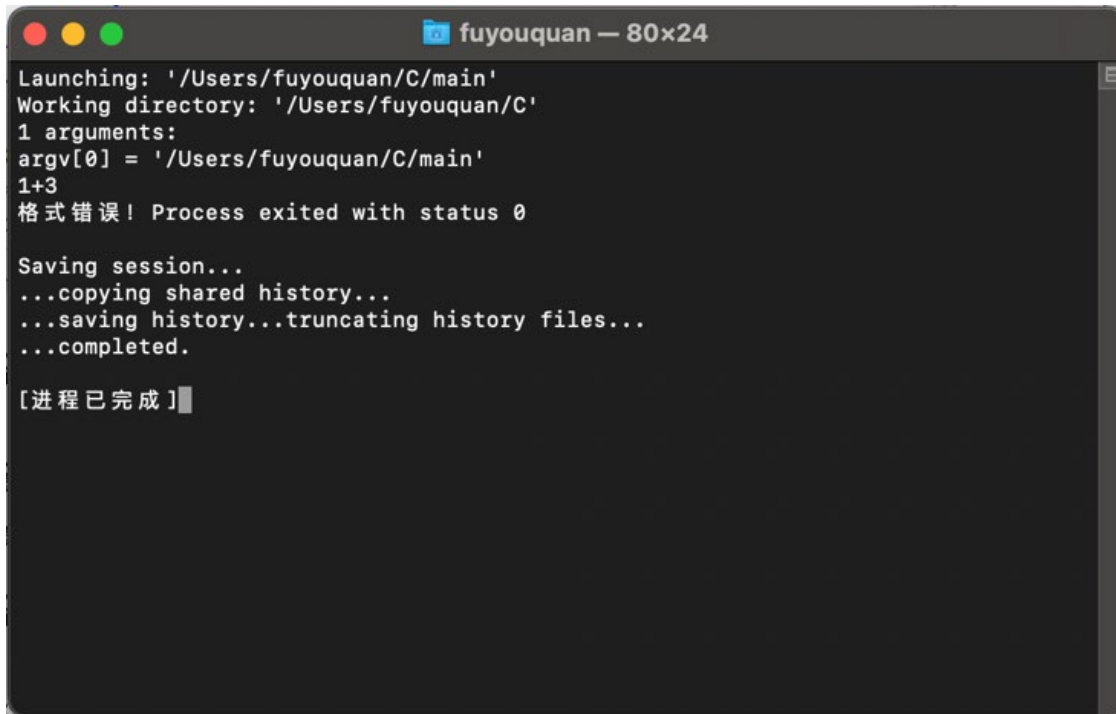
```
Launching: '/Users/fuyouquan/C/main'
Working directory: '/Users/fuyouquan/C'
1 arguments:
argv[0] = '/Users/fuyouquan/C/main'
#1+(2+3*2)*2#
1+(2+3*2)*2运算结果为：17
Process exited with status 0

Saving session...
...copying shared history...
...saving history...truncating history files...
...completed.

[进程已完成]
```

格式错误响应截图：

1. 输入格式错误：

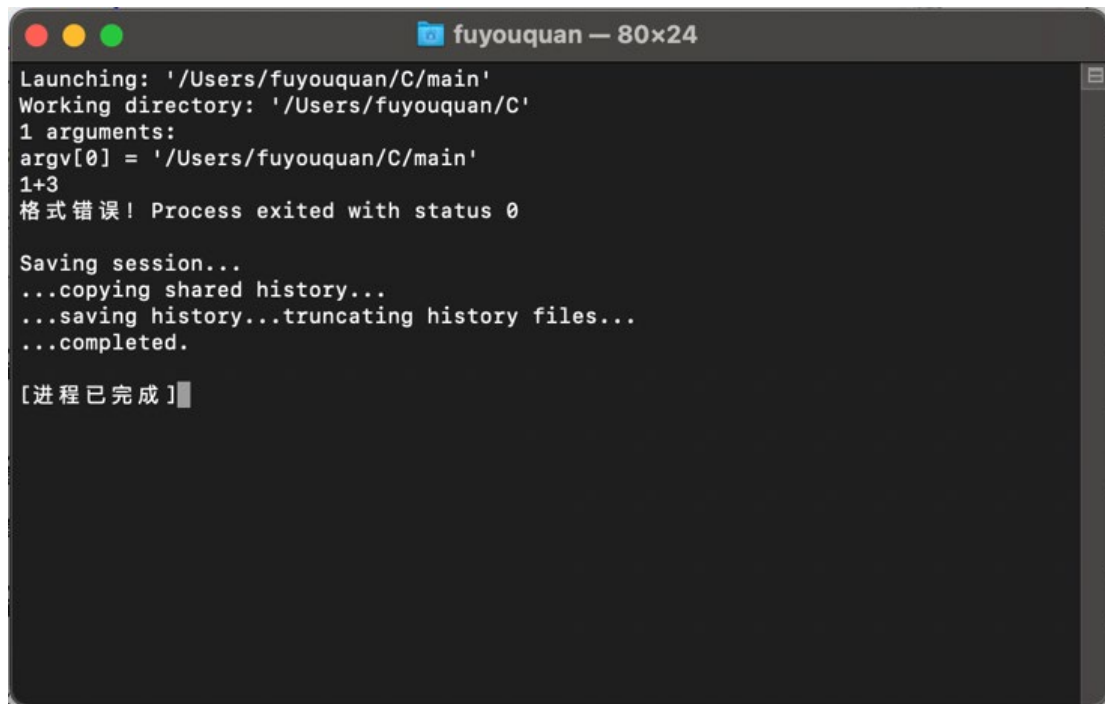


```
Launching: '/Users/fuyouquan/C/main'
Working directory: '/Users/fuyouquan/C'
1 arguments:
argv[0] = '/Users/fuyouquan/C/main'
1+3
格式错误！ Process exited with status 0

Saving session...
...copying shared history...
...saving history...truncating history files...
...completed.

[进程已完成]
```

2. 括号不匹配（左括号多余）：

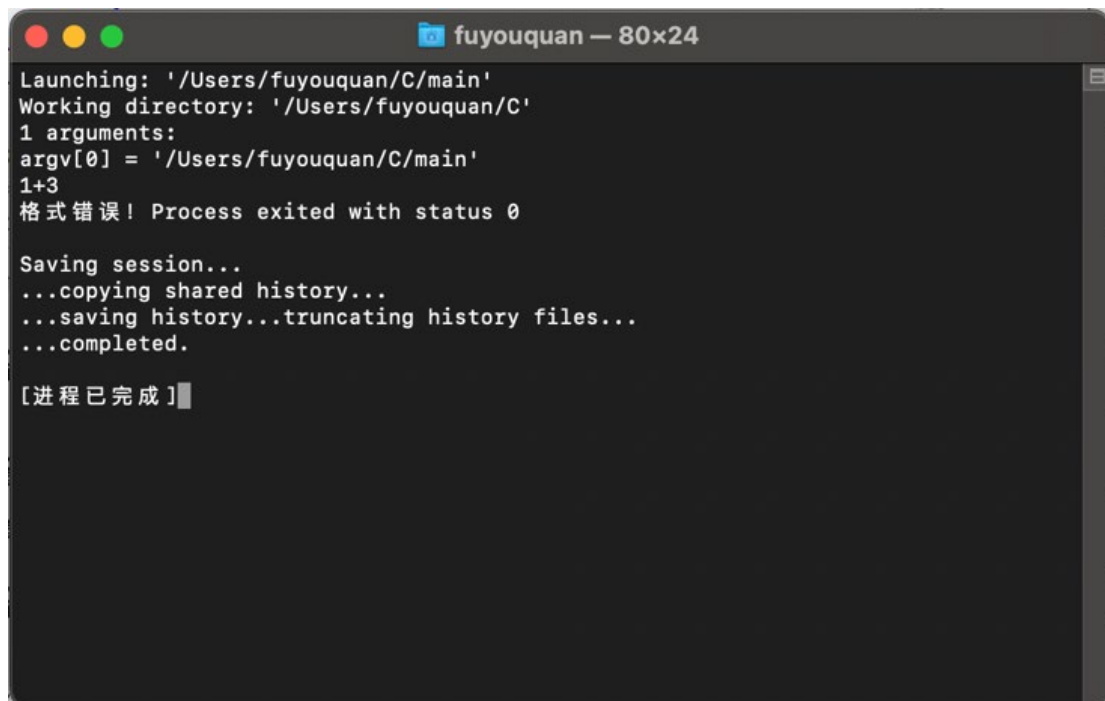


```
fyouquan — 80x24
Launching: '/Users/fyouquan/C/main'
Working directory: '/Users/fyouquan/C'
1 arguments:
argv[0] = '/Users/fyouquan/C/main'
1+3
格式错误! Process exited with status 0

Saving session...
...copying shared history...
...saving history...truncating history files...
...completed.

[进程已完成]
```

3. 括号不匹配 (右括号多余):



```
fyouquan — 80x24
Launching: '/Users/fyouquan/C/main'
Working directory: '/Users/fyouquan/C'
1 arguments:
argv[0] = '/Users/fyouquan/C/main'
1+3
格式错误! Process exited with status 0

Saving session...
...copying shared history...
...saving history...truncating history files...
...completed.

[进程已完成]
```

4. 运算数缺失:

```
fuyouquan — 80x24
Launching: '/Users/fuyouquan/C/main'
Working directory: '/Users/fuyouquan/C'
1 arguments:
argv[0] = '/Users/fuyouquan/C/main'
1+3
格式错误! Process exited with status 0

Saving session...
...copying shared history...
...saving history...truncating history files...
...completed.

[进程已完成]
```

```
fuyouquan — 80x24
Launching: '/Users/fuyouquan/C/main'
Working directory: '/Users/fuyouquan/C'
1 arguments:
argv[0] = '/Users/fuyouquan/C/main'
1+3
格式错误! Process exited with status 0

Saving session...
...copying shared history...
...saving history...truncating history files...
...completed.

[进程已完成]
```

```
fuyouquan — 80x24
Launching: '/Users/fuyouquan/C/main'
Working directory: '/Users/fuyouquan/C'
1 arguments:
argv[0] = '/Users/fuyouquan/C/main'
#(1*-4)#
(1*-)运算数缺失
Process exited with status 0

Saving session...
...copying shared history...
...saving history...truncating history files...
...completed.

[进程已完成]
```

5.非法算符:

```
fuyouquan — 80x24
Launching: '/Users/fuyouquan/C/main'
Working directory: '/Users/fuyouquan/C'
1 arguments:
argv[0] = '/Users/fuyouquan/C/main'
#(1*-4)#
(1*-)运算数缺失
Process exited with status 0

Saving session...
...copying shared history...
...saving history...truncating history files...
...completed.

[进程已完成]
```

栈的存储结构

优缺点

空间本题主要采用了栈的存储结构，利用了栈“先入后出，后入先出”的存储性质。

由于本题主要采用了两大操作步骤：

1. 中缀表达式转后缀表达式

2. 后缀表达式的求值运算

在中缀表达式转后缀中，需要先将运算符存储，当遇到其他运算符时，根据运算符的优先级以“先入后出”的存取模式进行提取。因此栈的存储结构最符合该步骤要求。

在后缀表达式的求值运算中，需要根据表达式按序进行运算。当遇到运算符时，需要将距离运算符最近的两个数进行运算。栈的“先入后出，后入先出”的性质符合该操作的需求。

实验心得体会

本程序解决了中缀表达式求值的问题。在编写程序中，多次运用了栈的相关操作，提高了对栈的存储模式，栈的性质的理解。熟练了出栈，入栈等相关操作。

在设计该算法时，遇到了操作符与数字字符性质不同，而均需以栈的模式进行存取的问题。为此本程序分别为操作符栈，数字字符栈设置对应栈结构与栈运算函数，分别对数字字符，运算符进行独立操作，互不干扰。

注

中缀表达式进行表达式求值题目编程环境为：VSCode on MacOS，由于系统限制，使用了 `sys/malloc.h` 包，如在 Windows 系统运行请将头文件中：

```
#include<sys/malloc.h>
```

替换为：

```
#include<malloc.h>
```

并在程序中注意其他系统差异。

题目 3

选择的题目

第七题

题目功能描述

根据后序遍历序列和中序遍历序列输出前序遍历序列

问题分析与算法设计思路

该题要求根据后续遍历序列和中序遍历序列输出前序遍历序列，实际就是根据后续遍历序列和中序遍历序列建立二叉树，然后输出该二叉树的前序遍历序列。由后序序列为主体，开始建立二叉树，每一个入树的都是后序序列中的，每一次入树，就去找中序序列中对应的字符下标所在，然后将中序序列分出左右，然后再入树，继续找……，重复上述步骤，直至后序序列中结点全部入树，然后输出该二叉树前序遍历序列。

数据结构定义

```
#define MAX 100
typedef char datatype;
typedef struct bitree{
    datatype data[MAX];
}Bitree;
```

函数功能描述

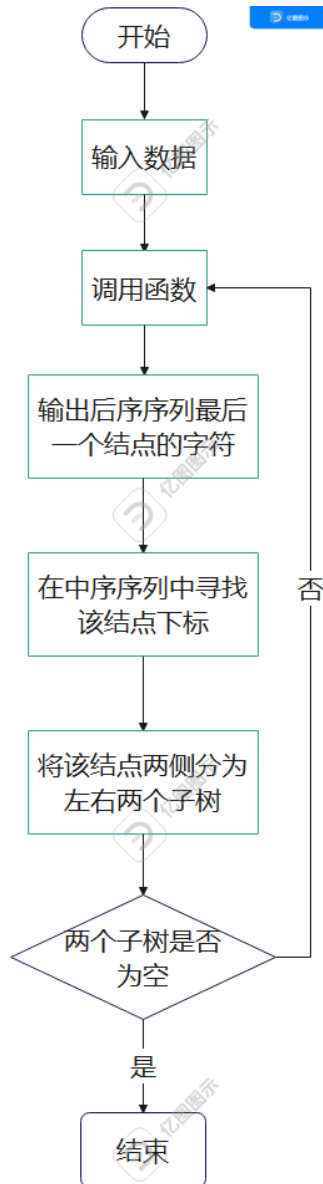
函数

```
void CreatTree(int n,char postorder[],char inorder[])//建立二叉树
{
    if(n<0){//如果输入字符个数小于 0， 则说明已经全部入树
        return;}
    else
    {
        printf("%c",postorder[n]);//输出该后序序列最后一个结点
        int i;
        for(i=0;i<=n;i++)//找到该结点在中序序列中的位置
        {
            if(inorder[i]==postorder[n])
                break;
        }
        CreatTree(i-1,postorder,inorder);//将该结点两侧分为左右子树，利用递归对左右子树重复上述步骤
        CreatTree(n-i-1,postorder+i,inorder+i+1);
        return ;
    }
}
```

功能描述

该函数利用递归的思想，根据后序遍历序列和中序遍历序列建树，输出该二叉树的前序遍历序列。

程序流程图



程序源代码

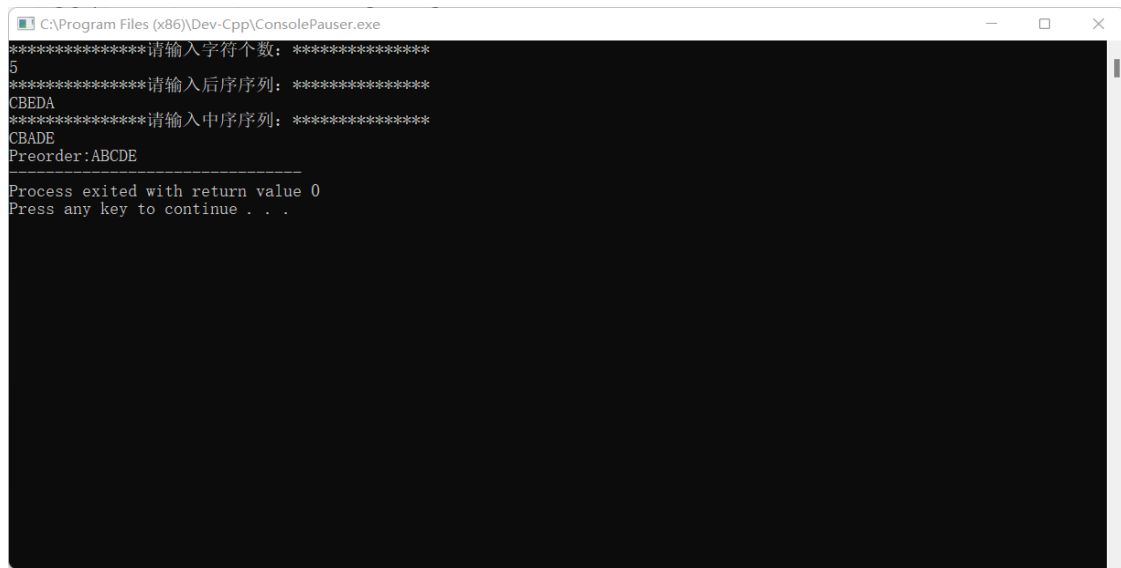
```
#include <stdio.h>
#include <stdlib.h>
#define MAX 100
typedef char datatype;
typedef struct bitree{
    datatype data[MAX];
```

```

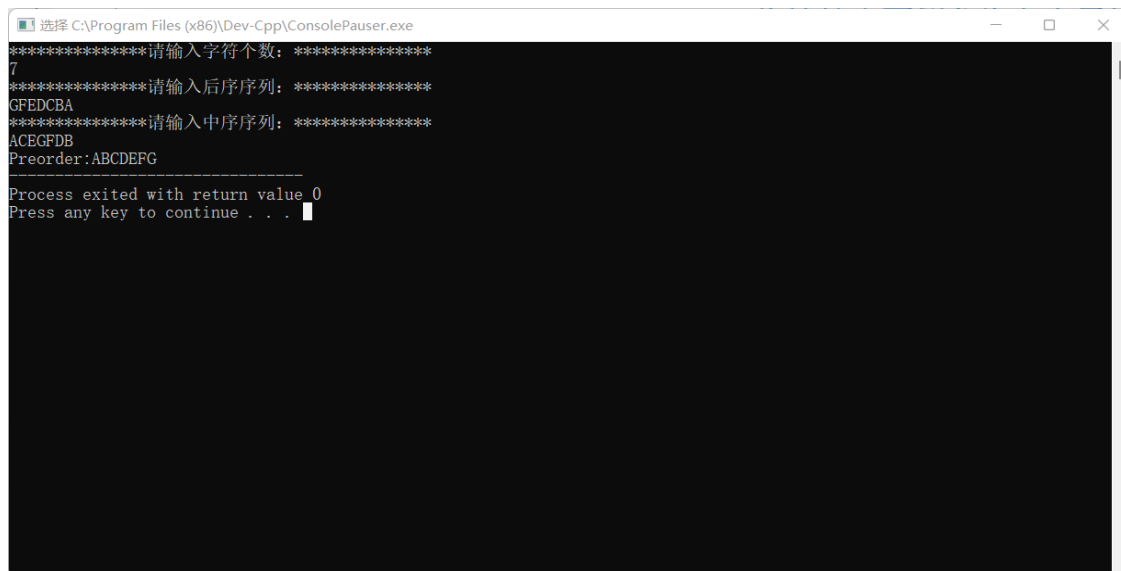
}Bitree;
int n;
void CreatTree(int n,char postorder[],char inorder[])//建立二叉树
{
    if(n<0){//如果输入字符个数小于 0， 则说明已经全部入树
        return;}
    else
    {
        printf("%c",postorder[n]);//输出该后序序列最后一个结点
        int i;
        for(i=0;i<=n;i++){//找到该结点在中序序列中的位置
            {
                if(inorder[i]==postorder[n])
                    break;
            }
            CreatTree(i-1,postorder,inorder);//将该结点两侧分为左右子树，利用递归对左右子
            树重复上述步骤
            CreatTree(n-i-1,postorder+i,inorder+i+1);
            return ;
        }
    }
}
int main()
{
    Bitree postorder;
    Bitree inorder;
    printf("*****请输入字符个数： *****\n");
    scanf("%d",&n);
    printf("*****请输入后序序列： *****\n");
    scanf("%s",postorder.data);
    printf("*****请输入中序序列： *****\n");
    scanf("%s",inorder.data);
    printf("Preorder:");
    CreatTree(n-1,postorder.data,inorder.data);
    return 0;
}

```

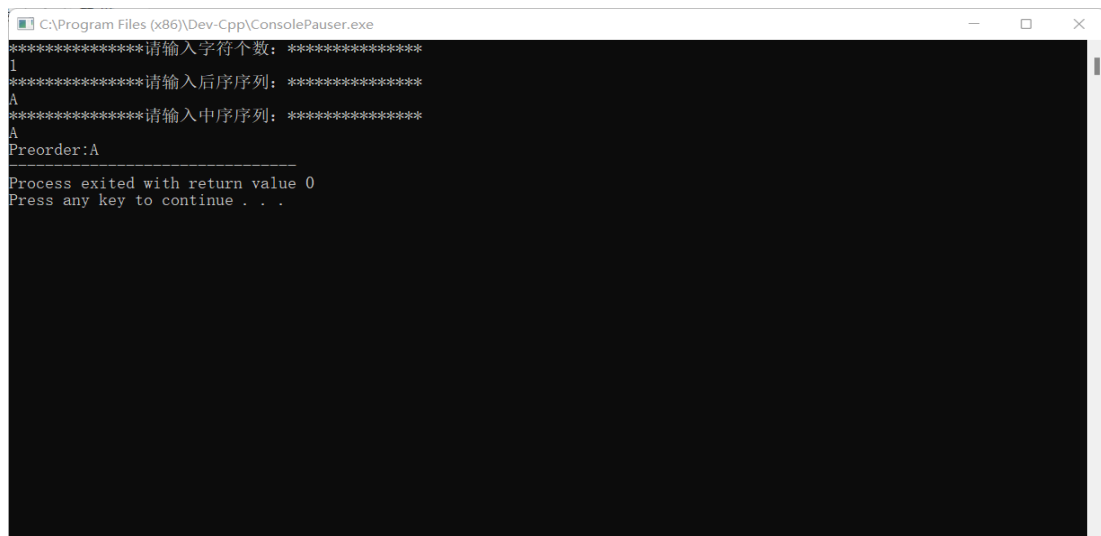
数据输入/输出截图



```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
***** 请输入字符个数: *****
5
***** 请输入后序序列: *****
CBEDA
***** 请输入中序序列: *****
CBADE
Preorder:ABCDE
-----
Process exited with return value 0
Press any key to continue . . .
```



```
选择 C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
***** 请输入字符个数: *****
7
***** 请输入后序序列: *****
GFEDCBA
***** 请输入中序序列: *****
ACEGFDB
Preorder:ABCDEFG
-----
Process exited with return value 0
Press any key to continue . . .
```



```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
***** 请输入字符个数: *****
1
***** 请输入后序序列: *****
A
***** 请输入中序序列: *****
A
Preorder:A
-----
Process exited with return value 0
Press any key to continue . . .
```

```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
*****请输入字符个数:*****
7
*****请输入后序序列:*****
BCAEGFD
*****请输入中序序列:*****
ABCDEFG
Preorder:DACBFEG
-----
Process exited with return value 0
Press any key to continue . . .
```

顺序存储结构

优缺点

空间上，如果数据较多，顺序存储更节省空间，数据较少，则浪费空间。因为链式存储每个节点都有一个指针域。

存储操作上，顺序支持随机存取，操作方便。

插入和删除上，顺序较为麻烦，空间复杂度更大。

采用顺序存储理由

存取速度高效,通过下标来直接存储。

无需为表示结点间的逻辑关系而增加额外的存储空间。

实验心得体会

解决了通过后序序列和中序序列建立二叉树，求出前序遍历序列的问题。我对二叉树的认识更深入了，对递归函数的使用更加熟练了。设计该算法时，最大的困难是如何利用递归的思想解决这个问题，我通过学习其他的一些递归函数，设计出了该题的递归函数，并且不断调试，最终解决了问题。

分工情况

小组成员（姓名 学号）

蔡明硕 21009200731 付友泉 21009200780 李嘉辉 21009201019

分工

蔡明硕：问题 1 题目 19 中等难度-----全部完成
付友泉：问题 2 题目 4 中等难度-----全部完成
李嘉辉：问题 3 题目 7 简单难度-----全部完成

成绩申报

成员	成绩
蔡明硕	85
付友泉	85
李嘉辉	85
总成绩	255