

数据结构实习报告—第四组

小组成员：毕定钧 周杰 法发

目录

1	全国交通咨询模拟	2
1.1	题目描述	2
1.2	头文件介绍	3
1.3	算法流程图	5
1.4	函数介绍	5
1.5	实现演示	10
1.6	测试文件及测试结果	12
2	多关键字排序	13
2.1	题目描述	13
2.2	头文件介绍	14
2.3	算法流程图	16
2.4	函数介绍	16
2.5	测试方案与测试结果	21

前言

本报告为 2023 年春期大学二年级数据结构课程大作业的实习报告，小组成员有毕定钧、周杰、法发。我们选取数据结构题集的实习题 5.3-全国交通咨询模拟、6.7-多关键字排序为本实习内容，基于 Windows 系统下使用 VS 2022 学生版进行开发。由于 L^AT_EX 中 listings 宏包的限制，我不得不删掉代码中含有下划线（即含有 “_”）的注释，具体代码可见我们的 GitHub 仓库：[点击此处跳转](#)。

1 全国交通咨询模拟

1.1 题目描述

5.3 全国交通咨询模拟

【问题描述】

出于不同目的的旅客对交通工具具有不同的要求。例如，因公出差的旅客希望在旅途中的时间尽可能短，出门旅游的游客则期望旅费尽可能省，而老年旅客则要求中转次数最少。编制一个全国城市间的交通咨询程序，为旅客提供两种或三种最优决策的交通咨询。

【基本要求】

- (1) 提供对城市信息进行编辑（如：添加或删除）的功能。
- (2) 城市之间有两种交通工具：火车和飞机。提供对列车时刻表和飞机航班进行编辑（增设或删除）的功能。
- (3) 提供两种最优决策：最快到达或最省钱到达。全程只考虑一种交通工具。
- (4) 旅途中耗费的总时间应该包括中转站的等候时间。
- (5) 咨询以用户和计算机的对话方式进行。由用户输入起始站、终点站、最优决策原则和交通工具，并输出信息：最快需要多长时间才能到达或者最少需要多少旅费才能到达，并详细说明依次于何时乘坐哪一趟列车或哪一次班机到何地。

【测试数据】

参考教科书 7.6 节图 7.33 的全国交通图，自行设计列车时刻表和飞机航班。

【实现提示】

- (1) 对全国城市交通图和列车时刻表及飞机航班表的编辑，应该提供文件形式输入和键盘输入两种方式。飞机航班表的信息应包括：起始站的出发时间、终点站的到达时间和票价；列车时刻表则需根据交通图给出各个路段的详细信息，例如：基于教科书 7.6 节图 7.33 的交通图，对从北京到上海的火车，需给出北京至天津、天津至徐州及徐州至上海各段的出发时间、到达时间及票价等信息。
- (2) 以邻接表作交通图的存储结构，表示边的结点内除含有邻接点的信息外，还应包括交通工具、路程中消耗的时间和花费以及出发和到达的时间等多项属性。

【选作内容】

增加旅途中转次数最少的最优决策。

1.2 头文件介绍

```

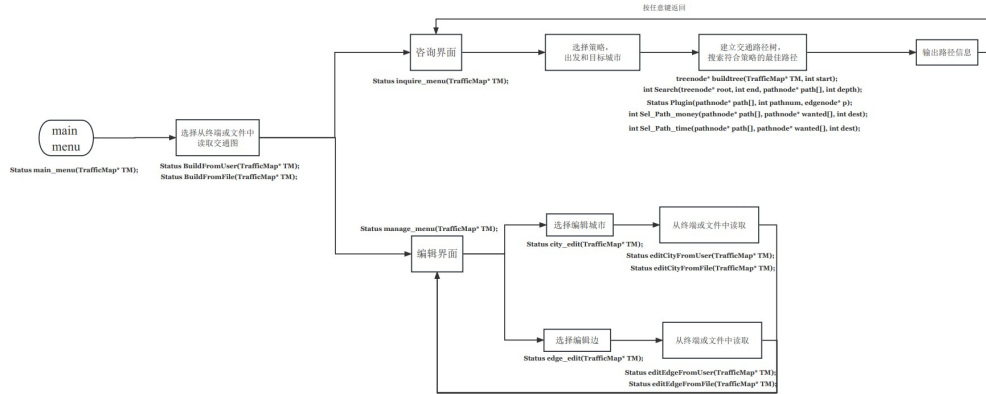
1 #pragma once
2 #define MAX_CITY_NUM 20//支持的城市数目
3 #define MAX_NAME_LEN 12//城市名称最大长度
4 #define MAX_PATH_LEN 64//读取文件路径的最大长度
5 #define MAX_STRING_LEN 64//读取输入的最大长度
6 #define MAX_PATH_NUM 100//存储路径的最大数
7 #define ERROR 0
8 #define OK 1
9 #define ADMIN_KEY "123456"
10 #define INF 999
11 #define scanf scanf_s
12 #define strcpy(s_to, s) strcpy_s(s_to, strlen(s) + 1, s)
13 #define fscanf fscanf_s
14
15 typedef enum { false = 0, true } bool;
16 typedef bool Status;
17 typedef int VertexType;
18
19 //从出发城市到到达城市的列车的相关信息
20 typedef struct edgeinfo {
21     char number[10]; //航班号或车次号, 可根据第二位是否是字母判断是飞机还是火车
22     int departure_time; //出发时间 (用一天中的分钟表示, e.g. 7: 30 表示为 450)
23     int arrival_time; //到达时间 (同上)
24     int cost; //费用
25 } edgeinfo;
26
27 //-----交通图的邻接表存储表示-----
28 // (请参考《数据结构 (C 语言版)》P163, 严蔚敏, 吴伟民, 清华大学出版社)
29 typedef struct edgenode {
30     int destination; //当前路径指向的城市的编号
31     struct edgenode* nextedge; //指向下一条路径的指针
32     edgeinfo info; //当前路径的相关信息
33 } edgenode;
34 typedef struct CityNode {
35     VertexType citynum; //城市编号
36     edgenode* firstedge; //指向从该城市出发的第一条路径
37 } CityNode, CityList[MAX_CITY_NUM];
38 typedef struct {
39     CityList cities; //以邻接表形式存储的交通图
40     int citynum, edgenum; //交通图当前的城市数与路径数
41 } TrafficMap;
42
43 char* CITY[MAX_NAME_LEN]; //全局变量, 用于存放城市名称与编号
44
45 typedef struct treenode {
46     edgenode* edge; //边
47     struct treenode* firstchild; //第一个孩子结点
48     struct treenode* nextsib; //下一个兄弟

```

```
49 } treenode;
50
51 typedef struct pathnode {
52     edgenode edge; //边
53     struct pathnode* next; //下一个路径结点
54 } pathnode;
55
56
57 int findflag = 0; //标记是否搜索到终点城市
58 int pathcount = 0; //记录搜索到的路径个数
59
60 //初始化交通图 TM
61 extern Status InitTrafficMap(TrafficMap* TM);
62 //读取并建立交通图 TM
63 extern Status InsertTrafficMap(TrafficMap* TM);
64 //从终端输入交通图 TM
65 extern Status BuildFromUser(TrafficMap* TM);
66 //从文件输入交通图 TM
67 extern Status BuildFromFile(TrafficMap* TM);
68 //系统主菜单
69 extern Status main_menu(TrafficMap* TM);
70 //需求选择菜单
71 extern Status inquire_menu(TrafficMap* TM);
72 //管理面板
73 extern Status manage_menu(TrafficMap* TM);
74 //编辑城市
75 extern Status city_edit(TrafficMap* TM);
76 //从输入中读取编辑城市的指令
77 extern Status editCityFromUser(TrafficMap* TM);
78 //从文件中读取编辑城市的指令
79 extern Status editCityFromFile(TrafficMap* TM);
80 //编辑城市之间的路径
81 extern Status edge_edit(TrafficMap* TM);
82 //从输入中读取编辑路径的指令
83 extern Status editEdgeFromUser(TrafficMap* TM);
84 //从文件中读取编辑路径的指令
85 extern Status editEdgeFromFile(TrafficMap* TM);
86 //查找同一城市出发的上一条路径
87 extern edgenode* findpre(TrafficMap* TM, int i, edgenode* p);
88 //建立交通路径树
89 treenode* buildtree(TrafficMap* TM, int start);
90 //搜索 end 并将可行的路径存于 path[] 中
91 int Search(treenode* root, int end, pathnode* path[], int depth);
92 //将 p 插入到 path[pathnum] 中
93 Status Plugin(pathnode* path[], int pathnum, edgenode* p);
94 //挑选到达 dest 的最省钱的路径存于 wanted[] 中
95 int Sel_Path_money(pathnode* path[], pathnode* wanted[], int dest);
96 //挑选到达 dest 的最省时的路径存于 wanted[] 中
```

```
97 int Sel_Path_time(pathnode* path[], pathnode* wanted[], int dest);
```

1.3 算法流程图



1.4 函数介绍

1. 建立交通路径树。

通过循环进行路径的构建，函数返回指向第一条从起点出发的边。所建成的树每一层为上一层之后可乘坐的列车/航班，层与层之间为父子关系，层内为兄弟关系，均使用指针相连。

```
1  treenode* buildtree(TrafficMap* TM, int start, edgenode* pre) {
2      edgenode* q = TM->cities[start].firstedge;
3      treenode* root = (treenode*)malloc(sizeof(treenode));
4      treenode* p;
5      treenode* k = root;
6      int first = 1;
7      if (q == NULL)
8          return NULL;
9      else
10     {
11         if (pre == NULL) {
12             root->edge = q;
13             root->firstchild = buildtree(TM, q->destination, q);
14             root->nextsib = NULL;
15             for (q = TM->cities[start].firstedge->nextedge; q != NULL; q = q->nextedge) {
16                 p = (treenode*)malloc(sizeof(treenode));
17                 k->nextsib = p;
18                 k = p;
19                 p->edge = q;
20                 p->firstchild = buildtree(TM, q->destination, q);
21                 p->nextsib = NULL;
22             }
23         }
24         else
25         {
```

```

26     for (q = TM->cities[start].firstedge; q != NULL; q = q->nextedge) {
27         if (q->info.departure_time > pre->info.arrival_time && !(isdigit(q->info.number[1])
~ isdigit(pre->info.number[1])))
28         {
29             if (first == 1) {
30                 root->edge = q;
31                 root->firstchild = buildtree(TM, q->destination, q);
32                 root->nextsib = NULL;
33                 first = 0;
34             }
35             else {
36                 p = (treenode*)malloc(sizeof(treenode));
37                 k->nextsib = p;
38                 k = p;
39                 p->edge = q;
40                 p->firstchild = buildtree(TM, q->destination, q);
41                 p->nextsib = NULL;
42             }
43         }
44     }
45 }
46 }
47
48 return root;
49 }//buildtree

```

2. 查找所有从起点城市到终点城市的路径

Search 函数使用递归思想进行搜索，在回溯时借助全局变量 pathcount 和 plugin 函数进行路径的记录，并通过函数形参 depth 进行深度记录，以标志路径记录的结束，从而进行 pathcount 的增 1。

```

1 int Search(treenode* root, int end, pathnode* path[], int depth) {
2     treenode* p = NULL;
3     for (p = root; p != NULL; p = p->nextsib) {
4         //找到终点城市
5         if (p->edge->destination == end) {
6             findflag = 1;
7             Plugin(path, pathcount, p->edge); //插入 path
8         }
9         //未找到，递归搜索孩子
10        else {
11            findflag = 0; //递归前重置标记
12            Search(p->firstchild, end, path, depth + 1); //深度加 1
13            //孩子中含终点城市
14            if (findflag == 1) {
15                Plugin(path, pathcount, p->edge); //插入 path
16                if (depth == 0) //追溯到深度为 0，则写入路径已完成，准备写下一条路径，并重置标记
17                {
18                    pathcount++;

```

```

19     findflag = 0;
20     }
21     }
22     }
23     }
24     return OK;
25 }//Search
26
27 Status Plugin(pathnode* path[], int pathnum, edgenode* p) {
28     pathnode* tmp = (pathnode*)malloc(sizeof(pathnode));
29     pathnode* ptr;
30     //填入 path[pathnum] 的首位 (考虑 path[num] 为空)
31     if (path[pathnum] == NULL)
32     {
33         tmp->edge = *p;
34         tmp->next = NULL;
35         path[pathnum] = tmp;
36     }
37     else {
38         ptr = path[pathnum];
39         tmp->edge = *p;
40         tmp->next = ptr;
41         path[pathnum] = tmp;
42     }
43     return OK;
44 }//Plugin

```

3. 挑选最省钱路径

对于形成好的 path[] 路径, 进行选择, 将最省钱的存入 wanted[] 中, wanted 每一个分量指向一个表示路径的链表。考虑到可能不止一条最省钱的路径, 故使用数组进行存放, 以便呈现完备的信息。参数 dest 表示到达城市的数组下标。

```

1 int Sel_Path_money(pathnode* path[], pathnode* wanted[], int dest) {
2     int number = 0; //wanted 数组下标
3     int money = 0; //记录该路径的费用
4     int min_money = 10000;
5     int last_money = 10000;
6     pathnode* ptr_w; //指向 wanted 中路径结点的指针
7     pathnode* ptr_p; //指向 path 中路径结点的指针
8     pathnode* tmp;
9     for (int i = 0; i < MAX_PATH_NUM && path[i] != NULL; i++) {
10         money = 0;
11         if (path[i]->edge.destination != dest) //非一步到达的路径
12         {
13             tmp = (pathnode*)malloc(sizeof(pathnode));
14             tmp->edge = path[i]->edge;
15             tmp->next = NULL;
16             wanted[number] = tmp;
17             ptr_w = tmp;

```

```
18     money += tmp->edge.info.cost;//更新 money
19
20     //写入所有的到达终点前一步的结点
21     for (ptr_p = path[i]->next; ptr_p != NULL && ptr_p->edge.destination != dest; ptr_p =
ptr_p->next) {
22         tmp = (pathnode*)malloc(sizeof(pathnode));
23         tmp->edge = ptr_p->edge;
24         tmp->next = NULL;
25         ptr_w->next = tmp;
26         ptr_w = tmp;
27         money += tmp->edge.info.cost;
28     }
29     tmp = (pathnode*)malloc(sizeof(pathnode));
30     last_money = 10000;
31     //处理到达目的地的最后一步
32     for (; ptr_p != NULL; ptr_p = ptr_p->next) {
33         if (last_money > ptr_p->edge.info.cost)
34         {
35             last_money = ptr_p->edge.info.cost;
36             tmp = (pathnode*)malloc(sizeof(pathnode));
37             tmp->edge = ptr_p->edge;
38             tmp->next = NULL;
39             ptr_w->next = tmp;
40         }
41     }
42     money += last_money;
43 }
44 //一步到达的路径
45 else {
46     last_money = 10000;
47     for (ptr_p = path[i]; ptr_p != NULL; ptr_p = ptr_p->next) {
48         //找到最小直接更新
49         if (last_money > ptr_p->edge.info.cost)
50         {
51             last_money = ptr_p->edge.info.cost;
52             tmp = (pathnode*)malloc(sizeof(pathnode));
53             tmp->edge = ptr_p->edge;
54             tmp->next = NULL;
55             wanted[number] = tmp;
56         }
57     }
58     money = last_money;
59 }
60 //若费用小于最小费用则写入 wanted[0], 其余删除
61 if (money < min_money)
62 {
63     wanted[0] = wanted[number];
64     for (int j = 1; j <= number; j++)
65         wanted[j] = NULL;
```



```
66     min_money = money;
67     number = 1;
68 }
69 //若费用为目前最低费用, 则保留, number 计数加一
70 else if (money == min_money) {
71     number++;
72 }
73 //若费用大于目前最低费用, 则清除刚写入的 wanted[number] 路径
74 else
75     wanted[number] = NULL;
76 }
77 return min_money;
78 }
```

4. 选择最省时路径与 3 类似, 只是改变了选择原则。

```
1 int Sel_Path_time(pathnode* path[], pathnode* wanted[], int dest) {
2     int number = 0;
3     int min_duration = 10000;
4     int start_time, end_time;
5     pathnode* ptr_w;
6     pathnode* ptr_p;
7     pathnode* tmp;
8     for (int i = 0; i < MAX_PATH_NUM && path[i] != NULL; i++) {
9         if (path[i]->edge.destination != dest)
10            {
11                tmp = (pathnode*)malloc(sizeof(pathnode));
12                tmp->edge = path[i]->edge;
13                tmp->next = NULL;
14                wanted[number] = tmp;
15                start_time = tmp->edge.info.departure_time;
16                ptr_w = tmp;
17                for (ptr_p = path[i]->next; ptr_p != NULL && ptr_p->edge.destination != dest; ptr_p = ptr_p->next) {
18                    tmp = (pathnode*)malloc(sizeof(pathnode));
19                    tmp->edge = ptr_p->edge;
20                    tmp->next = NULL;
21                    ptr_w->next = tmp;
22                    ptr_w = tmp;
23                }
24                tmp = (pathnode*)malloc(sizeof(pathnode));
25                end_time = 10000;
26                for (; ptr_p != NULL; ptr_p = ptr_p->next) {
27                    if (end_time > ptr_p->edge.info.arrival_time)
28                    {
29                        end_time = ptr_p->edge.info.arrival_time;
30                        tmp = (pathnode*)malloc(sizeof(pathnode));
31                        tmp->edge = ptr_p->edge;
32                        tmp->next = NULL;
```

```
33     ptr_w->next = tmp;
34 }
35 }
36 }
37 else
38 {
39     end_time = 10000;
40     start_time = 0;
41     for (ptr_p = path[i]; ptr_p != NULL; ptr_p = ptr_p->next) {
42         if ((ptr_p->edge.info.arrival_time - ptr_p->edge.info.departure_time) < (end_time -
start_time))
43         {
44             end_time = ptr_p->edge.info.arrival_time;
45             start_time = ptr_p->edge.info.departure_time;
46             tmp = (pathnode*)malloc(sizeof(pathnode));
47             tmp->edge = ptr_p->edge;
48             tmp->next = NULL;
49             wanted[number] = tmp;
50
51         }
52     }
53 }
54 if ((end_time - start_time) < min_duration)
55 {
56     wanted[0] = wanted[number];
57     for (int j = 1; j <= number; j++)
58         wanted[j] = NULL;
59     min_duration = end_time - start_time;
60     number = 1;
61 }
62 else if ((end_time - start_time) == min_duration) {
63     number++;
64 }
65 else
66     wanted[number] = NULL;
67 }
68 return min_duration;
69 }
```

1.5 实现演示

```
-----全国交通咨询系统-----
初始化完成。准备读取交通图。
请选择输入方式：
(0 = "从终端输入"; 1 = "从文件输入")
1
请输入文件地址：
test.txt_
```

```
-----全国交通咨询系统-----  
-----主菜单-----  
=====
```

请选择操作：
1: 用户界面
2: 管理面板
0: 退出

```
=====
```

请选择: _

```
-----全国交通咨询系统-----  
-----用户界面-----  
=====
```

欢迎您使用全国交通咨询系统!
请选择期望的出行策略:
1: 最快到达
2: 最省钱到达
0: 返回主菜单

```
=====
```

请选择:

```
-----全国交通咨询系统-----  
-----管理菜单-----  
=====
```

1: 城市编辑
2: 列车/航班信息编辑
0: 返回主菜单

```
=====
```

请选择: _

```
-----全国交通咨询系统-----  
-----管理菜单-----  
=====
```

1: 城市编辑
2: 列车/航班信息编辑
0: 返回主菜单

```
=====
```

请选择: 1
请输入管理员密码:

```
-----全国交通咨询系统-----  
-----管理菜单-----  
=====
```

城市编辑
选择编辑模式:
1: 直接输入
2: 文件输入
0: 退出

```
=====
```

请选择:

```

-----全国交通咨询系统-----
-----管理菜单-----
城市编辑
请选择需要执行的操作：
1： 添加
2： 删除
0： 退出编辑
=====
请选择：

```

1.6 测试文件及测试结果

测试所用的文件如下：

```

BeiJing TianJin K7727 06:00 08:20 20
BeiJing TianJin C2551 06:00 06:33 55
BeiJing TianJin C2001 06:15 06:55 55
BeiJing TianJin G171 06:00 06:34 56
BeiJing TianJin G887 07:40 08:20 61
BeiJing ShenYang D825 07:08 13:59 195
BeiJing ShenYang G951 06:50 10:17 329
TianJin ShenYang K1985 06:40 15:30 98
TianJin ShenYang G1298 06:40 10:40 219
TianJin ShenYang G1299 08:30 12:30 219
TianJin ShenYang CA2987 08:15 09:55 240
BeiJing ShenYang CA1651 09:00 10:30 1080
BeiJing ShenYang CZ6102 09:00 10:35 740

```

1. 从文件中读入交通图

```

-----全国交通咨询系统-----
初始化完成。准备读取交通图。
请选择输入方式：
(0 = "从终端输入"; 1 = "从文件输入")
1
请输入文件地址：
test.t

```

2. 咨询北京到沈阳最省时路径

```

-----全国交通咨询系统-----
-----用户界面-----
请输入起点城市：
BeiJing
请输入终点城市：
ShenYang
以下是为您找到的最省时旅程安排：
[方案1]:

```

起始站	终点站	航班/列车号	出发时间	到达时间	费用
BeiJing	ShenYang	CA1651	09:00	10:30	1080

```

-----
耗时:1h30min
按回车键返回咨询菜单

```

3. 咨询北京到沈阳最省钱路径

```

-----全国交通咨询系统-----
-----用户界面-----
请输入起点城市：
BeiJing
请输入终点城市：
ShenYang
以下是为您找到的最省钱旅程安排：
[方案1]:
起始站      终点站      航班/列车号  出发时间  到达时间  费用
BeiJing      TianJin      C2551       06:00    06:33     55
TianJin      ShenYang     K1985       06:40    15:30     98
-----
合计：                                           153
按回车键返回咨询菜单

```

4. 删除天津结点

```

-----全国交通咨询系统-----
-----管理菜单-----
城市编辑
请选择需要执行的操作：
1: 添加
2: 删除
0: 退出编辑
=====
请选择：2
请输入需要删除的城市名：TianJin
删除成功！
按回车键返回上一级

```

5. 再次咨询北京到沈阳最省钱路径

```

-----全国交通咨询系统-----
-----用户界面-----
请输入起点城市：
BeiJing
请输入终点城市：
ShenYang
以下是为您找到的最省钱旅程安排：
[方案1]:
起始站      终点站      航班/列车号  出发时间  到达时间  费用
BeiJing      ShenYang     D825        07:08    13:59     195
-----
合计：                                           195
按回车键返回咨询菜单

```

2 多关键字排序

2.1 题目描述

6.7 多关键字排序

【问题描述】

多关键字的排序有其一定的实用范围。例如：在进行高考分数处理时，除了需对总分进行排序

外，不同的专业对单科分数的要求不同，因此尚需在总分相同的情况下，按用户提出的单科分数的次序要求排出考生录取的次序。

【基本要求】

- (1) 假设待排序的记录数不超过 10000，表中记录的关键字数不超过 5，各个关键字的范围均为 0 至 100。按用户给定的进行排序的关键字的优先关系，输出排序结果。
- (2) 约定按 LSD 法进行多关键字的排序。在对各个关键字进行排序时采用两种策略：其一是利用稳定的内部排序法，其二是利用“分配”和“收集”的方法。并综合比较这两种策略。

【测试数据】

由随机数产生器生成。

【实现提示】

用 5 至 8 组数据比较不同排序策略所需时间。由于是按 LSD 方法进行排序，则对每个关键字均可进行整个序列的排序，但在利用通常的内部排序方法进行排序时，必须选用稳定的排序方法。借助“分配”和“收集”策略进行的排序，如同一趟“基数排序”，由于关键字的取值范围为 0 至 100，则分配时将得到 101 个链表。

【选作内容】

增添按 MSD 策略进行排序，并和上述两种排序策略进行综合比较

2.2 头文件介绍

宏定义如下：

```
1 #pragma once
2 #define MAXSIZE 10000
3 #define MAX_KEY_NUM 6
4 #define MAX_KEY_SIZE 100
5 #define MAX_PATH_LEN 64
6 #define MAX_STRING_LEN 64
7 #define OK 1
8 #define ERROR 0
9 #define OVERFLOW -2
10 #define scanf scanf_s
```

MAXSIZE 代表了最大支持的关键字数目，*MAX_KEY_NUM* 则是最大支持的关键字数目，*MAX_KEY_SIZE* 定义了最大支持的关键字名称的长度，*MAX_PATH_LEN* 为输入文件地址时支持的最大文件地址长度，*MAX_STRING_LEN* 是从文件输入时一次读取的最大长度，*OK* 与 *ERROR* 是函数的返回值，正常退出则返回 *OK*，出现异常则返回 *ERROR*。分配空间时可能出现失败，因此定义了 *OVERFLOW* 以在分配失败时退出。由于 VS 2022 认为 *scanf* 函数不安全，我们将 *scanf* 改为 *scanf_s* 以方便从各个系统切换。

数据结构如下：

```

1 typedef struct {
2     int length;//当前长度
3     int listsize;//当前分配的存储容量
4     int keynum;//关键字的数目
5 } InfoType;
6 typedef struct {
7     KeyType key[MAXSIZE];//关键词矩阵
8     char name[MAX_NAME_LEN];//当前关键词名称
9 } keyword;
10 typedef struct {
11     ElemType* elem;//存储空间基址
12     keyword keys[MAX_KEY_NUM];//关键词
13     InfoType info;
14 } SqList;//采用顺序表存储结构
15 int* num, *order;//用于存放排序后的序号
16 //用于排序所用时长
17 typedef struct RunTime
18 {
19     clock_t start;
20     clock_t end;
21     double duration;
22 }RunTime;
23 RunTime RS_time, LSDM_time;

```

本算法采用了顺序表的存储结构，在顺序表 SqList 中，包括其存储空间基址用于存放学号（测试数据的编号）、一个 keyword 数组用于存放每一个关键词、InfoType 则用于存放顺序表的相关信息。每一个 keyword 都包含一个大小为 MAXSIZE 的数组用于存放每个数据的关键字和一个字符串来存放该关键字的名称。而 InfoType 中包括当前已读入数据的长度、当前分配的存储容量、当前关键字的数目；其中当前分配的存储容量本来被用于在分配内存不足时重新分配内存，但是没有实现成功。后面的 RunTime 结构则用来计时。为了方便传参以及简化函数，我们定义了全局变量 num 数组来存放排序后的序号，由于分工的两个人没有达成共识，order 内存放的是第几个数据，而 num 存放的是数据的编号（如学号 10692 等），考虑到学号数据并不一定是递增的，我们保留了两个全局变量没有进行优化。

以下是我们使用的函数的声明，将在后面详细介绍：

```

1 //构造一个空的线性表 L
2 extern Status InitList(SqList* L);
3 //建立顺序表的存储结构
4 extern Status BuildList(SqList* L);
5 //基数为 10 的 LSD 基数排序，采取“分配”和“收集”的策略
6 extern Status Radix_Sort_10(SqList L, int inquire);
7 //输出到终端中
8 extern Status PrintToUser(SqList L);
9 //输出到文件中
10 extern Status PrintToFile_RS(SqList L);
11 //输出到文件中
12 extern Status PrintToFile_RS(SqList L);
13 //LSDmerge 排序的启动程序

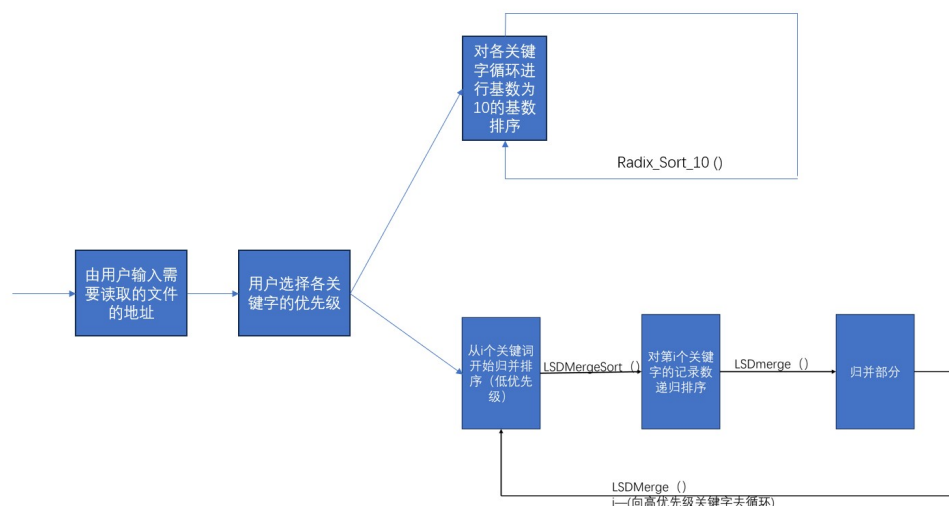
```

```

14 extern int LSDMerge(SqList* L);
15 //LSDmerge 排序的递归调用部分
16 extern int LSDMergeSort(SqList* L, int left, int right, int keynum);
17 //LSDmerge 合并部分
18 extern int LSDmerge(SqList* L, int left, int mid, int right, int keynum);

```

2.3 算法流程图



2.4 函数介绍

使用 InitList 函数为我们的顺序表分配内存空间：

```

1 Status InitList(SqList* L) {
2     L->elem = (ElemType*)malloc(LIST_INIT_SIZE * sizeof(ElemType)); //存储分配失败
3     if (!L->elem) exit(OVERFLOW);
4     L->info.length = 0;
5     L->info.listsize = LIST_INIT_SIZE;
6     L->info.keynum = 0;
7     return OK;
8 } //InitList

```

BuildList 函数用于从输入的文件中读取内容并且建立存储结构，以方便排序操作：

```

1 Status BuildList(SqList* L) {
2     char file_pos[MAX_PATH_LEN];
3     printf("请输入文件地址: \n");
4     scanf("%s", file_pos, MAX_PATH_LEN);
5     errno_t err; FILE* fp;
6     err = fopen_s(&fp, file_pos, "r"); //FILE fp = fopen(file_pos, "r");
7     while (fp == NULL) {
8         printf("文件读取失败! 请重新输入文件地址: \n");
9         scanf("%s", file_pos, MAX_PATH_LEN);
10        err = fopen_s(&fp, file_pos, "r");

```



```

11     }
12     char input[MAX_STRING_LEN];
13     fgets(input, MAX_STRING_LEN, fp);
14     int i = 0, j;
15     for (j = 0; input[j] != '\t'; j++);
16     while (input[j++] != '\n') {
17         for (; input[j] != '\t' && input[j] != '\n'; j++) {
18             L->keys[L->info.keynum].name[i++] = input[j];
19         }
20         L->keys[L->info.keynum].name[i] = '\0';
21         i = 0; L->info.keynum++;
22     }
23     do {
24         fscanf_s(fp, "%d", &L->elem[L->info.length]);
25         for (int k = 0; k < L->info.keynum; k++) {
26             fscanf_s(fp, "%d", &L->keys[k].key[L->info.length]);
27         }
28         L->info.length++;
29     } while (fgetc(fp) == '\n');
30     return OK;
31 } // BuildList

```

LSD 的基数排序的核心算法如下：

```

1 Status LSD_Radix_Sort_10(SqList L, int inquire) {
2     int Base[10][MAXSIZE];
3     int* sort, * sort_next, * tmp;
4     int* num_next;
5     sort_next = (int*)malloc((L.info.length + 1) * sizeof(int));
6     for (int i = 0; i < L.info.length; i++)
7         sort_next[i] = L.keys[inquire].key[num[i]];
8     tmp = (int*)malloc((L.info.length + 1) * sizeof(int));
9     for (int j = 1; j < 5; j++) {
10        sort = (int*)malloc((L.info.length + 1) * sizeof(int));
11        num_next = (int*)malloc((L.info.length + 1) * sizeof(int));
12        for (int i = 0; i < 10; i++) Base[i][0] = 0;
13        for (int i = 0; i < L.info.length; i++) {
14            tmp[i] = (sort_next[i] / (int)pow(10, j - 1)) % 10;
15            if (tmp[i] < 0 || tmp[i] > 9) tmp[i] = 0;
16            Base[tmp[i]][0]++;
17            Base[tmp[i]][Base[tmp[i]][0]] = sort_next[i];
18        }
19        for (int i = 1; i < 10; i++) Base[i][0] += Base[i - 1][0];
20        for (int i = L.info.length - 1; i >= 0; i--) {
21            Base[tmp[i]][0]--;
22            sort[Base[tmp[i]][0]] = sort_next[i];
23            num_next[Base[tmp[i]][0]] = num[i];
24        }
25        sort_next = sort;

```

```

26     num = num_next;
27 }
28 return OK;
29 }

```

基数方法通过依次读取 sort 中关键字的末位，将它按 0 9 归类到 Base 中，Base 一共有 10 类，分别为 0 9。之后再重新排序到 sort_next 数组中。然后释放 sort_next 并将其值存到 sort 中，对关键字新的末位进行上述操作，直到 sort 中的关键字均为 0。由于本次读取的成绩均为 4 位数以下，我们进行四次这样的循环即可。

MSD 的基数排序的核心算法如下：

```

1 Status MSD_Radix_Sort_10(SqList L, int inquire) {
2     int Base[10][MAXSIZE];
3     int* sort, * sort_next, *tmp;
4     int* num_next;
5     sort_next = (int*)malloc((L.info.length + 1) * sizeof(int));
6     for (int i = 0; i < L.info.length; i++)
7         sort_next[i] = L.keys[inquire].key[num2[i]];
8     tmp = (int*)malloc((L.info.length + 1) * sizeof(int));
9     for (int j = 4; j > 0; j--) {
10        sort = (int*)malloc((L.info.length + 1) * sizeof(int));
11        num_next = (int*)malloc((L.info.length + 1) * sizeof(int));
12        for (int i = 0; i < 10; i++) Base[i][0] = 0;
13        for (int i = 0; i < L.info.length; i++) {
14            tmp[i] = (sort_next[i] / (int)pow(10, j - 1)) % 10;
15            if (tmp[i] < 0 || tmp[i] > 9) tmp[i] = 0;
16            Base[9-tmp[i]][0]++;
17            Base[9-tmp[i]][Base[9-tmp[i]][0]] = sort_next[i];
18        }
19        for (int i = 1; i < 10; i++) Base[i][0] += Base[i - 1][0];
20        for (int i = L.info.length - 1; i >= 0; i--) {
21            Base[9-tmp[i]][0]--;
22            sort[Base[9-tmp[i]][0]] = sort_next[i];
23            num_next[Base[9-tmp[i]][0]] = num2[i];
24        }
25        sort_next = sort;
26        num2 = num_next;
27    }
28    return OK;
29 }

```

PrintToUser 用于向终端中输出排序结果，但是一般不使用这种方法：

```

1 Status PrintToUser(SqList L) {
2     printf("id\t");
3     for (int i = 0; i < L.info.keynum - 1; i++) printf("%s\t", L.keys[i].name);
4     printf("%s\n", L.keys[L.info.keynum - 1].name);
5     int k = 0;
6     for (int i = 0; i < L.info.length; i++) {

```

```

7     printf("%d\t", L.elem[num[i]]);
8     for (int j = 0; j < L.info.keynum - 1; j++)
9         printf("%d\t", L.keys[j].key[num[i]]);
10    printf("%d\n", L.keys[L.info.keynum - 1].key[num[i]]);
11    }
12    return OK;
13 } // PrintToUser

```

PrintToFile 用于将排序结果输出到文件中，可由外界指定输出的文件的名称及路径：

```

1 Status PrintToFile(SqList L, char* filepos) {
2     errno_t err; FILE* fp;
3     err = fopen_s(&fp, filepos, "w");
4     fprintf(fp, "id\t");
5     for (int i = 0; i < L.info.keynum - 1; i++) fprintf(fp, "%s\t", L.keys[i].name);
6     fprintf(fp, "%s\n", L.keys[L.info.keynum - 1].name);
7     int k = 0;
8     for (int i = 0; i < L.info.length; i++) {
9         fprintf(fp, "%d\t", L.elem[num[i]]); // num[i]/order[i] - order[0]
10        for (int j = 0; j < L.info.keynum - 1; j++)
11            fprintf(fp, "%d\t", L.keys[j].key[num[i]]);
12        fprintf(fp, "%d\n", L.keys[L.info.keynum - 1].key[num[i]]);
13    }
14    return OK;
15 } // PrintToFile

```

由于另一位负责写归并排序的同学代码比较特殊，只能再定义一个函数专门用于他的算法的输出，将归并排序的结果输出到文件 *LSDM_sort.txt*：

```

1 Status PrintToFile_LSDM(SqList L, char* filepos) {
2     errno_t err; FILE* fp;
3     err = fopen_s(&fp, filepos, "w");
4     fprintf(fp, "id\t");
5     for (int i = 0; i < L.info.keynum - 1; i++) fprintf(fp, "%s\t", L.keys[i].name);
6     fprintf(fp, "%s\n", L.keys[L.info.keynum - 1].name);
7     int k = 0;
8     for (int i = 0; i < L.info.length; i++) {
9         fprintf(fp, "%d\t", L.elem[order[i] - 10001]);
10        for (int j = 0; j < L.info.keynum - 1; j++)
11            fprintf(fp, "%d\t", L.keys[j].key[order[i] - 10001]);
12        fprintf(fp, "%d\n", L.keys[L.info.keynum - 1].key[order[i] - 10001]);
13    }
14    return OK;
15 }

```

将选中的关键词那一列的记录数需要排序的片段进行二路归并排序。

```

1 int LSDmerge(SqList* L, int left, int mid, int right, int number) {
2     int* elem1, * elem2;

```

```

3     elem1 = (int*)malloc((mid - left + 1) * sizeof(int));
4     elem2 = (int*)malloc((right - mid) * sizeof(int));
5     int i = 0, j = 0, k = 0;
6     for (i = 0; i <= mid - left; i++) {
7         elem1[i] = order[i + left];
8     }
9     for (j = 0; j <= right - mid - 1; j++) {
10        elem2[j] = order[j + mid + 1];
11    }
12    for (i = j = 0, k = left; i <= mid - left && j <= right - mid - 1; k++) {
13        int a, b;
14        a = L->keys[number].key[elem1[i] - 10001];
15        b = L->keys[number].key[elem2[j] - 10001];
16        if (a <= b) {
17            order[k] = elem1[i++];
18        }
19        else order[k] = elem2[j++];
20    }
21    if (i > mid - left) {
22        for (j; j <= right - mid - 1; j++) {
23            order[k] = elem2[j];
24            k++;
25        }
26    }
27    else if (j > right - mid - 1) {
28        for (i; i <= mid - left; i++) {
29            order[k] = elem1[i];
30            k++;
31        }
32    }
33    return OK;
34 }

```

归并排序递归部分，从小的关键字记录数片段进行排序，最后递归直至将所选关键字部分的记录数完成排序。

```

1 int LSDMergeSort(SqlList* L, int left, int right, int keynum) {
2     int mid;
3     if (left < right) {
4         mid = (left + right) / 2;
5         if (!LSDMergeSort(L, left, mid, keynum))
6             return ERROR;
7         if (!LSDMergeSort(L, mid + 1, right, keynum))
8             return ERROR;
9         if (!LSDmerge(L, left, mid, right, keynum))
10            return ERROR;
11    }
12    return OK;

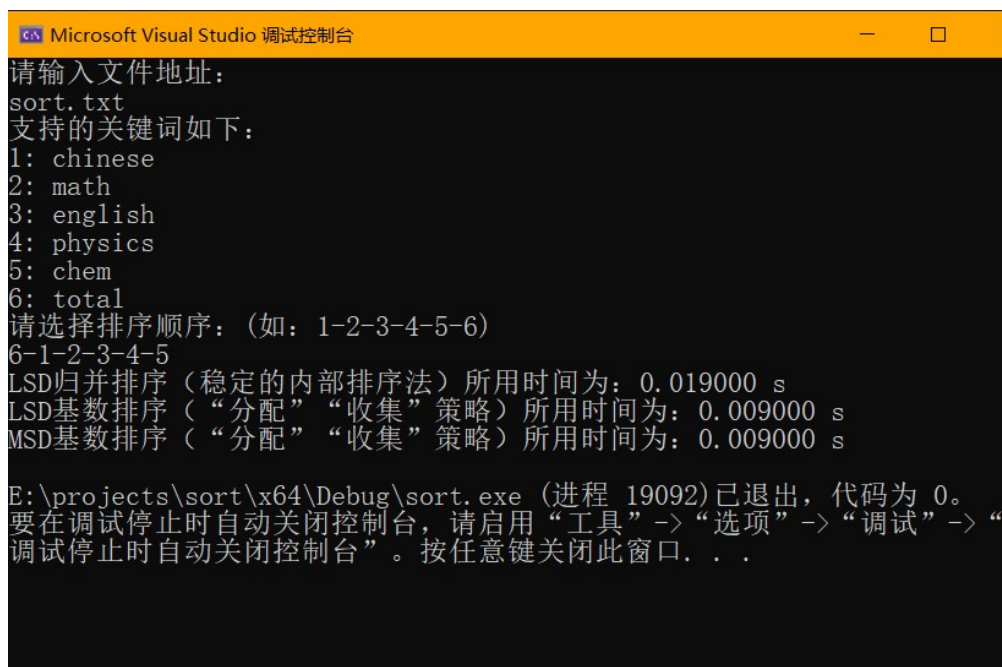
```

```
13 }
```

根据关键词的优先级循环启动归并排序程序。

```
1 int LSDMerge(SqlList* L, int* inquire) {
2     int i = 0;
3
4     for (i = 5; i >= 0; i--) {
5         if (!LSDMergeSort(L, 0, L->info.length - 1, inquire[i] - 1))
6             return ERROR;
7     }
8 }
```

2.5 测试方案与测试结果



```
Microsoft Visual Studio 调试控制台
请输入文件地址:
sort.txt
支持的关键词如下:
1: chinese
2: math
3: english
4: physics
5: chem
6: total
请选择排序顺序: (如: 1-2-3-4-5-6)
6-1-2-3-4-5
LSD归并排序 (稳定的内部排序法) 所用时间为: 0.019000 s
LSD基数排序 ("分配" "收集" 策略) 所用时间为: 0.009000 s
MSD基数排序 ("分配" "收集" 策略) 所用时间为: 0.009000 s

E:\projects\sort\x64\Debug\sort.exe (进程 19092) 已退出, 代码为 0。
要在调试停止时自动关闭控制台, 请启用 "工具" -> "选项" -> "调试" -> "
调试停止时自动关闭控制台"。按任意键关闭此窗口. . .
```

测试使用的文件可见: [点击此处跳转](#)。由于数据量较大, 此处仅展示终端输出结果, 如需结果文件可与作者联系。从终端输出的时间可以看出, 基数排序耗时少于归并排序, 这与理论相符合。

结语

历经一个多星期的努力, 最终还是在截止前完成了代码并进行了报告。这是我们第一次尝试小组合作完成一个项目, 虽然题目并不难, 但是互相的配合与交流有些困难。而且各位同学都是初学者, 代码中有很多想法没能实现, 以及可能存在一些问题, 还望见谅, 欢迎指正。

联系方式: bidingjun21@mails.ucas.ac.cn