

南京航空航天大学

课程设计

课 程	数据结构
班 级	1819001
学 号	161940233
姓 名	颜 宇 明
指导教师	秦 小 麟

目录

一、 区块链	4
(一) 数据结构	4
(二) 算法设计思想	4
(三) 源程序	4
(四) 测试数据及其结果	10
(五) 时间复杂度	11
(六) 改进方法	12
二、 迷宫问题	12
(一) 数据结构	12
(二) 算法设计思想	12
(三) 源程序	12
(四) 测试数据及其结果	19
(五) 时间复杂度	19
(六) 改进方法	19
三、 JSON 查找	20
(一) 数据结构	20
(二) 算法设计思想	20
(三) 源程序	20
(四) 测试数据及其结果	29
(五) 时间复杂度	30
(六) 改进方法	30
四、 公交线路提示	30
(一) 数据结构	30
(二) 算法设计思想	30
(三) 源程序	30
(四) 测试数据及其结果	37

(五) 时间复杂度	38
(六) 改进方法	39
五、 Hash 表应用	39
(一) 数据结构	39
(二) 算法设计思想	39
(三) 源程序	39
(四) 测试数据及其结果	44
(五) 时间复杂度	44
(六) 改进方法	44
六、 排序算法比较	44
(一) 数据结构	44
(二) 算法设计思想	44
(三) 源程序	45
(四) 测试数据及其结果	46
(五) 时间复杂度	47
(六) 改进方法	47
七、 地铁修建	47
(一) 数据结构	47
(二) 算法设计思想	47
(三) 源程序	48
(四) 测试数据及其结果	50
(五) 时间复杂度	50
(六) 改进方法	50
八、 社交网络图中结点的“重要性”计算	50
(一) 数据结构	50
(二) 算法设计思想	50
(三) 源程序	50

(四) 测试数据及其结果	54
(五) 时间复杂度	54
(六) 改进方法	54
九、 平衡二叉树操作的演示	54
(一) 数据结构	54
(二) 算法设计思想	55
(三) 源程序	55
(四) 测试数据及其结果	65
(五) 时间复杂度	65
(六) 改进方法	66
十、 Huffman 编码与解码	66
(一) 数据结构	66
(二) 算法设计思想	66
(三) 源程序	66
(四) 测试数据及其结果	77
(五) 时间复杂度	78
(六) 改进方法	78
十一、 家谱管理系统	78
(一) 数据结构	78
(二) 算法设计思想	78
(三) 源程序	78
(四) 测试数据及其结果	94
(五) 时间复杂度	96
(六) 改进方法	96
十二、 总结	96
(一) 代码行数	96
(二) 心得体会	96

一、区块链

(一) 数据结构

链表

(二) 算法设计思想

将区块设计成链表节点，每增加一个节点，计算校验码都要结合之前所有的节点的信息。

(三) 源程序

```
1  #ifndef LINKED_LIST_H
2  #define LINKED_LIST_H
3  class ADT_list{
4  public:
5      typedef struct node
6      {
7          int number;
8          char information[100];
9          int Checkcode;
10         node *next;
11     } node;
12     node *head;
13     int length = -1;
14
15     void InitList();
16     void DestoryList();
17     void ClearList();
18     bool ListEmpty();
19     int ListLength();
20     node* GetElem(int);
21     int LocateElem(int);
22     int PriorElem(int);
23     int NextElem(int);
24     void ListTraverse();
25     void CreateList(char*, int);
26     int CheckList();
27     void SetStr(int, const char*);
```

```

28     void InsertElem(char*);
29     int GetCheckcode();
30     void DeleteElem(int index);
31     void Reverse();
32 };
33 #endif

```

代码 1: Linked_list.h

```

1  #include <malloc.h>
2  #include <stdio.h>
3  #include <string.h>
4  #include "Linked_list.h"
5
6  void ADT_list::InitList(){
7      node *tmp = (node *)malloc(sizeof(node));
8      length = 0;
9      head = tmp;
10     tmp->next = NULL;
11 }
12
13 void ADT_list::DestoryList(){
14     node *p, *temp;
15     p = head;
16     while (p != NULL){
17         temp = p;
18         p = p->next;
19         free(temp);
20     }
21     length = -1;
22 }
23
24 void ADT_list::ClearList(){
25     node *p, *temp;
26     p = head->next;
27     while (p != NULL){
28         temp = p;
29         p = p->next;
30         free(temp);
31     }
32     length = 0;
33     head->next = NULL;

```

```

34     }
35
36     bool ADT_list::ListEmpty(){
37         if (length < 1) return true;
38         return false;
39     }
40
41     int ADT_list::ListLength(){
42         return length;
43     }
44
45     ADT_list::node* ADT_list::GetElem(int index){
46         node *p;
47         int i = 0;
48         p = head->next;
49         while (p != NULL){
50             if (index == ++i)
51                 return p;
52             p = p->next;
53         }
54         return NULL;
55     }
56
57     int ADT_list::LocateElem(int num){
58         node *p;
59         p = head->next;
60         int index = 0;
61         while (p != NULL){
62             if (p->number == num)
63                 return index;
64             p = p->next;
65             index++;
66         }
67         return -1;
68     }
69
70     int ADT_list::PriorElem(int cur_num){
71         node *p, *temp;
72         p = head->next;
73         while (p != NULL){
74             temp = p;
75             p = p->next;

```

```

76         if (p != NULL && p->number == cur_num)
77             return temp->number;
78     }
79     return 0;
80 }
81
82 int ADT_list::NextElem(int cur_num){
83     node *p, *temp;
84     p = head->next;
85     while (p != NULL){
86         temp = p;
87         p = p->next;
88         if (p != NULL && temp->number == cur_num)
89             return p->number;
90     }
91     return 0;
92 }
93
94 void ADT_list::ListTraverse(){
95     node *p;
96     p = head->next;
97     puts("Block Chain Output:");
98     while (p != NULL){
99         printf("%d %s %d\n", p->number, p->information, p->Checkcode);
100         p = p->next;
101     }
102     printf("\n");
103 }
104
105 void ADT_list::CreateList(char* str, int check){
106     node *p = head;
107     while (p->next != NULL) p = p->next;
108     node *tmp = (node *)malloc(sizeof(node));
109     p->next = tmp;
110     tmp->next = NULL;
111     tmp->number = length;
112     strcpy(tmp->information, str);
113     length++;
114     tmp->Checkcode = check;
115 }
116
117 int ADT_list::CheckList(){

```



```

118     node* p = head->next, *tmp = p;
119     if (!p) return 1;
120     while(p) {
121         int sum = 0;
122         for (int i = 0; i < strlen(p->information); i++)
123             sum += p->information[i];
124         if (p->number) {
125             node *temp = head;
126             sum += p->number;
127             while (temp->next != p) temp = temp->next;
128             sum += temp->Checkcode;
129         }
130         // printf("%d %d %d\n", sum % 113, p->Checkcode, p->number);
131         if (sum % 113 != p->Checkcode) return p->number + 100;
132         p = p->next;
133     }
134     return 1;
135 }
136
137 void ADT_list::SetStr(int index, const char* str){
138     node *p;
139     int i = 0;
140     p = head->next;
141     while (p && index != i++) p = p->next;
142     strcpy(p->information, str);
143     while(p) {
144         int sum = 0;
145         for (int i = 0; i < strlen(p->information); i++)
146             sum += p->information[i];
147         if (p->number) {
148             node *temp = head;
149             sum += p->number;
150             while (temp->next != p) temp = temp->next;
151             sum += temp->Checkcode;
152         }
153         // printf("%d %d %d\n", sum % 113, p->Checkcode, p->number);
154         // if (sum % 113 != p->Checkcode) return p->number + 100;
155         p->Checkcode = sum % 113;
156         p = p->next;
157     }
158 }
159

```

```

160 void ADT_list::InsertElem(char* str){
161     node *p = head;
162     while (p->next != NULL) p = p->next;
163     node *tmp = (node *)malloc(sizeof(node));
164     p->next = tmp;
165     tmp->next = NULL;
166     tmp->number = length;
167     strcpy(tmp->information, str);
168     length++;
169     tmp->Checkcode = GetCheckcode();
170 }
171 int ADT_list::GetCheckcode(){
172     node* p = head;
173     while (p->next != NULL) p = p->next;
174     int sum = 0;
175     for (int i = 0; i < strlen(p->information); i++)
176         sum += p->information[i];
177     if (length > 1) {
178         sum += p->number;
179         p = head;
180         while (p->next->next) p = p->next;
181         sum += p->Checkcode;
182     }
183     return sum % 113;
184 }
185 void ADT_list::DeleteElem(int index){
186     node *p, *temp;
187     int i = 0;
188     temp = head;
189     p = head->next;
190     length--;
191     while (p != NULL){
192         if (index != ++i) {
193             temp = p;
194             p = p->next;
195             continue;
196         }
197         temp->next = p->next;
198         free(p);
199     }
200 }
201

```

```

202 void ADT_list::Reverse(){
203     if (length < 1) return;
204     node *p, *temp, *tmp = NULL;
205     temp = p = head->next;
206     while(temp != NULL){
207         temp = p->next;
208         p->next = tmp;
209         tmp = p;
210         p = temp;
211     }
212     head->next = tmp;
213 }

```

代码 2: Linked_list.cpp

```

1  #include <stdio.h>
2  #include "Linked_list.h"
3  char str[100];
4  int num, check;
5  int main(){
6      ADT_list list;
7      list.InitList();
8      freopen("insertnode", "r", stdin);
9      while (scanf("%s", &str) != EOF) list.InsertElem(str);
10     list.ListTraverse();
11     list.ClearList();
12     freopen("CheckBlockChain", "r", stdin);
13     while (scanf("%s%d", &str, &check) != EOF) list.CreateList(str,
        check);
14     list.ListTraverse();
15     (num = list.CheckList()) < 100 ? puts("Accept!") : printf("The %
        dth node Error!\n\n", num - 100);
16     list.SetStr(1, "22");
17     list.ListTraverse();
18
19 }

```

代码 3: main.cpp

(四) 测试数据及其结果

文件 insertnode 创建区块链输入:

```
1      3
2      2
3      1
```

文件 CheckBlockChain 创建一个错误的区块链：

```
1      3 51
2      2 102
3      1 4
```

main.cpp 输出为：

```
1      Block Chain Output:
2      0 3 51
3      1 2 102
4      2 1 40
5
6      Block Chain Output:
7      0 3 51
8      1 2 102
9      2 1 4
10
11     The 2th node Error!
12
13     Block Chain Output:
14     0 3 51
15     1 22 39
16     2 1 90
```

由以上输出结果可知，题目要求所有完全达到了。

(五) 时间复杂度

$O(n)$

(六) 改进方法

算校验码不需要每次都把前面所有的节点都遍历一遍，可以直接用最后一个节点的校验码。

二、迷宫问题

(一) 数据结构

用栈来实现深度优先搜索

(二) 算法设计思想

先生成迷宫，设定起点，然后用栈实现的深度优先搜索来寻找迷宫的出口，迷宫保证路径唯一。

(三) 源程序

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "Sequence_Stack.h"
4  #include <fstream>
5  using namespace std;
6  int maze[100][100], n, visited[100][100];
7  int xx[] = {-1, 1, 0, 0};
8  int yy[] = {0, 0, -1, 1};
9  int endx, endy, flag;
10 ADT_Stack Stack;
11 string temp;
12 int main(){
13     ifstream ReadFile("maze");
14     while(getline(ReadFile, temp)) n++;
15
16     Stack.InitStack();
17     freopen("maze", "r", stdin);
18     for (int i = 1; i <= n; i++)
19         for (int j = 1; j <= n; j++)
20             scanf("%d", &maze[i][j]);
21     for (int i = n; i >= 1; i--)
22         if (!maze[i][n]) endx = i, endy = n;
```

```

23 Pos *point = new Pos;
24 point->x = 2;
25 point->y = 1;
26 point->last = NULL;
27 Stack.Push(*point);
28 visited[point->x][point->y] = 1;
29 while (Stack.rear) {
30     Pos pp = Stack.Pop();
31     Pos *p = new Pos;
32     p->x = pp.x;
33     p->y = pp.y;
34     p->last = pp.last;
35     visited[p->x][p->y] = 1;
36     if (p->x == endx && p->y == endy) {
37         point = p;
38         while (point)
39             maze[point->x][point->y] = 2, point = point->last;
40
41         for (int i = 0; i <= n; i++) {
42             for (int j = 0; j <= n; j++) {
43                 if (maze[i][j] == 0)
44                     printf(" ");
45                 else if (maze[i][j] == 2)
46                     printf("■");
47                 else
48                     printf("░");
49             }
50             printf("\n");
51         }
52         return 0;
53     }
54     for (int i = 0; i < 4; i++)
55         if (!maze[p->x + xx[i]][p->y + yy[i]] && !visited[p->x + xx[
56             i][p->y + yy[i]]) {
57             point = new Pos;
58             point->x = p->x + xx[i];
59             point->y = p->y + yy[i];
60             point->last = p;
61             Stack.Push(*point);
62         }
63 }

```

64 }

代码 4: main.cpp

```
1  #include<stdio.h>
2  #include<fstream>
3  #include<Windows.h>
4  #include<time.h>
5  #include<math.h>
6
7  //地图长度L，包括迷宫主体40，外侧的包围的墙体2，最外侧包围路径2（之后会
   解释）
8  #define L 44
9
10 //墙和路径的标识
11 #define WALL 0
12 #define ROUTE 1
13 using namespace std;
14 //控制迷宫的复杂度，数值越大复杂度越低，最小值为0
15 static int Rank = 0;
16 int startx = 2, starty = 1;
17
18 //生成迷宫
19 void CreateMaze(int **maze, int x, int y);
20
21 int main(void) {
22     srand((unsigned)time(NULL));
23
24     int **Maze = (int**)malloc(L * sizeof(int *));
25     for (int i = 0; i < L; i++) {
26         Maze[i] = (int*)calloc(L, sizeof(int));
27     }
28
29     //最外围层设为路径的原因，为了防止挖路时挖出边界，同时为了保护迷宫主
   体外的一圈墙体被挖穿
30     for (int i = 0; i < L; i++){
31         Maze[i][0] = ROUTE;
32         Maze[0][i] = ROUTE;
33         Maze[i][L - 1] = ROUTE;
34         Maze[L - 1][i] = ROUTE;
35     }
36 }
```

```

37 //创造迷宫, (2, 2) 为起点
38 CreateMaze(Maze, startx, starty + 1);
39
40 //画迷宫的入口和出口
41 Maze[2][1] = ROUTE;
42
43 //由于算法随机性, 出口有一定概率不在 (L-3,L-2) 处, 此时需要寻找出口
44 for (int i = L - 3; i >= 0; i--) {
45     if (Maze[i][L - 3] == ROUTE) {
46         Maze[i][L - 2] = ROUTE;
47         break;
48     }
49 }
50
51 //画迷宫
52 for (int i = 0; i < L; i++) {
53     for (int j = 0; j < L; j++) {
54         if (Maze[i][j] == ROUTE) {
55             printf(" ");
56         }
57         else {
58             printf("■");
59         }
60     }
61     printf("\n");
62 }
63 ofstream out("maze");
64 if (out.is_open()){
65     for (int i = 1; i < L - 1; i++) {
66         for (int j = 1; j < L - 1; j++) {
67             if (Maze[i][j] == ROUTE)
68                 out << "0 ";
69             else
70                 out << "1 ";
71         }
72         out << "\n";
73     }
74 }
75
76 for (int i = 0; i < L; i++) free(Maze[i]);
77 free(Maze);
78

```



```

79     return 0;
80 }
81
82 void CreateMaze(int **maze, int x, int y) {
83     maze[x][y] = ROUTE;
84
85     //确保四个方向随机
86     int direction[4][2] = { { 1,0 }, { -1,0 }, { 0,1 }, { 0,-1 } };
87     for (int i = 0; i < 4; i++) {
88         int r = rand() % 4;
89         int temp = direction[0][0];
90         direction[0][0] = direction[r][0];
91         direction[r][0] = temp;
92
93         temp = direction[0][1];
94         direction[0][1] = direction[r][1];
95         direction[r][1] = temp;
96     }
97
98     //向四个方向开挖
99     for (int i = 0; i < 4; i++) {
100         int dx = x;
101         int dy = y;
102
103         //控制挖的距离，由Rank来调整大小
104         int range = 1 + (Rank == 0 ? 0 : rand() % Rank);
105         while (range > 0) {
106             dx += direction[i][0];
107             dy += direction[i][1];
108
109             //排除掉回头路
110             if (maze[dx][dy] == ROUTE) {
111                 break;
112             }
113
114             //判断是否挖穿路径
115             int count = 0;
116             for (int j = dx - 1; j < dx + 2; j++) {
117                 for (int k = dy - 1; k < dy + 2; k++) {
118                     //abs(j - dx) + abs(k - dy) == 1 确保只判断九宫格的
119                     //四个特定位置
120                     if (abs(j - dx) + abs(k - dy) == 1 && maze[j][k] ==

```

```

120             ROUTE) {
121                 count++;
122             }
123         }
124     }
125     if (count > 1) {
126         break;
127     }
128
129     //确保不会挖穿时，前进
130     --range;
131     maze[dx][dy] = ROUTE;
132 }
133
134 //没有挖穿危险，以此为节点递归
135 if (range <= 0) {
136     CreateMaze(maze, dx, dy);
137 }
138 }
139 }

```

代码 5: MazeGeneration.cpp

```

1  #ifndef SEQUENCE_Stack_H
2  #define SEQUENCE_Stack_H
3  typedef struct Pos
4  {
5      int x;
6      int y;
7      Pos *last;
8  } Pos;
9  class ADT_Stack{
10 public:
11     Pos Stack[9999];
12     int rear = -1;
13     void InitStack();
14     void DestoryStack();
15     void ClearStack();
16     bool StackEmpty();
17     int StackLength();
18     Pos GetTop();

```

```

19     void StackTraverse();
20     void Push(Pos);
21     Pos Pop();
22     Pos operator [] (int);
23     // int LocateElem(int num);
24     // int PriorElem(int cur_num);
25     // int NextElem(int cur_num);
26     // int SetElem(int index, int num);
27     // void InsertElem(int index, int num);
28     // void DeleteElem(int index);
29     // void Remove();
30     // void Bubble_Sort();
31     // void Select_sort();
32     // ADT_Stack Union(ADT_Stack);
33     // void Josephus(int);
34 };
35 #endif

```

代码 6: Sequence_Stack.h

```

1 CC = g++
2
3 OUT1 = MazeGeneration.exe
4 OBJ1 = MazeGeneration.o
5
6 OUT2 = main.exe
7 OBJ2 = main.o Sequence_Stack.o
8
9 IN = main.cpp MazeGeneration.cpp Sequence_Stack.cpp
10
11 build: $(OUT1) $(OUT2)
12
13 run: $(OUT1) $(OUT2)
14     @./$(OUT1);./$(OUT2)
15
16 clean:
17     @rm -f *.o $(OUT1) $(OUT2)
18
19 $(OUT1): $(OBJ1)
20     @$(CC) $(OBJ1) -o $(OUT1)
21
22 $(OUT2): $(OBJ2)

```

```
23     @$(CC) $(OBJ2) -o $(OUT2)
24
25 $(OBJ): $(IN)
26     @$(CC) -c $(IN)
```

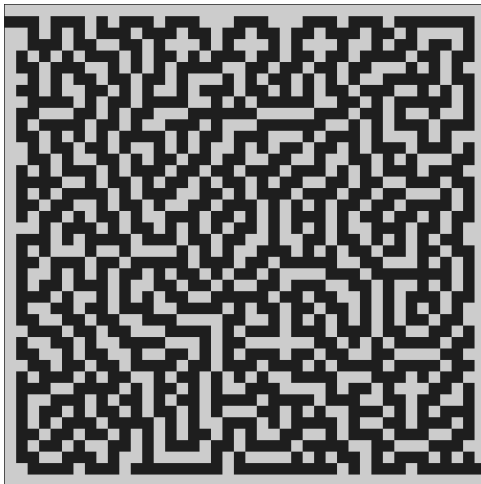
代码 7: Makefile

(四) 测试数据及其结果

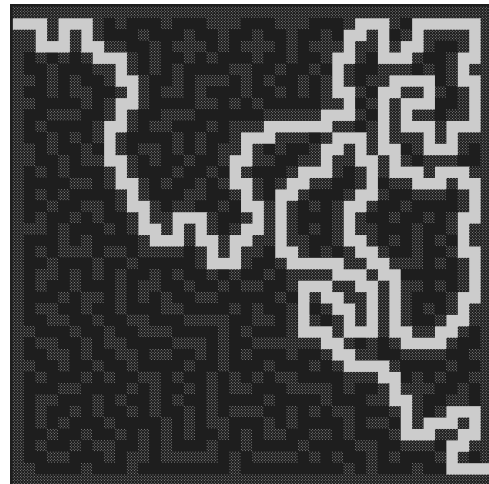
在项目目录下运行命令：

```
1 clear;make clean;make run
```

运行结果为：



(a) 自动生成的迷宫



(b) 基于 DFS 的迷宫路径可视化

图 1: 实验结果

由以上结果可知，实现题目的全部要求。

(五) 时间复杂度

$$O(n^2)$$

(六) 改进方法

可以尝试用 Dijkstra 算法来解决这个问题。

三、JSON 查找

(一) 数据结构

孩子兄弟表示法，列表

(二) 算法设计思想

对 json 源文件进行词法分析，再进行语法分析，同时构建孩子兄弟树存储与表示

(三) 源程序

```
1  #include <iostream>
2  #include <fstream>
3  #include <cstring>
4  #include <list>
5  using namespace std;
6  enum token_type
7  {
8      unknown_type = 0,
9      LeftBrace,
10     RightBrace,
11     TokString,
12     Comma,
13     Colon
14 };
15
16 enum node_type
17 {
18     ErrorType = 0,
19     Root,
20     NotExist,
21     NodeString,
22     NodeObj,
23 };
24
25 list<string> str_search_list; // this list is set for search-function of
    node
26 struct node
```

```

27 {
28     int type;
29     string context;
30     list<node> children;
31     node &operator=(const node &p)
32     {
33         type = p.type;
34         context = p.context;
35         children = p.children;
36         return *this;
37     }
38     void child_search(list<string>::iterator &iter)
39     {
40         for (list<node>::iterator i = this->children.begin(); i != this
41             ->children.end(); ++i)
42             if (i->context == *iter)
43             {
44                 switch (i->type)
45                 {
46                     case NodeString:
47                         cout << "STRING " << i->children.front().context <<
48                             endl;
49                         return;
50                         break;
51                     case NodeObj:
52                         ++iter;
53                         if (iter == str_search_list.end())
54                             cout << "OBJECT" << endl;
55                         else
56                             i->child_search(iter);
57                         return;
58                         break;
59                 }
60             }
61         cout << "NOTEXIST" << endl;
62         return;
63     }
64     void root_search()
65     {
66         list<string>::iterator iter = str_search_list.begin();
67         for (list<node>::iterator i = this->children.begin(); i != this
68             ->children.end(); ++i)

```

```

66         if (i->context == *iter)
67         {
68             switch (i->type)
69             {
70                 case NodeString:
71                     cout << "STRING " << i->children.front().context <<
72                         endl;
73                     return;
74                     break;
75                 case NodeObj:
76                     ++iter;
77                     if (iter == str_search_list.end())
78                         cout << "OBJECT" << endl;
79                     else
80                         i->child_search(iter);
81                     return;
82                     break;
83             }
84         }
85         cout << "NOTEXIST" << endl;
86         return;
87     }
88 };
89
90 struct token
91 {
92     int type;
93     string context;
94 };
95
96 list<char> resource;
97 list<token> token_list;
98 list<token>::iterator this_token;
99 string temp_string;
100 node root;
101
102 bool lexer() //词法分析
103 {
104     // scan the whole resource code and generate tokens
105     string tmp;
106     for (list<char>::iterator iter = resource.begin(); iter != resource.

```

```

end(); ++iter)
107 {
108     token temp_token;
109     if (*iter == '{')
110     {
111         temp_token.type = LeftBrace;
112         temp_token.context = "{}";
113         token_list.push_back(temp_token);
114     }
115     else if (*iter == '}')
116     {
117         temp_token.type = RightBrace;
118         temp_token.context = "{}";
119         token_list.push_back(temp_token);
120     }
121     else if (*iter == ',')
122     {
123         temp_token.type = Comma;
124         temp_token.context = ",";
125         token_list.push_back(temp_token);
126     }
127     else if (*iter == ':')
128     {
129         temp_token.type = Colon;
130         temp_token.context = ":";
131         token_list.push_back(temp_token);
132     }
133     else if (*iter == '\"')
134     {
135         ++iter;
136         tmp = "";
137         while (*iter != '\\')
138         {
139             if (*iter == '\\')
140             {
141                 ++iter;
142                 if (iter != resource.end() && (*iter == '\\ ' || *
                    iter == '\\'))
143                     tmp += *iter;
144             else
145             {
146                 // '\\' with no '\\' or '\"' after it

```



```

147         cout << "[error] incorrect string: " << tmp << "
148             \\ " << endl;
149         return false;
150     }
151     else
152         tmp += *iter;
153     ++iter;
154     if (iter == resource.end() || *iter == '\n')
155     {
156         // string with an incorrect end
157         cout << "[error] incorrect string: " << tmp << " .
158             must end with a '\"'\"' . " << endl;
159         return false;
160     }
161     temp_token.type = TokString;
162     temp_token.context = tmp;
163     token_list.push_back(temp_token);
164 }
165 }
166 return true;
167 }
168
169 node node_gen()
170 {
171     node ret;
172     ret.type = ErrorType;
173     ++this_token;
174     if (this_token->type == TokString)
175     {
176         ret.type = NodeString;
177         ret.context = this_token->context;
178     }
179     else if (this_token->type == LeftBrace)
180     {
181         ret.type = NodeObj;
182         while (this_token != token_list.end())
183         {
184             ++this_token;
185             if (this_token == token_list.end())
186                 {

```

```

187         cout << "[error] expect a '}' at the end." << endl;
188         ret.type = ErrorType;
189         return ret;
190     }
191     // }
192     if (this_token->type == RightBrace)
193         break;
194     if (this_token->type == TokString)
195     {
196         node tmp;
197         tmp.context = this_token->context;
198         ++this_token;
199         // :
200         if (this_token == token_list.end() || this_token->type
201             != Colon)
202         {
203             cout << "[error] expect a \':\' this token: " << tmp
204                 .context << endl;
205             ret.type = ErrorType;
206             return ret;
207         }
208         ++this_token;
209         if (this_token == token_list.end() || (this_token->type
210             != TokString && this_token->type != LeftBrace))
211         {
212             cout << "[error] expect a string or object after
213                 this token: " << tmp.context << endl;
214             ret.type = ErrorType;
215             return ret;
216         }
217         switch (this_token->type)
218         {
219             // string must get a child typed string
220             case TokString:
221                 tmp.type = NodeString;
222                 --this_token;
223                 tmp.children.push_back(node_gen());
224                 break;
225             // object get a return type object so it's children list
226             // must set to the same as return object
227             case LeftBrace:
228                 tmp.type = NodeObj;

```

```

224         --this_token;
225         tmp.children = node_gen().children;
226         break;
227     }
228     // if get error type ,parse failed
229     if (tmp.children.back().type == ErrorType)
230     {
231         ret.type = ErrorType;
232         return ret;
233     }
234     ret.children.push_back(tmp);
235     ++this_token;
236     if (this_token->type == RightBrace)
237         --this_token;
238     else if (this_token->type != Comma)
239     {
240         cout << "[error] expect a ',' here." << endl;
241         ret.type = ErrorType;
242         return ret;
243     }
244     }
245     }
246 }
247 else
248     cout << "[error] unknown type." << endl;
249 return ret;
250 }
251
252 bool parse() //语法分析
253 {
254     root.type = Root;
255     this_token = token_list.begin();
256     // {
257     if (this_token->type != LeftBrace)
258     {
259         cout << "[error] expect a '{' at the beginning." << endl;
260         return false;
261     }
262     while (this_token != token_list.end())
263     {
264         ++this_token;
265         if (this_token == token_list.end())

```

```

266     {
267         cout << "[error] expect a '}' at the end." << endl;
268         return false;
269     }
270     // }
271     if (this_token->type == RightBrace)
272         break;
273     if (this_token->type == TokString)
274     {
275         node tmp;
276         tmp.context = this_token->context;
277         ++this_token;
278         if (this_token == token_list.end() || this_token->type !=
                Colon)
279         {
280             cout << "[error] expect a \':\' this token: " << tmp.
                context << endl;
281             return false;
282         }
283         ++this_token;
284         if (this_token == token_list.end() || (this_token->type !=
                TokString && this_token->type != LeftBrace))
285         {
286             cout << "[error] expect a string or object after this
                token: " << tmp.context << endl;
287             return false;
288         }
289         switch (this_token->type)
290         {
291             // string must get a child typed string
292             case TokString:
293                 tmp.type = NodeString;
294                 --this_token;
295                 tmp.children.push_back(node_gen());
296                 break;
297             // object get a return type object so it's children list
                must set to the same as return object
298             case LeftBrace:
299                 tmp.type = NodeObj;
300                 --this_token;
301                 tmp.children = node_gen().children;
302                 break;

```

```

303         }
304         // if get error type ,parse failed
305         if (tmp.children.back().type == ErrorType)
306             return false;
307         root.children.push_back(tmp);
308         ++this_token;
309         if (this_token->type == RightBrace)
310             --this_token;
311         else if (this_token->type != Comma)
312         {
313             cout << "[error] expect a \',\' here." << endl;
314             return false;
315         }
316     }
317 }
318 return true;
319 }
320
321 int main()
322 {
323     int n, m;
324     resource.clear();
325     token_list.clear();
326     ifstream fin("input");
327     if (fin.fail())
328     {
329         cout << "Open file failed." << endl;
330         return 0;
331     }
332     fin >> n >> m;
333     getline(fin, temp_string); // 避免回车
334     for (int i = 0; i < n; ++i)
335     {
336         getline(fin, temp_string);
337         for (int j = 0; j < temp_string.length(); ++j)
338             resource.push_back(temp_string[j]);
339     }
340     if (!lexer())
341     {
342         cout << "[lexer] error occurred ,stop." << endl;
343         return 0;
344     }

```

```

345     if (!parse())
346     {
347         cout << "[parse] error occurred, stop." << endl;
348         return 0;
349     }
350     for (int i = 0; i < m; ++i)
351     {
352         getline(fin, temp_string);
353         str_search_list.clear();
354         string t = "";
355         for (int j = 0; j < temp_string.length(); ++j)
356         {
357             if (temp_string[j] != ' ')
358                 t += temp_string[j];
359             else
360             {
361                 str_search_list.push_back(t);
362                 t = "";
363             }
364         }
365         str_search_list.push_back(t);
366         root.root_search();
367     }
368     return 0;
369 }

```

代码 8: lexer.cpp

(四) 测试数据及其结果

程序运行后，结果为：

```

1  STRING John
2  OBJECT
3  STRING NewYork
4  NOTEXIST
5  STRING "hello"

```

(五) 时间复杂度

时间复杂度为

$$O(1)$$

(六) 改进方法

程序已经比较完整了，没有改进方法。

四、公交线路提示

(一) 数据结构

栈，队列，邻接矩阵，Dijkstra 算法，广度优先搜索

(二) 算法设计思想

- 1) 最短路径：利用邻接矩阵 Dijkstra 算法，计算两点之间的最短路径。
- 2) 最少转乘：采用广度优先遍历算法，判断起点所涉及的每辆公交车经过的所有站点，如果有终点，则无需转车。若无终点，则让起点所涉及的每辆公交车经过的所有站点进入队列，通过 Visited 数组判断该站点是否已经访问过。队列元素出队，继续判断该点所涉及的每辆公交车经过的所有站点，重复以上步骤，直至找到所需终点。

(三) 源程序

```
1  #include <map>
2  #include <list>
3  #include <queue>
4  #include <string>
5  #include <vector>
6  #include <fstream>
7  #include <cstring>
8  #include <iostream>
9  #include <algorithm>
10 using namespace std;
11 #define INF 0x3f3f3f3f
```

```

12 #define maxn 40000
13
14 vector<string> split(const string& str, const string& delim) {
15     vector<string> res;
16     if( "" == str) return res;
17     //先将要切割的字符串从string类型转换为char*类型
18     char * strs = new char[str.length() + 1];
19     strcpy(strs, str.c_str());
20
21     char * d = new char[delim.length() + 1];
22     strcpy(d, delim.c_str());
23
24     char *p = strtok(strs, d);
25     while(p) {
26         string s = p; //分割得到的字符串转换为string类型
27         res.push_back(s); //存入结果数组
28         p = strtok(NULL, d);
29     }
30     return res;
31 }
32
33 class FindKey
34 {
35 private:
36     int num;
37 public:
38     FindKey(int n) :num(n){}
39     bool operator()(map<string, int>::value_type item){
40         return item.second == num;
41     }
42 };
43
44 class BusStationDatabase
45 {
46 private:
47     typedef struct Pos
48     {
49         int StationNum;
50         int BusNum;
51         Pos *last;
52     } Pos;
53

```



```

54     list<string> stations;
55     int StationNumber;
56     int RouteNumber;
57     int StartStaionNum;
58     map<string, int> StationMap;           //站点及其编号的映射
59     map<int, vector<string> > BusInfo;    //第几路公交车及其经过站
        点的映射
60     map<string, vector<int> > StationInfo; //每个站点都有哪些公交车
        经过
61     queue<Pos> Queue;
62
63     struct Edge{
64         int v;
65         int len;
66         int next;
67     } edge[111111];
68     int head[maxn], cnt = 0;
69     int from, to;
70     string start, end;
71     int visit[maxn];
72     int dis[maxn], path[maxn];
73
74
75 public:
76     BusStationDatabase(){
77         StationNumber = 1;
78         RouteNumber = 0;
79     }
80     void LoadData(){
81         ifstream inputfile("NanjingBusRoute");
82         if(inputfile.fail()){
83             cout<<"Open failed."<<endl;
84             inputfile.close();
85             return;
86         }
87         string input_line;
88         while(!inputfile.eof()){
89             getline(inputfile, input_line);
90             vector<string> splitarr = split(input_line, " ");
91             splitarr[0].pop_back();
92             vector<string> curBusStation = split(splitarr[1], ",");
93             BusInfo[stoi(splitarr[0])] = curBusStation;

```

```

94         for (auto i : curBusStation){
95             auto key = StationMap.find(i);
96             if (key == StationMap.end())
97                 StationMap[i] = StationNumber++;
98             StationInfo[i].push_back(stoi(splitarr[0]));
99         }
100     }
101     inputfile.close();
102     StationNumber--;
103     RouteNumber = BusInfo.size();
104     return;
105 }
106 void BuildGraph(){
107     for (auto i : BusInfo) {
108         for(int j = 0; j < i.second.size() - 1; j++){
109             from = StationMap[i.second[j]];
110             to = StationMap[i.second[j + 1]];
111             addedge(from, to, 1);
112             addedge(to, from, 1);
113         }
114     }
115 }
116 void PrintLeastStationRoute(int s){
117     if (!s) return;
118     PrintLeastStationRoute(path[s]);
119     auto it = find_if(StationMap.begin(), StationMap.end(),
120                     FindKey(s));
121     if (it != StationMap.end())
122         if (path[s])
123             cout << ">" << (*it).first;
124         else
125             cout << (*it).first;
126 }
127 void LeastStationRoute(){
128     freopen("StartEndStation", "r", stdin);
129     cin >> start >> end;
130     memset(path, 0, sizeof(path));
131     memset(visit, 0, sizeof(visit));
132     int flag = 0;
133     auto key = StationMap.find(start);
134     if (key == StationMap.end()) flag = 1;
135     key = StationMap.find(end);

```

```

135         if (key == StationMap.end()) flag = 1;
136         if (flag) {
137             cout << "error!" << endl;
138             return;
139         }
140         dijkstra();
141         for (int i = 1; i <= StationNumber; i++)
142             if (i == StationMap[end]) {
143                 cout << "经过站点最少: " << dis[i] << "站" << endl;
144                 int p = path[i];
145                 PrintLeastStationRoute(i);
146                 cout << endl << endl;
147             }
148     }
149     void LeastChangeRoute(){
150         memset(path, 0, sizeof(path));
151         memset(visit, 0, sizeof(visit));
152         Pos *point = new Pos;
153         point->BusNum = 0;
154         point->last = NULL;
155         point->StationNum = StationMap[start];
156         Queue.push(*point);
157         visit[point->StationNum] = 1;
158         while (!Queue.empty()) {
159             Pos front = Queue.front();
160             Queue.pop();
161             Pos *p = new Pos;
162             p->StationNum = front.StationNum;
163             p->BusNum = front.BusNum;
164             p->last = front.last;
165             visit[p->StationNum] = 1;
166             if (p->StationNum == StationMap[end]) {
167                 point = ReverseList(p);
168                 point->BusNum = point->last->BusNum;
169                 cout << "转车次数最少: " << endl;
170                 while (point->last){
171                     int tmp = point->StationNum;
172                     auto ss = find_if(StationMap.begin(), StationMap
                        .end(), FindKey(tmp));
173                     auto ee = find_if(StationMap.begin(), StationMap
                        .end(), FindKey(point->last->StationNum));
174                     auto StartStation = find(BusInfo[point->last->

```

```

        BusNum].begin(), BusInfo[point->last->BusNum
        ].end(), (*ss).first);
175     auto EndStation = find(BusInfo[point->last->
        BusNum].begin(), BusInfo[point->last->BusNum
        ].end(), (*ee).first);
176     if (StartStation > EndStation) reverse(BusInfo[
        point->last->BusNum].begin(), BusInfo[point
        ->last->BusNum].end());
177     int flag = 0;
178     for (auto i : BusInfo[point->last->BusNum]) {
179         if (i == (*ss).first || flag) {
180             if (i == (*ee).first){
181                 cout << i << "(" << point->last->
                    BusNum << "路)";
182                 break;
183             }
184             cout << i << "(" << point->last->BusNum
                    << "路)->";
185             flag = 1;
186         }
187     }
188     if (point->last->last) cout << endl << "转" <<
        endl;
189     point = point->last;
190 }
191 return;
192 }
193 auto StationName = find_if(StationMap.begin(),
        StationMap.end(), FindKey(p->StationNum));
194 for (auto i : StationInfo[(*StationName).first]) { //能
        经过该站点所有公交车的列表
195     for (auto j : BusInfo[i]) { // 该公交车经过的所有站
        点列表
196         if (!visit[StationMap[j]]) { //如果这个站点没有
            被访问过，就加到队列里
197             point = new Pos;
198             point->StationNum = StationMap[j];
199             point->BusNum = i;
200             point->last = p;
201             Queue.push(*point);
202         }
203     }
}

```

```

204         }
205     }
206 }
207 Pos* ReverseList(Pos *root){
208     Pos *p, *temp, *tmp = NULL;
209     temp = p = root;
210     while(temp){
211         temp = p->last;
212         p->last = tmp;
213         tmp = p;
214         p = temp;
215     }
216     return tmp;
217 }
218 void addedge(int from, int to, int len){
219     edge[++cnt].v = to;
220     edge[cnt].len = len;
221     edge[cnt].next = head[from];
222     head[from] = cnt;
223 }
224 void dijkstra(){
225     for (int i = 1; i <= StationNumber; i++)
226         dis[i] = INF;
227     int temp = StationMap[start];
228     dis[temp] = 0;
229     int minn;
230     while (!visit[temp]){
231         visit[temp] = 1;
232         for (int j = head[temp]; j; j = edge[j].next)
233             if (!visit[edge[j].v] && dis[edge[j].v] > dis[temp]
                + edge[j].len)
234                 dis[edge[j].v] = dis[temp] + edge[j].len, path[
                    edge[j].v] = temp;
235         minn = INF;
236         for (int j = 1; j <= StationNumber; j++)
237             if (!visit[j] && minn > dis[j])
238                 minn = dis[j], temp = j;
239     }
240 }
241 int get_StationNumber(){return StationNumber;}
242 int get_route_number(){return RouteNumber;}
243 list<string>& get_station_name_list(){return stations;}

```

```

244 };
245 BusStationDatabase lib;
246 int main()
247 {
248     lib.LoadData();
249     lib.BuildGraph();
250     lib.LeastStationRoute();
251     lib.LeastChangeRoute();
252     return 0;
253 }

```

代码 9: BusRoute.cpp

(四) 测试数据及其结果

起点：干休所站

终点：墨香路站

```

1  经过站点最少：10 站
2  干休所站->河路道站->中央门北站->中央门东站->龙蟠路南京站西站->龙蟠路南京
   站东站->新庄广场南站->长途东站站->樱铁村站->经五立交站->墨香路站
3
4  转车次数最少：
5  干休所站(1路)->河路道站(1路)->中央门北站(1路)->中央门南站(1路)
6  转
7  中央门南站(22路)->中央门东站(22路)->龙蟠路南京站西站(22路)->龙蟠路南京站
   东站(22路)->新庄广场西站(22路)->新庄广场北站(22路)->曹后村站(22路)->
   江南公交一公司站(22路)->红山森林动物园站(22路)->十字街站(22路)->省中
   中西医结合医院站(22路)->红山路
8  迈皋桥站(22路)->华电路站(22路)->长营村站(22路)->北苑新村站(22路)->月苑小
   区站(22路)->月苑南路站(22路)->墨香路站(22路)

```

代码 10: 测试用例 1

起点：干休所站

终点：国际青年文化中心站

```

1  经过站点最少：24 站
2  干休所站->河路道站->中央门北站->中央门南站->许府巷站->玄武湖公园站->中央
   路鼓楼

```

3	站->中山路珠江路北站->新街口南站->三元巷站->中山南路升州路站->中华路瞻园路站->
4	钓鱼台站->窑湾街站->雨花西路站->能仁里站->龙福山庄站->秋叶村站->梦都大街庐山路
5	站->庐山路牡丹江街站->庐山路奥体大街站->庐山路中央公园站->庐山路嘉陵江东街站->
6	江东中路江山大街站->国际青年文化中心站
7	
8	转车次数最少:
9	千休所站(25路)->河路道站(25路)->中央门北站站(25路)->中央门南站站(25路)->许府巷
10	站(25路)->玄武湖公园站(25路)->中山路鼓楼站(25路)->鼓楼医院站(25路)->中山路珠江
11	路南站(25路)->新街口东站(25路)->大行宫西站(25路)->大行宫东站(25路)->中山东路逸
12	仙桥(毗卢寺)站(25路)->西安门站(25路)->瑞金北村站(25路)->瑞金路站(25路)
13	转
14	瑞金路站(7路)->解放南路站(7路)->大光路站(7路)->通济门站(7路)->建康路大中桥站(7
15	路)->建康路夫子庙站(7路)->升州路三山街站(7路)->评事街站(7路)->水西门站(7路)->莫
16	愁湖公园南门站(7路)->水西门大街大士茶亭站(7路)->茶亭东街站(7路)->江东门纪念馆站
17	(7路)->水西门大街江东门站(7路)->江东万达广场站(7路)->江东中路集庆门大街站(7路)->江东中路应天大街站(7路)->江东中路月安街站(7路)->江东中路兴隆大街站(7路)->兴隆
18	大街燕山路站(7路)->清竹园南门站(7路)->兴隆大街华山路站(7路)->乐山路梦都大街站(7路)->金陵图书馆站(7路)->奥体中心西门站(7路)->乐山路富春江西街站(7路)->乐山路楠
19	溪江西街站(7路)->乐山路河西大街站(7路)->乐山路白龙江西街站(7路)->乐山路仁恒江湾
20	城站(7路)->乐山路金沙江西街站(7路)->青年文化中心东门站(7路)->国际青年文化中心站
21	(7路)

代码 11: 测试用例 2

(五) 时间复杂度

1) Dijkstra 算法时间复杂度:

$$O(V^2)$$

2) 广度优先搜索时间复杂度:

$$O(V)$$

(六) 改进方法

由于节点太多, 转乘3次以上会导致广度优先搜索是速度变慢, 可以使用减少不必要的步骤, 来减少时间开销。

五、Hash 表应用

(一) 数据结构

Hash 表

(二) 算法设计思想

利用开放定址 (自行选择和设计定址方案) 和链地址法构造 hash 表

(三) 源程序

```
1  #include "ChainAddress.h"
2  #include <iostream>
3  #include <fstream>
4  #include <vector>
5  #include <cstring>
6  using namespace std;
7  Hash_Table HashList;
8  vector<string> split(const string& str, const string& delim) {
9      vector<string> res;
10     if( "" == str) return res;
11     //先将要切割的字符串从string类型转换为char*类型
12     char * strs = new char[str.length() + 1];
13     strcpy(strs, str.c_str());
14
15     char * d = new char[delim.length() + 1];
16     strcpy(d, delim.c_str());
17
18     char *p = strtok(strs, d);
19     while(p) {
```



```

20     string s = p; //分割得到的字符串转换为string类型
21     res.push_back(s); //存入结果数组
22     p = strtok(NULL, d);
23 }
24 return res;
25 }
26
27 int main(){
28     ifstream inputfile("input");
29     if(inputfile.fail()){
30         cout<<"Open failed."<<endl;
31         inputfile.close();
32         return 0;
33     }
34     string input_line;
35     while(!inputfile.eof()){
36         getline(inputfile, input_line);
37         vector<string> splitarr = split(input_line, ",");
38         // for (auto i : splitarr) {
39         //     cout << i << endl;
40         // }
41         // strcpy(HashList[const_cast<char*>(splitarr[0].c_str())]->
42             Identifier, const_cast<char*>(splitarr[0].c_str()));
43         // strcpy(HashList[const_cast<char*>(splitarr[0].c_str())]->name
44             , const_cast<char*>(splitarr[1].c_str()));
45         // strcpy(HashList[const_cast<char*>(splitarr[0].c_str())]->time
46             , const_cast<char*>(splitarr[3].c_str()));
47
48         HashList[const_cast<char*>(splitarr[0].c_str())]->Identifier =
49             const_cast<char*>(splitarr[0].c_str());
50         HashList[const_cast<char*>(splitarr[0].c_str())]->name =
51             const_cast<char*>(splitarr[1].c_str());
52         HashList[const_cast<char*>(splitarr[0].c_str())]->line = stoi(
53             splitarr[2]);
54         HashList[const_cast<char*>(splitarr[0].c_str())]->time =
55             const_cast<char*>(splitarr[3].c_str());
56         HashList[const_cast<char*>(splitarr[0].c_str())]->TotalMileage =
57             stoi(splitarr[4]);
58         // cout << HashList[const_cast<char*>(splitarr[0].c_str())]->
59             name << "*" << endl;
60         // cout << HashList[const_cast<char*>(splitarr[0].c_str())]->
61             line << "*" << endl;

```

```

52         // cout << HashList[const_cast<char*>(splitarr[0].c_str())]->
           time << "*" << endl;
53         // cout << HashList[const_cast<char*>(splitarr[0].c_str())]->
           TotalMileage << "*" << endl;
54     }
55     inputfile.close();
56     HashList.printTable();
57     return 0;
58 }

```

```

1  #include <bits/stdc++.h>
2  #include "ChainAddress.h"
3  using namespace std;
4  struct Entry a[KEY];
5  unsigned long long HashTable[KEY];
6  int head = 0;
7
8  Information::Information(char *s) {
9      Identifier = strdup(s);
10 }
11 Information::~Information() { free(Identifier); Identifier = NULL; }
12 bool Information::Equ(char* a) {
13     return strcmp(Identifier, a) == 0;
14 }
15
16 Entry::Entry() { }
17 Entry::~Entry() { delete info; info = NULL; }
18
19 unsigned long long Hash_Table::Get_Hash(char* Identifier) {
20     unsigned long long Ans = 0;
21     for (int i = 0; Identifier[i]; ++i)
22         Ans = Ans * P + Identifier[i];
23     return Ans;
24 }
25
26 Information*& Hash_Table::New_Entry(char* Identifier, unsigned long long
    Hash) {
27     int nowindex = first[Hash % KEY];
28     HashTable[head++] = Hash % KEY;
29     while(a[nowindex].next != 0) nowindex = a[nowindex].next;
30     if (first[Hash % KEY] == 0) {

```

```

31         first[Hash % KEY] = ++e;
32         a[e].next = 0;
33     }
34     else
35         a[nowindex].next = ++e;
36     a[e].Hash = Hash;
37     a[e].info = new Information(Identifier);
38     return a[e].info;
39 }
40
41 Information*& Hash_Table::operator [] (char * Identifier) {
42     unsigned long long Hash = Get_Hash(Identifier);
43     for (int i = first[Hash % KEY]; i; i = a[i].next)
44         if (a[i].info && a[i].info -> Equ(Identifier))
45             return a[i].info;
46     return New_Entry(Identifier, Hash);
47 }
48
49 bool Hash_Table::Count(char* Identifier) {
50     unsigned long long Hash = Get_Hash(Identifier);
51     for (int i = first[Hash % KEY]; i; i = a[i].next)
52         if (a[i].info && a[i].info -> Equ(Identifier))
53             return true;
54     return false;
55 }
56
57 void Hash_Table::printTable() {
58     printf("\n
59         +-----+-----+-----+-----+-----+
60         n");
61     printf("|                                     Hash   Table
62         | \n");
63     printf("
64         +-----+-----+-----+-----+-----+
65         n");
66     printf("|          IDnumber          |   Name   |   Line   |   Time   |
67         Total   | \n");
68     printf("
69         +-----+-----+-----+-----+-----+
70         n");
71     for(int i; i < head; i++){
72         for (unsigned long long j = first[HashTable[i]]; j; j = a[j].

```

```

        next){
65         // cout << a[j].info->Identifier << a[j].info->name << a[j].
            info->line << a[j].info->time << a[j].info->TotalMileage
                << endl;
66         printf("|%19s | %10s |%10d|%12s|%11d|\n", a[j].info->
            Identifier, a[j].info->name, a[j].info->line, a[j].info
                ->time, a[j].info->TotalMileage);
67         printf("
            +-----+-----+-----+-----+
            n");
68
69     }
70 }
71 }

```

```

1  #ifndef SYMBOL_TABLE_H
2  #define SYMBOL_TABLE_H
3
4  const int KEY = 76543;
5  const int P = 107;
6  struct Information {
7      char *Identifier; // 该符号名
8      char *name; // 符号对应的类型
9      int line; // 该符号对应的值
10     char *time; // 该符号对应的值
11     int TotalMileage; // 定义所在的行号
12
13     Information(char *);
14     ~Information();
15     bool Equ(char*);
16 };
17
18 struct Entry{
19     unsigned long long Hash;
20     Information *info;
21     int next;
22     Entry();
23     ~Entry();
24 };
25
26

```

```

27 class Hash_Table {
28 public:
29     int first[KEY], e;
30     unsigned long long Get_Hash(char*);
31     Information*& New_Entry(char*, unsigned long long);
32     Information*& operator [] (char *);
33     bool Count(char*);
34     void printTable();
35 };
36 #endif

```

(四) 测试数据及其结果

(五) 时间复杂度

$O(1)$

(六) 改进方法

使用其他避免冲突的办法。

六、排序算法比较

(一) 数据结构

各种排序算法，都是数组。

(二) 算法设计思想

1) 直接插入排序：将一条记录插入到已排好的有序表中，从而得到一个新的、记录数量增 1 的有序表。

2) 希尔排序：希尔排序是把记录按下标的一定增量分组，对每组使用直接插入排序算法排序；随着增量逐渐减少，每组包含的关键词越来越多，当增量减至 1 时，整个文件恰被分成一组，算法便终止。

3) 冒泡排序：依次比较两个相邻的元素，如果顺序（如从大到小、首字母从 Z 到 A）错误就把他们交换过来。走访元素的工作是重复地进行直到没有相邻元素

需要交换，也就是说该元素列已经排序完成。

4) 快速排序：挑选基准值：从数列中挑出一个元素，称为“基准”（pivot），分割：重新排序数列，所有比基准值小的元素摆放在基准前面，所有比基准值大的元素摆在基准后面（与基准值相等的数可以到任何一边）。在这个分割结束之后，对基准值的排序就已经完成，递归排序子序列：递归地将小于基准值元素的子序列和大于基准值元素的子序列排序。

5) 选择排序：第一次从待排序的数据元素中选出最小（或最大）的一个元素，存放在序列的起始位置，然后再从剩余的未排序元素中寻找到最小（大）元素，然后放到已排序的序列的末尾。以此类推，直到全部待排序的数据元素的个数为零。

6) 堆排序：利用堆这种数据结构所设计的一种排序算法。堆积是一个近似完全二叉树的结构，并同时满足堆积的性质：即子结点的键值或索引总是小于（或者大于）它的父节点。

7) 归并排序：将已有序的子序列合并，得到完全有序的序列；即先使每个子序列有序，再使子序列段间有序。若将两个有序表合并成一个有序表，称为二路归并。

8) 基数排序：将整数按位数切割成不同的数字，然后按每个位数分别比较。具体做法是：将所有待比较数值统一为同样的数位长度，数位较短的数前面补零。然后，从最低位开始，依次进行一次排序。这样从最低位排序一直到最高位排序完成以后，数列就变成一个有序序列。

（三）源程序

代码 12: Sort.cpp

```
1 #include <stdio.h>
2 #include <iostream>
3 #include <fstream>
4 #include <time.h>
5 using namespace std;
6 int num = 50000, n, a = num;
7 int main(void) {
8     srand((unsigned)time(NULL));
9     ofstream out("1Sample");
```

```

10     while (num)
11         out << a - num-- << endl;
12     out.close();
13
14     out.open("2Sample");
15     num = a;
16     while (num-- > 0) out << num << endl;
17     out.close();
18
19     num = 10;
20     for (int i = 3; i <= num; i++){
21         out.open(to_string(i) + "Sample");
22         n = a;
23         while (n-- > 0)
24             out << rand() % a << endl;
25         out.close();
26     }
27     return 0;
28 }

```

代码 13: NumGeneration.cpp

```

1 CC = g++
2 SOURCE := NumGeneration.cpp Sort.cpp
3
4 build:
5     @$(foreach var,$(SOURCE),\
6         $(CC) -c $(var); \
7         $(CC) $(subst .cpp,.o,$(var)) -o $(subst .cpp,.exe,$(var)); \
8         ./${subst .cpp,.exe,$(var)};\
9     )
10
11 clean:
12     @rm -f *.o *.exe *Sample

```

代码 14: Makefile

(四) 测试数据及其结果

测试数据用 NumGeneration.cpp 生成 10 个每个含有 50000 个元素的样本。

	Insert	Shell	Bubble	Select	Heap	Merge	Radix	Quick
1Sample:	0.000s	0.003s	3.064s	2.685s	0.010s	0.000s	0.015s	0.008s

3	2Sample:	4.707s	0.003s	5.329s	2.825s	0.009s	0.004s	0.004s	0.010s
4	3Sample:	2.413s	0.010s	7.257s	2.761s	0.011s	0.007s	0.004s	0.014s
5	4Sample:	2.793s	0.010s	7.300s	2.763s	0.011s	0.007s	0.004s	0.014s
6	5Sample:	2.566s	0.011s	7.392s	2.733s	0.011s	0.007s	0.004s	0.014s
7	6Sample:	2.472s	0.010s	7.217s	2.662s	0.012s	0.007s	0.004s	0.014s
8	7Sample:	2.367s	0.010s	6.910s	2.611s	0.011s	0.006s	0.004s	0.014s
9	8Sample:	2.302s	0.010s	6.812s	2.656s	0.011s	0.006s	0.003s	0.014s
10	9Sample:	2.412s	0.011s	7.042s	2.717s	0.011s	0.007s	0.004s	0.014s
11	10Sample:	2.337s	0.010s	7.519s	2.815s	0.016s	0.008s	0.005s	0.014s

代码 15: 结果

(五) 时间复杂度

排序方法	时间复杂度	最坏情况	空间复杂度	稳定性
直接插入排序	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
希尔排序	$O(n^{1.5})$	$O(n^{1.5})$	$O(1)$	不稳定
冒泡排序	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
快速排序	$O(n\log_2 n)$	$O(n^2)$	$O(n\log_2 n)$	不稳定
简单选择排序	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	不稳定
归并排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$	稳定
基数排序	$O(dn)$	$O(dn)$	$O(rd)$	稳定

图 2: 排序算法比较

(六) 改进方法

每种排序算法都是固定的，不需要改进。

七、地铁修建

(一) 数据结构

无向图、并查集

(二) 算法设计思想

先读入所有的边，并按权值从小到大排序。之后按权从小到大归并与边相连的两个顶点，并通过并查集检查顶点是否相连，当顶点 1 与顶点 N 连通时，当前边的权值即为最少的施工天数中最大的天数。

(三) 源程序

```
1    #include <iostream>
2    #include <vector>
3    #include <algorithm>
4    #include <string.h>
5    #define NONE -1
6
7    using namespace std;
8
9    struct Edge{
10        int v1;
11        int v2;
12        int weight;
13
14        Edge(int v1In, int v2In, int weightIn): v1(v1In), v2(v2In),
            weight(weightIn){}
15
16        bool operator<(const Edge& e)
17        {
18            return weight < e.weight;
19        }
20    };
21
22    int findRoot(int fa[], int v)
23    {
24        if (fa[v] == NONE)
25            return v;
26        else
27            return fa[v] = findRoot(fa, fa[v]);
28    }
29
30    int N, M;
31    int main()
32    {
33        cin >> N >> M;
34        vector<Edge> edges;
35
36        int v1, v2, weight;
37        for (int i = 1; i <= M; i++)
38        {
39            cin >> v1 >> v2 >> weight;
```

```

40         edges.push_back(Edge(v1, v2, weight));
41     }
42
43     int father[N+1];
44     memset(father, NONE, sizeof(father)); //并查集初始化
45
46     sort(edges.begin(), edges.end()); //按照施工天数从小到大排序
47
48     //模拟施工过程
49     for (int i = 0; i < M; i++)
50     {
51         //归并两棵子树
52         father[findRoot(father, edges[i].v1)] = findRoot(father,
53             edges[i].v2);
54
55         if (findRoot(father, 1) == findRoot(father, N))
56         {
57             cout << edges[i].weight;
58             return 0;
59         }
60     }
61
62     /*
63     input sample1:
64     6 6
65     1 2 4
66     2 3 4
67     3 6 7
68     1 4 2
69     4 5 5
70     5 6 6
71
72     output sample1:
73     6
74
75     input sample2:
76
77     output sample2:
78
79     input sample3:
80

```

```

81     output sample3:
82
83     */

```

代码 16: main.cpp

(四) 测试数据及其结果

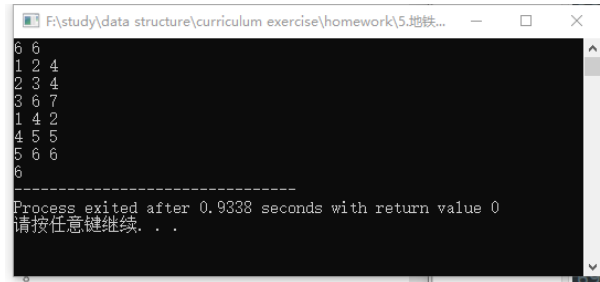


图 3: 实验截图

(五) 时间复杂度

$$O(n \log_2 n)$$

(六) 改进方法

本题有多种解法，还可以采用 dijkstra 算法的思想求解。

八、 社交网络图中结点的“重要性”计算

(一) 数据结构

无向图

(二) 算法设计思想

核心为求无权图的单源最短路径，方法依旧采用 BFS，然后重复 n 次即可。

(三) 源程序

```

1    #include <iostream>

```

```

2      #include <queue>
3      #include <string.h>
4
5      #define INT_MAX 0x3f3f3f3f
6      #define N_MAX 1001
7      #define ERROR -1.0
8
9      using namespace std;
10
11     void ReadIn();
12     bool Calculate();
13     void Print();
14
15     bool graph[N_MAX][N_MAX]; //存放该无权无向图
16     double Cc[N_MAX]; //紧密度中心性
17     int N, M;
18     int K;
19
20     int main()
21     {
22         ReadIn(); //读入数据
23
24         if (!Calculate())
25             memset(Cc, 0, sizeof(Cc)); //若是非连通图，全设0
26
27         Print(); //输出结果
28     }
29
30     void ReadIn()
31     {
32         cin >> N >> M;
33
34         for (int i = 1; i <= M; i++)
35         {
36             //建立无权无向图
37             int v1, v2;
38             cin >> v1 >> v2;
39             graph[v1][v2] = true;
40             graph[v2][v1] = true;
41         }
42     }
43

```

```

44 //若图连通，返回start结点的紧密度中心性
45 //若图不连通返回ERROR
46 double BFS(int start)
47 {
48     queue<int> queue;
49     int dist[N+1];
50     memset(dist, INT_MAX, sizeof(dist));
51
52     queue.push(start);
53     dist[start] = 0;//起点
54
55     int vertex;
56     int count = 0;//统计被访问结点个数
57     double sum = 0;//总的最短距离
58     while (!queue.empty())
59     {
60         vertex = queue.front();
61         queue.pop();
62         count++;
63
64         for (int i = 1; i <= N; i++)
65         {
66             if ((dist[i] == INT_MAX) && (graph[vertex][i])){
67                 dist[i] = dist[vertex] + 1;
68                 sum += (double)dist[i];
69                 queue.push(i);
70             }
71         }
72     }
73
74     if (count < N)
75         return ERROR;
76     else if(count == N)
77         return sum;
78 }
79
80 bool Calculate()
81 {
82     for (int i = 1; i <= N; i++)
83     {
84         double result = BFS(i);
85         if (result == ERROR)

```

```

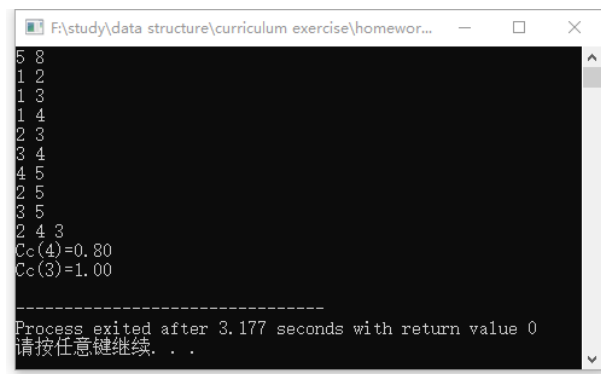
86         return false;//图不连通
87     else
88         Cc[i] = (double)(N-1)/result;
89     }
90
91     return true;
92 }
93
94 void Print()
95 {
96     cin >> K;
97     int vertex;
98     for (int i = 1; i <= K; i++)
99     {
100         cin >> vertex;
101         printf("Cc(%d)=%.2f\n", vertex, Cc[vertex]);
102     }
103 }
104
105 /*
106
107 //此题可直接在PTA上提交
108
109 input sample1:
110 5 8
111 1 2
112 1 3
113 1 4
114 2 3
115 3 4
116 4 5
117 2 5
118 3 5
119 2 4 3
120
121 output sample1:
122 Cc(4)=0.80
123 Cc(3)=1.00
124
125 input sample2:
126
127 output sample2:

```

```
128
129     input sample3:
130
131     output sample3:
132
133     */
```

代码 17: main.cpp

(四) 测试数据及其结果



```
F:\study\data structure\curriculum exercise\homewor...
5 8
1 2
1 3
1 4
2 3
3 4
4 5
2 5
3 5
2 4 3
Cc(4)=0.80
Cc(3)=1.00
-----
Process exited after 3.177 seconds with return value 0
请按任意键继续. . .
```

图 4: 实验截图

(五) 时间复杂度

$$O(n^2)$$

(六) 改进方法

暂无改进想法。

九、平衡二叉树操作的演示

(一) 数据结构

平衡二叉排序树

(二) 算法设计思想

插入：

- 1) 寻找数据插入位置。若小于当前根结点，将其递归插入至左子树，若大于，将其递归插入至右子树，若等于则插入。
- 2) 向父结点回溯。若当前二叉树不平衡，回溯至最小子树并调整其至平衡。

删除：

- 1) 寻找应当删除的结点。若小于当前根结点，则在左子树中寻找，若大于，将在右子树中寻找，直至找到，执行步骤 2。
- 2) 若该结点为叶结点，执行步骤 4。若非叶节点，执行步骤 3。
- 3) 若其左子树比右子树高，将该结点数据与左子树最大结点的数据交换，再删除左子树的最大结点。若右子树比左子树高同理。
- 4) 删除该叶结点。
- 5) 向父结点回溯。若当前二叉树不平衡，回溯至最小子树并调整其至平衡。

(三) 源程序

```
1      #include <iostream>
2      #include <string>
3      #include "avl_tree.h"
4
5      /* run this program using the console pauser or add your own getch,
6         system("pause") or input loop */
7
8      int main(int argc, char** argv)
9      {
10         AVL_Tree avl_tree;
11         const string fileName("test1.txt");
12         fstream dataFile(fileName, ios::in);
13         avl_tree.Create(fileName);
14
15         //1.插入操作
16         avl_tree.Insert(avl_tree.root, 9999);
17         avl_tree.Insert(avl_tree.root, -9999);
18         avl_tree.Insert(avl_tree.root, 2);
```



```

18     avl_tree.print(avl_tree.root);
19     cout << endl;
20
21     //2.查找操作
22     if (avl_tree.Find(2))
23         cout << "YES" << endl;
24     else
25         cout << "NO" << endl;
26     if (avl_tree.Find(80))
27         cout << "YES" << endl;
28     else
29         cout << "NO" << endl;
30     if (avl_tree.Find(0))
31         cout << "YES" << endl;
32     else
33         cout << "NO" << endl;
34     avl_tree.print(avl_tree.root);
35     cout << endl;
36
37     //3.删除操作
38     avl_tree.Delete(avl_tree.root, 5);
39     avl_tree.Delete(avl_tree.root, 0);
40     avl_tree.Delete(avl_tree.root, 9999);
41     avl_tree.Delete(avl_tree.root, 91);
42     avl_tree.Delete(avl_tree.root, 5);
43     avl_tree.Delete(avl_tree.root, 30);
44     avl_tree.print(avl_tree.root);
45     cout << endl;
46     return 0;
47 }

```

代码 18: main.cpp

```

1     #ifndef AVL_TREE_H
2     #define AVL_TREE_H
3
4     #include <iostream>
5     #include <fstream>
6     #include <queue>
7     #include <algorithm>
8     using namespace std;
9

```

```

10     typedef int ElemType;
11     typedef struct BiNode* Position;
12     typedef Position BiTree;
13
14     struct BiNode{
15         ElemType data;//关键字
16         BiTree left;//左子树
17         BiTree right;//右子树
18
19         BiNode(ElemType dataIn);
20     };
21
22     class AVL_Tree{
23     public:
24         AVL_Tree();
25         ~AVL_Tree();//销毁树
26
27         bool Create(const string fileName);//建树
28         Position Find(ElemType data);//查找值为data的结点
29         Position FindMax(BiTree root);//查找以root为根的子树中，最大
           值结点
30         Position FindMin(BiTree root);//查找以root为根的子树中，最小
           值结点
31
32         Position Insert(BiTree root, ElemType data);//插入结点
33         Position Delete(BiTree root);//删除子树
34         Position Delete(BiTree root, ElemType data);//删除值为data的
           结点
35
36         int height(BiTree root);//返回以root为根结点的树高
37         Position LL_rotation(BiTree root);
38         Position LR_rotation(BiTree root);
39         Position RL_rotation(BiTree root);
40         Position RR_rotation(BiTree root);
41
42         void print(BiTree root);//以中序遍历方式输出结果
43
44     // private:
45         BiTree root;//树根
46     };
47
48     //选做题 9.32

```

```

49     void find_a_b(AVL_Tree& avl_tree, BiTree root, ElemType x, ElemType*
        a, ElemType* b);
50
51     #endif

```

代码 19: avl_tree.h

```

1     #include "avl_tree.h"
2
3     BiNode::BiNode(ElemType dataIn): data(dataIn)
4     {
5         left = right = NULL;
6     }
7
8     AVL_Tree::AVL_Tree()
9     {
10        root = NULL;
11    }
12
13    AVL_Tree::~~AVL_Tree()
14    {
15        Delete(root);
16    }
17
18    bool AVL_Tree::Create(const string fileName)
19    {
20        fstream dataFile(fileName, ios::in);
21        if (!dataFile){
22            cout << "打开文件" << fileName << "失败!" << endl;
23            return false;
24        }
25
26        int data;
27        while (!dataFile.eof())
28        {
29            dataFile >> data;
30            // cout << data << " "; //debug
31            root = Insert(root, data);
32        }
33
34        dataFile.close();
35        return true;

```

```

36     }
37
38     Position AVL_Tree::Find(ElemType data)
39     {
40         Position pos = root;
41         while (pos != NULL)
42         {
43             if (data < pos->data)
44                 pos = pos->left;
45             else if (data > pos->data)
46                 pos = pos->right;
47             else
48                 return pos;
49         }
50         return NULL;
51     }
52
53     Position AVL_Tree::FindMax(BiTree root)
54     {
55         BiNode* p = root;
56         if (p == NULL)
57             return NULL;
58         else
59         {
60             while (p->right != NULL)
61             {
62                 p = p->right;
63             }
64         }
65         return p;
66     }
67
68     Position AVL_Tree::FindMin(BiTree root)
69     {
70         BiNode* p = root;
71         if (p == NULL)
72             return NULL;
73         else
74         {
75             while (p->left != NULL)
76             {
77                 p = p->left;

```

```

78         }
79     }
80     return p;
81 }
82
83 Position AVL_Tree::Insert(BiTree root, ElemType data)
84 {
85     if(root == NULL)
86     {
87         root = new BiNode(data);
88     }
89     else
90     {
91         if(data < root->data)
92         {
93             //插入到左子树
94             root->left = Insert(root->left, data);
95             //计算平衡因子, 再判断如何旋转
96             if(height(root->left) - height(root->right) == 2)
97             {
98                 if(data < root->left->data)
99                     root = LL_rotation(root);
100                 else if(data > root->left->data)
101                     root = LR_rotation(root);
102             }
103         }
104         else if(data > root->data)
105         {
106             //插入到右子树
107             root->right = Insert(root->right, data);
108             //计算平衡因子, 再判断如何旋转
109             if(height(root->right) - height(root->left) == 2)
110             {
111                 if(data > root->right->data)
112                     root = RR_rotation(root);
113                 else if(data < root->right->data)
114                     root = RL_rotation(root);
115             }
116         }
117     }
118     return root;
119 }

```

```

120
121 Position AVL_Tree::Delete(BiTree root)
122 {
123     if (root != NULL){
124         root->left = Delete(root->left);
125         root->right = Delete(root->right);
126         delete root;
127         return NULL;
128     }
129 }
130
131 Position AVL_Tree::Delete(BiTree root, ElemType data)
132 {
133     //思路:
134     //1.找到结点 (若不为叶结点, 转化为叶结点)
135     //2.删除
136     //3.平衡调整
137     if (root == NULL)
138         return NULL;
139     else
140     {
141         if (data < root->data) //1.找到结点
142         {
143             root->left = Delete(root->left, data);
144         }
145         else if (data > root->data) //1.找到结点
146         {
147             root->right = Delete(root->right, data);
148         }
149         else
150         {
151             //2.删除
152             if ((root->left == NULL) && (root->right == NULL))
153             {
154                 delete root;
155                 if (root == this->root)//特殊情况
156                     this->root = NULL;
157                 return NULL;
158             }
159             else//1. (若不为叶结点, 转化为叶结点)
160             {
161                 Position pos;

```

```

162         if (height(root->left) > height(root->right))
163         {
164             pos = FindMax(root->left);
165             //交换data与左子树最大值
166             root->data = pos->data;
167             pos->data = data;
168             root->left = Delete(root->left, data);
169         }
170         else
171         {
172             pos = FindMin(root->right);
173             //交换data与右子树最小值
174             root->data = pos->data;
175             pos->data = data;
176             root->right = Delete(root->right, data);
177         }
178     }
179 }
180
181
182 //3.平衡调整
183 // if (height(root->left) - height(root->right) == 2)
184 // {
185 //     if (height(root->left->left) >= height(root->left->right))
186 //         root = LL_rotation(root);
187 //     else
188 //         root = LR_rotation(root);
189 // }
190 // else if (height(root->left) - height(root->right) == -2)
191 // {
192 //     if (height(root->right->right) >= height(root->right->left))
193 //         root = RR_rotation(root);
194 //     else
195 //         root = RL_rotation(root);
196 // }
197
198     return root;
199 }
200
201 int AVL_Tree::height(BiNode* root)
202 {
203     if (root == NULL)

```

```

204         return 0;
205     else
206         return 1 + max(height(root->left), height(root->right));
207 }
208
209 Position AVL_Tree::LL_rotation(BiTree root)
210 {
211     BiNode* p1 = root->left;
212     BiNode* p2 = p1->right;
213     root->left = p2;
214     p1->right = root;
215     return p1;
216 }
217
218 Position AVL_Tree::LR_rotation(BiTree root)
219 {
220     BiNode* p1 = root->left;
221     BiNode* p2 = p1->right;
222     root->left = p2->right;
223     p2->right = root;
224     p1->right = p2->left;
225     p2->left = p1;
226     return p2;
227 }
228
229 Position AVL_Tree::RL_rotation(BiTree root)
230 {
231     BiNode* p1 = root->right;
232     BiNode* p2 = p1->left;
233     root->right = p2->left;
234     p2->left = root;
235     p1->left = p2->right;
236     p2->right = p1;
237     return p2;
238 }
239
240 Position AVL_Tree::RR_rotation(BiTree root)
241 {
242     BiNode* p1 = root->right;
243     BiNode* p2 = p1->left;
244     root->right = p2;
245     p1->left = root;

```



```

246         return p1;
247     }
248
249     void AVL_Tree::print(BiTree root)
250     {
251         if (root == this->root)
252             cout << endl << "平衡二叉树的中序遍历如下: " << endl;
253         if (root == NULL)
254             return;
255         else if (root != NULL){
256             print(root->left);
257             cout << root->data << " ";
258             print(root->right);
259         }
260
261         // queue<BiNode*> queue;
262         // queue.push(root);
263         // BiNode* p;
264         // while (!queue.empty())
265         // {
266         //     p = queue.front();
267         //     queue.pop();
268         //
269         //     if (p->left != NULL)
270         //         queue.push(p->left);
271         //     if (p->right != NULL)
272         //         queue.push(p->right);
273         //
274         //     cout << p->data << " 对应结点树高为: " << height(p) << endl;
275         // }
276     }
277
278     //选做题 9.32
279     void find_a_b(AVL_Tree& avl_tree, BiTree root, ElemType x, ElemType*
        a, ElemType* b)
280     {
281         if (root == NULL)
282             return;
283         else
284             {
285                 if (root->data < x)
286                     {

```

```

287         *a = root->data;
288         find_a_b(avl_tree, root->right, x, a, b);
289     }
290     else if (root->data > x)
291     {
292         *b = root->data;
293         find_a_b(avl_tree, root->left, x, a, b);
294     }
295     else
296     {
297         Position pos1 = avl_tree.FindMax(root->left);
298         Position pos2 = avl_tree.FindMin(root->right);
299
300         if (pos1 != NULL)
301             *a = pos1->data;
302         if (pos2 != NULL)
303             *b = pos2->data;
304     }
305 }
306 }

```

代码 20: avl_tree.cpp

(四) 测试数据及其结果

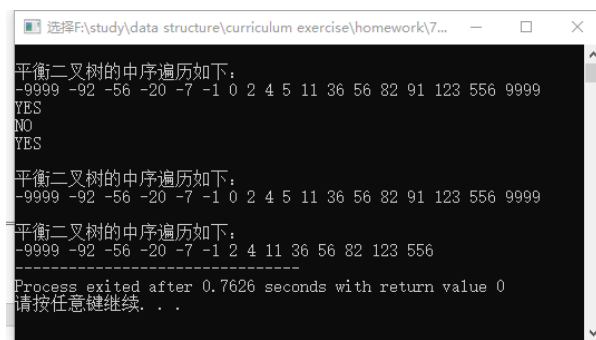


图 5: 实验截图

(五) 时间复杂度

$$O(\log_2 n)$$

(六) 改进方法

将递归函数改为非递归函数，进一步加深理解。

十、Huffman 编码与解码

(一) 数据结构

Huffman 树

(二) 算法设计思想

利用最小堆建树，为每个结点编码时采用层序遍历的方式编码。为每个字符编码时只需在对应的 Huffman 树上查找其对应的叶结点即可。文件读写时，二进制文件中前四个字节存放数据的有效位数，第五个字节开始存放数据，解码时先读入前四字节确定数据的总位数，再进行解码。解码过程即编码逆过程。

(三) 源程序

```
1      #include <iostream>
2      #include "huffmantree.h"
3
4      /* run this program using the console pauser or add your own getch,
5         system("pause") or input loop */
6
7      int main(int argc, char** argv)
8      {
9          const string sourceFile("source.txt");//未编码文件名
10         // const string sourceFile("test.txt");//测试文件
11
12         //要求一
13         CharTable charTable;
14         charTable.calculateFreq(sourceFile);//统计词频
15         HuffmanTree hTree(charTable);//建树
16         hTree.show("Huffman.txt");//打印词频及编码，并写入文件
17
18         //要求二
19         hTree.encode(sourceFile, "code.dat");//给文件编码
```

```

20     //要求三
21     hTree.decode("code.dat", "解码结果.txt");//给文件解码
22
23     return 0;
24 }

```

代码 21: main.cpp

```

1     #ifndef HUFFMANTREE_H
2     #define HUFFMANTREE_H
3
4     #include <iostream>
5     #include <fstream>
6     #include <string>
7     #include <queue>
8     #include <iomanip>
9     #include <string.h>
10    #include <bitset>
11    #include "chartable.h"
12
13    using namespace std;
14
15    struct BiNode{
16        char data;//英文字符
17        int weight;//字符权值，即其词频
18        BiNode* left;
19        BiNode* right;
20        string codes;//编码结果
21
22        BiNode(char c, int w);
23        BiNode(int w);
24        ~BiNode();
25
26        void show();//打印词频及编码
27    };
28
29    //重写仿函数以建立最小堆
30    struct cmp{
31        bool operator()(BiNode* n1, BiNode* n2);
32    };
33
34    class HuffmanTree{

```

```

35     public:
36         HuffmanTree(CharTable& charTable); //建树
37         ~HuffmanTree(); //销毁Huffman tree (递归)
38
39         void encode(const string& filename1, const string& filename2
40                     ); //编码
41         void decode(const string& filename1, const string& filename2
42                     ); //解码
43
44         void show(const string& filename); //打印词频及编码，并写入文件
45
46     private:
47         BiNode* root; //树根
48         map<char, string> codeTable; //编码表
49 };

```

代码 22: huffmantree.h

```

1  #include "huffmantree.h"
2
3  const int width = 25; //设置输出域宽
4
5  bool cmp::operator()(BiNode* n1, BiNode* n2)
6  {
7      return n1->weight > n2->weight;
8  }
9
10 BiNode::BiNode(char c, int w): data(c), weight(w)
11 {
12     left = right = NULL;
13 }
14
15 BiNode::BiNode(int w): weight(w)
16 {
17     left = right = NULL;
18 }
19
20 BiNode::~BiNode()
21 {

```

```

22     if (left != NULL)
23     {
24         delete left;
25         left = NULL;
26     }
27     if (right != NULL)
28     {
29         delete right;
30         right = NULL;
31     }
32 }
33
34 void BiNode::show()
35 {
36     cout << "char: " << data
37         << setw(width) << "frequency: " << weight
38         << setw(width) << "codes: " << codes
39         << endl;
40 }
41
42 HuffmanTree::HuffmanTree(CharTable& charTable)
43 {
44     //建立最小堆
45     priority_queue<BiNode*, vector<BiNode*>, cmp> minHeap;
46
47     auto iter = charTable.table.begin();
48     for (; iter != charTable.table.end(); iter++)
49     {
50         BiNode* p = new BiNode(iter->first, iter->second);
51         minHeap.push(p);
52     }
53
54     //建树
55     BiNode* min1;
56     BiNode* min2;
57     while (minHeap.size() != 1)
58     {
59         min1 = minHeap.top();
60         minHeap.pop();
61         min2 = minHeap.top();
62         minHeap.pop();
63

```

```

64         BiNode* root = new BiNode(min1->weight + min2->weight);
65         root->left = min1;
66         root->right = min2;
67
68         minHeap.push(root);
69     }
70
71     //记录树根
72     root = minHeap.top();
73
74     //给每个结点编码（基于层序遍历）
75     queue<BiNode*> queue;
76     BiNode* p = root;
77
78     if (p == NULL)
79         return;
80
81     queue.push(p);
82     while (!queue.empty())
83     {
84         p = queue.front();
85         queue.pop();
86
87         if (p->left != NULL){
88             p->left->codes = p->codes + "0";
89             queue.push(p->left);
90         }
91         if (p->right != NULL){
92             p->right->codes = p->codes + "1";
93             queue.push(p->right);
94         }
95
96         //将叶结点加入编码表
97         if ((p->left == NULL) && (p->right == NULL))
98         {
99             codeTable.insert(pair<char, string>(p->data, p->codes));
100         }
101     }
102 }
103
104 HuffmanTree::~HuffmanTree()
105 {

```

```

106     if (root != NULL)
107     {
108         delete root; //递归实现
109         root = NULL;
110     }
111 }
112
113 void HuffmanTree::show(const string& filename)
114 {
115     fstream file(filename.data(), ios::out);
116     if (file.bad()){
117         cout << "cann't open " << filename << endl;
118         return;
119     }
120
121     queue<BiNode*> queue;
122     BiNode* p = root;
123
124     if (p == NULL)
125         return;
126
127     cout << "字符:  "
128          << setw(width) << "词频:  "
129          << setw(width) << "编码:  "
130          << endl;
131     file << "字符:  "
132          << setw(width) << "词频:  "
133          << setw(width) << "编码:  "
134          << endl;
135
136     queue.push(p);
137     while (!queue.empty())
138     {
139         p = queue.front();
140         queue.pop();
141
142         if (p->left != NULL){
143             queue.push(p->left);
144         }
145         if (p->right != NULL){
146             queue.push(p->right);
147         }

```



```

148
149         //只对叶结点进行输出
150         if ((p->left == NULL) && (p->right == NULL))
151         {
152             //打印
153             p->show();
154             //写入文件
155             file << "char: " << p->data
156                 << setw(width) << "frequency: " << p->weight
157                 << setw(width) << "codes: " << p->codes
158                 << endl;
159         }
160     }
161
162     file.close();
163 }
164
165 void HuffmanTree::encode(const string& filename1, const string&
    filename2)
166 {
167     fstream sourceFile(filename1.data(), ios::in); //打开原文本文件
168     if (sourceFile.bad()){
169         cout << "cann't open " << filename1 << endl;
170         return;
171     }
172     fstream desFile(filename2.data(), ios::out | ios::binary); //创建
        目标二进制文件
173     if (desFile.bad()){
174         cout << "cann't open " << filename2 << endl;
175         return;
176     }
177
178     string codes; //所有编码
179     //每次读入一个字符，将其编码追加加入codes
180     char c1;
181     while (sourceFile.get(c1)) //每次读入一字符
182     {
183         auto iter = codeTable.find(c1);
184         if (iter == codeTable.end()) //c1不在编码表中，报错
185         {
186             cout << "ERROR!!! ";
187             return;

```

```

188         }
189         else//c1在编码表中
190             codes.append(iter->second);
191     }
192
193     // cout << "codes: " << codes << endl;//debug
194
195     //二进制文件前四个字节存放数据的有效位数
196     int bitSize = codes.size();
197     desFile.write((const char*)&bitSize, sizeof(bitSize));
198
199     //第五个字节开始存放数据
200     int byteSize = codes.size()/8 + 1;
201
202     for (int i = 0; i < byteSize; i++)
203     {
204         bitset<8> byte(codes.substr(i*8, 8));//直接用string初始化
                byte
205
206         //将一字节编码写入文件
207         desFile.write((const char*)&byte, byte.size()/8);
208     }
209
210     sourceFile.close();
211     desFile.close();
212 }
213
214 void HuffmanTree::decode(const string& filename1, const string&
    filename2)
215 {
216     fstream sourceFile(filename1.data(), ios::in | ios::binary);//打
        开原二进制文件
217     if (sourceFile.bad()){
218         cout << "cann't open " << filename1 << endl;
219         return;
220     }
221     fstream desFile(filename2.data(), ios::out);//创建目标文本文件
222     if (desFile.bad()){
223         cout << "cann't open " << filename2 << endl;
224         return;
225     }
226

```

```

227 //先读入前四字节
228 int bitSize;
229 sourceFile.read((char*)&bitSize, sizeof(bitSize));
230 int byteSize = bitSize/8 + 1;
231
232 string codes;
233 for (int i = 0; i < byteSize; i++)
234 {
235     //读入一字节
236     bitset<8> byte;
237     sourceFile.read((char*)&byte, byte.size()/8);
238     //将这一字节编码转为string
239     codes.append(byte.to_string());
240 }
241
242 //先将最后八位全部读入，再删去冗余部分
243 int n = byteSize*8 - bitSize;
244 cout << "n: " << n << endl; //debug
245 codes.erase((byteSize-1)*8, n);
246
247 // cout << "codes: " << codes << endl; //debug
248 // return;
249
250 string curCodes;
251 char c;
252 BiNode* p;
253 int i = 0;
254 while (i < codes.size())
255 {
256     curCodes.clear(); //先清空上次的编码
257
258     while (1) //一次循环解码一字符
259     {
260         if (i >= codes.size()) //表明编码结束
261             break;
262
263         curCodes += codes.at(i++);
264         p = root;
265         int pos = 0;
266
267         //寻找curCodes对应的叶结点
268         while ((p != NULL) && (pos < curCodes.size()))

```

```

269         {
270             if (curCodes.at(pos) == '0')
271                 p = p->left;
272             else if (curCodes.at(pos) == '1')
273                 p = p->right;
274             pos++;
275         }
276
277         //若p指向叶结点且pos指向curCodes末尾, 则可解码一个字符
278         if ((p->left == NULL) && (p->right == NULL) &&
279             (pos == curCodes.size()))
280         {
281             break;
282         }
283         //反之, 需要继续读入二进制码
284     }
285
286     desFile << p->data; //成功解码一个字符
287 }
288
289 sourceFile.close();
290 desFile.close();
291 }

```

代码 23: huffmantree.cpp

```

1  #ifndef CHARTABLE_H
2  #define CHARTABLE_H
3
4  #include <string>
5  #include <map>
6
7  using namespace std;
8
9  class CharTable{
10 public:
11     //计算文件中所有出现字符的词频
12     void calculateFreq(const string& filename);
13
14 private:
15     //存放词频
16     map<char, int> table;

```

```

17
18     friend class HuffmanTree;
19 };
20
21 #endif

```

代码 24: chartable.h

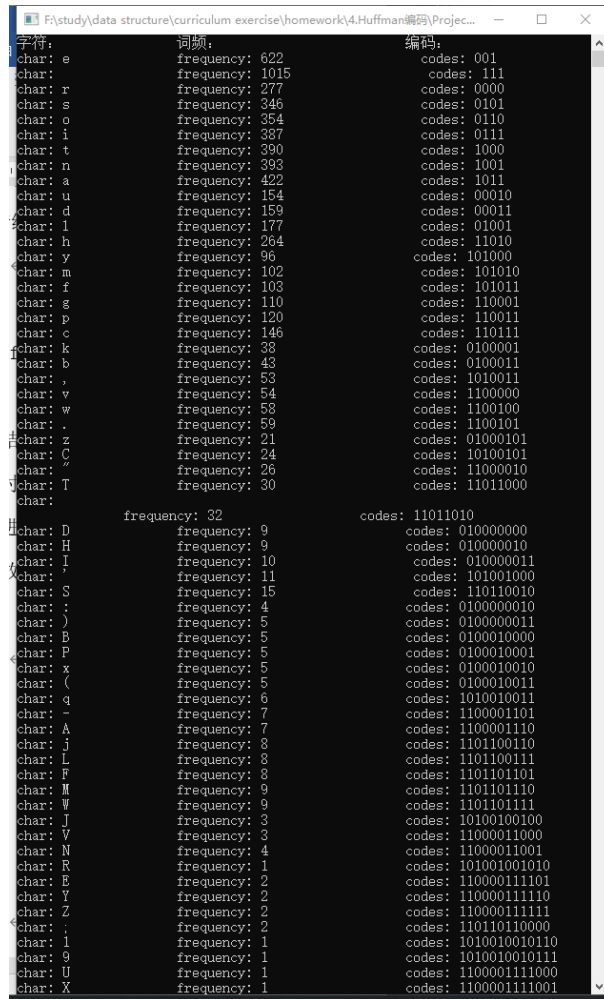
```

1     #include "chartable.h"
2
3     #include <iostream>
4     #include <fstream>
5
6     void CharTable::calculateFreq(const string& filename)
7     {
8         fstream file(filename.data());
9         if (file.bad()){
10             cout << "cann't open " << filename << endl;
11             return;
12         }
13
14         //每次读入一个字符
15         //若不在表中，将其插入表中
16         //若在表中，其词频加1
17         char c1;
18         while (file.get(c1))//每次读入一字符
19         {
20             auto iter = table.find(c1);
21             if (iter == table.end())//c1不在表中
22             {
23                 table.insert(pair<char, int>(c1, 1));
24             }
25             else//c1在表中
26             {
27                 iter->second++;
28             }
29         }
30
31         file.close();
32     }

```

代码 25: chartable.cpp

(四) 测试数据及其结果



字符	词频	编码
char: e	frequency: 622	codes: 001
char: r	frequency: 1015	codes: 111
char: s	frequency: 277	codes: 0000
char: o	frequency: 346	codes: 0101
char: i	frequency: 354	codes: 0110
char: t	frequency: 387	codes: 0111
char: n	frequency: 390	codes: 1000
char: a	frequency: 393	codes: 1001
char: u	frequency: 422	codes: 1011
char: d	frequency: 154	codes: 00010
char: l	frequency: 159	codes: 00011
char: h	frequency: 177	codes: 01001
char: y	frequency: 264	codes: 11010
char: m	frequency: 96	codes: 101000
char: f	frequency: 102	codes: 101010
char: g	frequency: 103	codes: 101011
char: p	frequency: 110	codes: 110001
char: c	frequency: 120	codes: 110011
char: k	frequency: 146	codes: 110111
char: b	frequency: 38	codes: 0100001
char: ,	frequency: 43	codes: 0100011
char: v	frequency: 53	codes: 01010011
char: w	frequency: 54	codes: 11000000
char: .	frequency: 58	codes: 1100100
char: z	frequency: 59	codes: 1100101
char: C	frequency: 21	codes: 01000101
char: "	frequency: 24	codes: 10100101
char: T	frequency: 26	codes: 11000010
char: frequency: 30	codes: 11011000	
char: frequency: 32	codes: 11011010	
char: D	frequency: 9	codes: 010000000
char: H	frequency: 9	codes: 010000010
char: I	frequency: 10	codes: 010000011
char: S	frequency: 11	codes: 101001000
char: ;	frequency: 15	codes: 110110010
char:)	frequency: 4	codes: 010000000
char: B	frequency: 5	codes: 010000001
char: P	frequency: 5	codes: 0100010000
char: x	frequency: 5	codes: 0100010001
char: (frequency: 5	codes: 0100010010
char: q	frequency: 5	codes: 0100010011
char: -	frequency: 6	codes: 1010010011
char: A	frequency: 7	codes: 1100001101
char: j	frequency: 7	codes: 1100001110
char: L	frequency: 8	codes: 1101100111
char: F	frequency: 8	codes: 1101101101
char: M	frequency: 8	codes: 1101101110
char: W	frequency: 9	codes: 1101101111
char: J	frequency: 9	codes: 10100100100
char: V	frequency: 3	codes: 11000011000
char: N	frequency: 3	codes: 11000011001
char: R	frequency: 4	codes: 101001001010
char: E	frequency: 1	codes: 110000111101
char: Y	frequency: 2	codes: 110000111110
char: Z	frequency: 2	codes: 110000111111
char: !	frequency: 2	codes: 110101100000
char: l	frequency: 1	codes: 1010010010110
char: 9	frequency: 1	codes: 1010010010111
char: U	frequency: 1	codes: 1100001111000
char: X	frequency: 1	codes: 1100001111001

图 6: 实验截图

(五) 时间复杂度

$$O(n)$$

(六) 改进方法

采用其他编码方法如 LZW 编码。

十一、家谱管理系统

(一) 数据结构

树

(二) 算法设计思想

将祖先结点作为家族类的成员变量，每一个结点记录一对夫妻的信息，以及存放后代夫妻结点的数组。

(三) 源程序

```
1  #include <iostream>
2  #include "familytree.h"
3
4  /* run this program using the console pauser or add your own getch,
   system("pause") or input loop */
5
6  int main(int argc, char** argv)
7  {
8      const string fileName("data.txt");//原始数据文件
9      // const string fileName("test.txt");
10
11     familyTree family(fileName);
12
13     int x;
14     while(1)
15     {
16         // system("cls");
```

```

17
18         cout << "请输入数字实现相应功能噢。" << endl
19         << "若想显示家谱所有人信息，请按1" << endl
20         << "若想显示第n代所有人的信息，请按2" << endl
21         << "若想显示某人的信息，请按3" << endl
22         << "若想查询两人之间关系，请按4" << endl
23         << "若想为某夫妇添加孩子，请按5" << endl
24         << "若想为某人添加配偶，请按6" << endl
25         << "若想删除某成员，请按7" << endl
26         << "若想修改某成员信息，请按8" << endl
27         << "若想按照出生/死亡日期查询成员信息，请按9" << endl
28         << "若想退出程序，请按0" << endl << endl
29         << "请输入：";
30         cin >> x;
31         cout << endl;
32         Operation(x, family); // 执行相应操作
33
34         system("pause");
35     }
36 }

```

代码 26: main.cpp

```

1     #ifndef FAMILYTREE_H
2     #define FAMILYTREE_H
3
4     #include <iostream>
5     #include <fstream>
6     #include <vector>
7     #include <queue>
8     #include <string>
9     #include <string.h>
10    #include <algorithm>
11    #define NONE -1
12    using namespace std;
13
14    struct Person{
15        string name; // 姓名
16        bool gender; // 性别
17        bool married; // 婚否
18        bool alive; // 健在否
19        string date; // 出生/死亡日期

```



```

20     string address;//住址
21
22     Person();
23     Person(string& nameIn, bool genderIn, bool marriedIn, bool
        aliveIn, string& dateIn, string& addressIn);
24 };
25
26 struct Couple{
27     // bool which;//若真, husband为子, 若假, wife为女
28     Person husband;
29     Person wife;
30     int generation;//第几代人
31
32     vector<Couple> offspring;//后代
33
34     Couple();
35     Couple(Person husbandIn, Person wifeIn, int geIn = NONE);
36 };
37
38 class familyTree{
39     public:
40         familyTree(const string& fileName);//读取数据并建树
41         ~familyTree();//销毁树
42
43         void show();//显示所有人信息
44         void show(int generation);//显示第n代所有人信息
45         void show(const Person& person, bool familyMember = false);
            //显示此人信息
46         void show(bool alive, const string& date);//按出生/死亡日期
            显示信息
47         void showRelation(const string& name1, const string& name2);
            //显示两人关系
48
49         Couple* find(const string& name);//按姓名查找
50         Couple* findParents(const string& name);//返回其双亲
51
52         void addCouple(const string& name, const Person person);//添
            加配偶
53         void addChild(const string& name, Couple couple);//添加孩子
54         void deletePerson(const string& name);//删除成员
55         void modify(const string& name, Person person);//修改成员信
            息

```

```

56
57 // private:
58     Couple ancestor;//根结点
59
60     int personCnt;//总人数
61 };
62
63 void Operation(int x, familyTree& family);
64 void operation0();//退出程序
65 void operation1(familyTree& family);//显示家谱所有人信息
66 void operation2(familyTree& family);//显示第n代所有人的信息
67 void operation3(familyTree& family);//显示某人的信息
68 void operation4(familyTree& family);//查询两人之间关系
69 void operation5(familyTree& family);//为某夫妇添加孩子
70 void operation6(familyTree& family);//为某人添加配偶
71 void operation7(familyTree& family);//删除某成员
72 void operation8(familyTree& family);//修改某成员信息
73 void operation9(familyTree& family);//按照出生/死亡日期查询成员信息
74
75 #endif

```

代码 27: familytree.h

```

1  #include "familytree.h"
2
3  Person::Person(){}
4
5  Person::Person(string& nameIn, bool genderIn, bool marriedIn, bool
    aliveIn, string& dateIn, string& addressIn)
6      :name(nameIn), gender(genderIn), married(marriedIn),
    alive(aliveIn), date(dateIn), address(addressIn)
7  {
8
9  }
10
11 Couple::Couple(){}
12
13 Couple::Couple(Person husbandIn, Person wifeIn, int geIn)
14     : husband(husbandIn), wife(wifeIn), generation(geIn)
15 {
16
17 }

```

```
18
19 familyTree::familyTree(const string& fileName)
20 {
21     personCnt = 0;
22     fstream dataFile(fileName, ios::in);
23
24     string name1, name2, date, address, tmp1, tmp2, tmp3;
25     bool gender, married, alive;
26     char x;
27     while (dataFile.get(x))
28     {
29         if (personCnt != 0)
30             dataFile >> name1;
31
32         dataFile >> name2 >> tmp1 >> tmp2 >> tmp3 >> date >> address
33             ;
34         dataFile.ignore(); // 读掉换行符
35
36         if (tmp1 == "男")            gender = true;
37         else if (tmp1 == "女")        gender = false;
38         if (tmp2 == "已婚")           married = true;
39         else if (tmp2 == "未婚")      married = false;
40         if (tmp3 == "健在")           alive = true;
41         else if (tmp3 == "已故")      alive = false;
42         Person person(name2, gender, married, alive, date, address);
43
44         if (x == '5') // 添加孩子
45         {
46             if (gender)
47                 addChild(name1, Couple(person, Person()));
48             else
49                 addChild(name1, Couple(Person(), person));
50         }
51         else if (x == '6') // 添加配偶
52         {
53             addCouple(name1, person);
54         }
55     }
56     dataFile.close();
57 }
58
```

```

59     familyTree::~familyTree(){}
60
61     //显示所有人信息（层序遍历）
62     void familyTree::show()
63     {
64         cout << "以下是家谱中所有人的信息：" << endl;
65
66         queue<Couple*> Queue;
67         Queue.push(&ancestor);
68
69         while (!Queue.empty())
70         {
71             Couple* cpl = Queue.front();
72             Queue.pop();
73
74             //输出夫妇信息
75             show(cpl->husband);
76             show(cpl->wife);
77
78             for (int i = 0; i < cpl->offspring.size(); i++)
79                 Queue.push(&cpl->offspring.at(i));
80         }
81     }
82
83     //显示第n代所有人信息（层序遍历）
84     void familyTree::show(int generation)
85     {
86         cout << "以下是第 " << generation << " 代所有人的信息：" << endl
87             ;
88
89         queue<Couple*> Queue;
90         Queue.push(&ancestor);
91         // Couple* lastPtr = Queue.front();//指向每层最后一个结点
92         // int level = 1;
93
94         while (!Queue.empty())
95         {
96             Couple* cpl = Queue.front();
97             Queue.pop();
98
99             //输出夫妇信息
100             if (cpl->generation == generation)

```

```

100         {
101             show(cpl->husband);
102             show(cpl->wife);
103         }
104         else if (cpl->generation > generation)
105             return;
106
107         for (int i = 0; i < cpl->offspring.size(); i++)
108             Queue.push(&cpl->offspring.at(i));
109
110         //     if (cpl == lastPtr)
111         //     {
112         //         lastPtr = Queue.back();
113         //         level++;
114         //     }
115     }
116 }
117
118 //显示此人信息
119 void familyTree::show(const Person& person, bool familyMember)
120 {
121     if (person.name.empty())
122         return;
123     // cout << "here4" << endl; //debug
124
125     cout << "以下为 " << person.name << " 的全部信息: " << endl
126          << person.name << " ";
127     if (person.gender)        cout << "男 ";
128     else                      cout << "女 ";
129     if (person.married)       cout << "已婚 ";
130     else                      cout << "未婚 ";
131     if (person.alive)         cout << "健在 ";
132     else                      cout << "已故 ";
133     cout << person.date << " " << person.address << endl << endl;
134
135     if (familyMember)
136     {
137         Couple* cpl = findParents(person.name); //找到其双亲
138         if (cpl != NULL)
139         {
140             cout << "以下为 " << person.name << " 双亲的全部信息: "
141                  << endl << endl;

```

```

141         show(cpl->husband);
142         show(cpl->wife);
143     }
144     else
145         cout << "未找到 " << person.name << " 的双亲。" << endl
            << endl;
146
147     cpl = find(person.name); // 找到其本人
148     if (cpl == NULL)
149         cout << "未找到 " << person.name << ", 请检查输入是否合
            法。" << endl << endl;
150     else if (cpl->offspring.size() == 0)
151         cout << person.name << " 无子女。" << endl << endl;
152     else
153     {
154         cout << "以下为 " << person.name << " 子女的全部信息: "
            << endl << endl;
155         for (int i = 0; i < cpl->offspring.size(); i++)
156         {
157             show(cpl->offspring[i].husband);
158             show(cpl->offspring[i].wife);
159         }
160     }
161 }
162 }
163
164 // 显示两人关系
165 void familyTree::showRelation(const string& name1, const string&
    name2)
166 {
167     if (name1 == name2)
168     {
169         cout << "这是同一人。" << endl;
170         return;
171     }
172
173     Couple* cpl1 = find(name1);
174     Couple* cpl2 = find(name2);
175     if (cpl1 == cpl2)
176     {
177         cout << name1 << "与 " << name2 << " 是夫妻关系。" << endl;
178         return;

```

```
179     }
180
181     int ge1 = cpl1->generation;
182     int ge2 = cpl2->generation;
183     int minG = min(ge1, ge2);
184     Couple* cpl;
185
186     if (ge1 > minG)
187     {
188         string tmp = name1;
189
190         cout << name1 << " ";
191         while (ge1-- > minG)
192         {
193             cout << "的父亲";
194             cpl = findParents(tmp);
195             tmp = cpl->husband.name;
196         }
197
198         if (cpl->husband.name == name2)
199             cout << "是 " << name2 << endl;
200         else if (cpl->wife.name == name2)
201             cout << "的妻子是 " << name2 << endl;
202         else
203             cout << "与 " << name2 << " 是兄弟姐妹关系。" << endl;
204     }
205     else if (ge2 > minG)
206     {
207         string tmp = name2;
208         cout << name2 << " ";
209         while (ge2-- > minG)
210         {
211             cout << "的父亲";
212             cpl = findParents(tmp);
213             tmp = cpl->husband.name;
214         }
215
216         if (cpl->husband.name == name1)
217             cout << "是 " << name1 << endl;
218         else if (cpl->wife.name == name1)
219             cout << "的妻子是 " << name1 << endl;
220         else
```

```

221         cout << "与 " << name1 << " 是兄弟姐妹关系。" << endl;
222     }
223     else
224         cout << name1 << "与 " << name2 << " 是兄弟姐妹关系。" <<
            endl;
225 }
226
227 //按姓名查找 (层序遍历)
228 Couple* familyTree::find(const string& name)
229 {
230     queue<Couple*> Queue;
231     Queue.push(&ancestor);
232
233     while (!Queue.empty())
234     {
235         Couple* cpl = Queue.front();
236         Queue.pop();
237
238         if ((cpl->husband.name == name) || (cpl->wife.name == name))
239             //找到
240             return cpl;
241
242         for (int i = 0; i < cpl->offspring.size(); i++)
243             Queue.push(&cpl->offspring.at(i));
244     }
245     //未找到
246     return NULL;
247 }
248
249 //返回其双亲 (层序遍历)
250 Couple* familyTree::findParents(const string& name)
251 {
252     queue<Couple*> Queue;
253     Queue.push(&ancestor);
254
255     while (!Queue.empty())
256     {
257         Couple* cpl = Queue.front();
258         Queue.pop();
259
260         for (int i = 0; i < cpl->offspring.size(); i++)
261             if ((cpl->offspring[i].husband.name == name) || (cpl->

```



```

        offspring[i].wife.name == name))//找到
261         return cpl;
262
263         for (int i = 0; i < cpl->offspring.size(); i++)
264             Queue.push(&cpl->offspring.at(i));
265     }
266     //未找到
267     return NULL;
268 }
269
270 //按出生/死亡日期查找 (层序遍历)
271 void familyTree::show(bool alive, const string& date)
272 {
273     cout << "以下为符合要求的所有成员信息: " << endl;
274     queue<Couple*> Queue;
275     Queue.push(&ancestor);
276
277     while (!Queue.empty())
278     {
279         Couple* cpl = Queue.front();
280         Queue.pop();
281
282         if ((cpl->husband.alive == alive) && (cpl->husband.date ==
283             date))
284             show(cpl->husband);
285         if ((cpl->wife.alive == alive) && (cpl->wife.date == date))
286             show(cpl->wife);
287
288         for (int i = 0; i < cpl->offspring.size(); i++)
289             Queue.push(&cpl->offspring.at(i));
290     }
291
292     //添加孩子
293     void familyTree::addChild(const string& name, Couple couple)
294     {
295         if (personCnt == 0)
296         {
297             ancestor = couple;
298             ancestor.generation = 1;
299         }
300         else

```

```

301     {
302         Couple* cpl = find(name);
303         if (cpl == NULL)
304         {
305             cout << "未找到 " << name << ", 请检查输入是否合法。" <<
                endl << endl;
306             return;
307         }
308         couple.generation = cpl->generation+1;
309         cpl->offspring.push_back(couple);
310     }
311     personCnt++;
312 }
313
314 //添加配偶
315 void familyTree::addCouple(const string& name, const Person person)
316 {
317     // cout << "here5" << endl; //debug
318     Couple* cpl = find(name);
319     if (cpl->husband.name == name)
320         cpl->wife = person;
321     else
322         cpl->husband = person;
323     personCnt++;
324 }
325
326 //删除成员
327 void familyTree::deletePerson(const string& name)
328 {
329     if ((ancestor.husband.name == name) || (ancestor.wife.name ==
        name))
330         ancestor.~Couple();
331     else
332     {
333         Couple* cpl = findParents(name);
334         if (cpl == NULL)
335         {
336             cout << "未找到 " << name << ", 请检查输入是否合法。" <<
                endl << endl;
337             return;
338         }
339         auto iter = cpl->offspring.begin();

```

```
340         for (; iter != cpl->offspring.end(); )
341             if ((iter->husband.name == name) || (iter->wife.name ==
342                 name))
343             {
344                 iter = cpl->offspring.erase(iter);
345             }
346             else
347                 iter++;
348     }
349 }
350 //修改成员信息
351 void familyTree::modify(const string& name, Person person)
352 {
353     Couple* cpl = find(name);
354     if (cpl->husband.name == name)
355         cpl->husband = person;
356     else if (cpl->wife.name == name)
357         cpl->wife = person;
358 }
359
360 void Operation(int x, familyTree& family)
361 {
362     switch (x)
363     {
364         case 0: operation0();break;
365         case 1: operation1(family);break;
366         case 2: operation2(family);break;
367         case 3: operation3(family);break;
368         case 4: operation4(family);break;
369         case 5: operation5(family);break;
370         case 6: operation6(family);break;
371         case 7: operation7(family);break;
372         case 8: operation8(family);break;
373         case 9: operation9(family);break;
374         default :operation0();break;
375     }
376 }
377
378 void operation0()
379 {
380     cout<< "程序已退出。" << endl;
```

```
381         exit(0);
382     }
383
384     void operation1(familyTree& family)
385     {
386         family.show();
387     }
388
389     void operation2(familyTree& family)
390     {
391         int generation;
392         cout << "您想显示第几代人的信息呢？" << endl;
393         cin >> generation;
394         family.show(generation);
395     }
396
397     void operation3(familyTree& family)
398     {
399         string name;
400         cout << "您想显示谁的信息呢？" << endl;
401         cin >> name;
402         bool familyMember;
403         cout << "您想显示 " << name << " 家庭成员的信息吗？若想输入1，反
           之输入0：" << endl;
404         cin >> familyMember;
405         Couple* cpl = family.find(name);
406         if (cpl == NULL)
407         {
408             cout << "未找到 " << name << "，请检查输入是否合法。" <<
               endl << endl;
409             return;
410         }
411         else if (cpl->husband.name == name)
412             family.show(cpl->husband, familyMember);
413         else
414             family.show(cpl->wife, familyMember);
415     }
416
417     void operation4(familyTree& family)
418     {
419         string name1, name2;
420         cout << "您想查询谁与谁之间的信息呢？" << endl;
```

```

421         cin >> name1 >> name2;
422         family.showRelation(name1, name2);
423     }
424
425     void operation5(familyTree& family)
426     {
427         string name1, name2, date, address, tmp1, tmp2, tmp3;
428         bool gender, married, alive;
429
430         cout << "您想为谁添加孩子呢？" << endl;
431         cin >> name1;
432         cout << "请输入此人所有信息：" << endl;
433         cin >> name2 >> tmp1 >> tmp2 >> tmp3 >> date >> address;
434
435         if (tmp1 == "男")            gender = true;
436         else if (tmp1 == "女")        gender = false;
437         if (tmp2 == "已婚")           married = true;
438         else if (tmp2 == "未婚")      married = false;
439         if (tmp3 == "健在")           alive = true;
440         else if (tmp3 == "已故")      alive = false;
441         Person person(name2, gender, married, alive, date, address);
442
443         if (gender)
444             family.addChild(name1, Couple(person, Person()));
445         else
446             family.addChild(name1, Couple(Person(), person));
447     }
448
449     void operation6(familyTree& family)
450     {
451         string name1, name2, date, address, tmp1, tmp2, tmp3;
452         bool gender, married, alive;
453
454         cout << "您想为谁添加配偶呢？" << endl;
455         cin >> name1;
456         cout << "请输入此人所有信息：" << endl;
457         cin >> name2 >> tmp1 >> tmp2 >> tmp3 >> date >> address;
458
459         if (tmp1 == "男")            gender = true;
460         else if (tmp1 == "女")        gender = false;
461         if (tmp2 == "已婚")           married = true;
462         else if (tmp2 == "未婚")      married = false;

```

```

463         if (tmp3 == "健在")           alive = true;
464         else if (tmp3 == "已故")       alive = false;
465         Person person(name2, gender, married, alive, date, address);
466
467         family.addCouple(name1, person);
468     }
469
470     void operation7(familyTree& family)
471     {
472         string name;
473         cout << "您想删除哪位成员呢?" << endl;
474         cin >> name;
475
476         family.deletePerson(name);
477     }
478
479     void operation8(familyTree& family)
480     {
481         string name1, name2, date, address, tmp1, tmp2, tmp3;
482         bool gender, married, alive;
483
484         cout << "您想修改哪位成员的信息呢?" << endl;
485         cin >> name1;
486         cout << "请输入此人所有信息:" << endl;
487         cin >> name2 >> tmp1 >> tmp2 >> tmp3 >> date >> address;
488
489         if (tmp1 == "男")               gender = true;
490         else if (tmp1 == "女")           gender = false;
491         if (tmp2 == "已婚")               married = true;
492         else if (tmp2 == "未婚")          married = false;
493         if (tmp3 == "健在")               alive = true;
494         else if (tmp3 == "已故")          alive = false;
495         Person person(name2, gender, married, alive, date, address);
496
497         family.modify(name1, person);
498     }
499
500     void operation9(familyTree& family)
501     {
502         bool alive;
503         cout << "此人是否健在, 若是输入1, 反之输入0:" << endl;
504         cin >> alive;

```

```

505     string date;
506     cout << "请输入出生/死亡日期：" << endl;
507     cin >> date;
508
509     family.show(alive, date);
510 }

```

代码 28: familytree.cpp

(四) 测试数据及其结果

```

1  请输入数字实现相应功能噢。
2  若想显示家谱所有人信息，请按1
3  若想显示第n代所有人的信息，请按2
4  若想显示某人的信息，请按3
5  若想查询两人之间关系，请按4
6  若想为某夫妇添加孩子，请按5
7  若想为某人添加配偶，请按6
8  若想删除某成员，请按7
9  若想修改某成员信息，请按8
10 若想按照出生/死亡日期查询成员信息，请按9
11 若想退出程序，请按0
12
13 请输入：2
14
15 您想显示第几代人的信息呢？
16 3
17 以下是第3代所有人的信息：
18 以下为 颜俊哲 的全部信息：
19 颜俊哲 男 已婚 健在 1976 江苏省姜堰市
20
21 以下为 申润青 的全部信息：
22 申润青 女 已婚 健在 1978 江苏省姜堰市
23
24 以下为 颜俊平 的全部信息：
25 颜俊平 男 已婚 健在 1979 北京市
26
27 以下为 朱榕 的全部信息：
28 朱榕 女 已婚 健在 1981 北京市
29
30 以下为 颜俊臣 的全部信息：
31 颜俊臣 男 已婚 健在 1982 江苏省兴化市

```

32
33 以下为 吴晓艳 的全部信息：
34 吴晓艳 女 已婚 健在 1984 江苏省兴化市
35
36 以下为 范宏奇 的全部信息：
37 范宏奇 男 已婚 健在 1982 天津市
38
39 以下为 颜俊卿 的全部信息：
40 颜俊卿 女 已婚 健在 1984 天津市
41
42 以下为 张[图] 的全部信息：
43 张[图] 男 已婚 健在 1985 天津市
44
45 以下为 赵玉冰 的全部信息：
46 赵玉冰 女 已婚 健在 1986 天津市
47
48 以下为 丁建华 的全部信息：
49 丁建华 男 已婚 健在 1982 江苏省兴化市
50
51 以下为 颜俊嫣 的全部信息：
52 颜俊嫣 女 已婚 健在 1985 江苏省兴化市
53
54 以下为 颜俊浩 的全部信息：
55 颜俊浩 男 已婚 健在 1986 北京市
56
57 以下为 孙文静 的全部信息：
58 孙文静 女 已婚 健在 1987 北京市
59
60 以下为 颜[图]圻 的全部信息：
61 颜[图]圻 女 未婚 健在 1996 北京市
62
63 以下为 郭恩义 的全部信息：
64 郭恩义 男 未婚 健在 1999 福建省厦门市
65
66 以下为 颜宇明 的全部信息：
67 颜宇明 男 未婚 健在 2001 江苏省南京市
68
69 以下为 颜俊曜 的全部信息：
70 颜俊曜 男 未婚 健在 2017 江苏省泰州市

(五) 时间复杂度

$$O(\log n)$$

(六) 改进方法

可在界面设计上加以优化，力争用图形方式显示家族树。另外由于中国的亲缘关系错综复杂，在查询两个人关系的算法方面也有很大的改进空间。

十二、总结

(一) 代码行数

题目	行数
区块链	283
迷宫问题	388
JSON 查找	369
公交线路提示	253
Hash 表应用	
排序算法比较	253
地铁修建	60
社交网络图中结点的“重要性”计算	103
平衡二叉树操作的演示	413
Huffman 编码与解码	417
家谱管理系统	621

1 总代码行数为：3160.

(二) 心得体会

总体来说，本次课设难度较大，任务量也很大，但完成之后也收获满满。

我总共完成了六个个必做题，五个选做题，这些选做题分别为 7 题、10 题、18 题、19 题、20 题，选做题分值总计 12 分，并且充分完成了每个题目及其要求的所有功能。

开始时，感觉必做题难度较大，因为每个题都会或多或少涉及一些课外的操作或技巧，比如第一题中如何获取系统进程、第四题中二进制文件按位读写、第五题中并查集的实现、第七题中平衡二叉排序树的删除操作等等，但通过与同学交流，探讨方法，以及在 CSDN 上参考其他博主的方法，这些问题最终都一一解决。

这次课设充分巩固了我在理论课上的所学知识，并得以实践。相比上机题，课设的很多题都更具有有一定的实际意义，比如 Huffman 编码这道题，若真正实现将其按位存储，能够将文件压缩存储，又比如公交线路提示，利用真实南京公交线路图建立图的存储结构，并给出提示，充分说明了图结构的广泛应用价值。

这次课设也拓宽了我的知识面，学习到了很多实用的操作与技巧，在解决一个个问题的过程中也极大地锻炼了我的自学能力与知识迁移能力。

同时做题过程中当然免不了会有各种奇奇怪怪的 bug，在给自己以及给同学调代码的过程中，自己的 debug 能力也有了进一步的提升。

总之这次课设虽然过程曲折艰辛，但完成之后还是很有成就感的。