**计算机科学与工程学院实验报告**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **实验课程名称** | | **数据结构实验** | | | **实验总成绩** |  |
| **专业** | **计算机科学与技术** | | **班级** | **计算机1907** | **指导教师签字** |  |
| **学号** | **20194701** | | **姓名** | **白千一** | **实验报告批改时间** |  |
| **实验报告分项成绩**   |  |  |  | | --- | --- | --- | | **序号** | **实验项目** | **成绩** | | **1** | **约瑟夫环问题** |  | | **2** | **停车场管理** |  | | **3** | **哈夫曼编码** |  | | **4** | **哈希表应用** |  | | **5** | **STL的栈stack类** |  | | | | | | | |
| **实验课程总结**  从以下方面总结：1.实验体现知识应用和初步研究能力；2. 反映基本观察、发现问题和分析问题能力；3. 实验项目内容或者实验课程是否存在问题及下一年度改进意见）4.其他方面   1. 通过实验，加强了对数据结构知识的理解，提高了数据结构知识应用的能力，初步掌握了将课堂知识转化为实际应用项目的技能。对实验给出的题目要求，每次都要认真理解题目含义，仔细研究需要用到数据结构，以及数据结构的实现过程。 2. 实验有助于观察问题，发现问题，解决问题的能力的提高。理解题目的过程中需要将题目需求转化程序运行目标，认真分析思考如何利用数据结构实现；另外在debug的过程中，需要准确定位bug所在语句，并认真观察，推理运行流程，并从中找到错误所在。 3. 实验内容难易不均。 4. 在实验的过程中，经常因为小小的问题，而导致实验的难以继续，当解决了之后，发现是因为基础数据结构知识的掌握不牢，所以在以后的学习和实验中，要更加注重知识的准确性，编程熟练度的提高，而不是只注重结果的完成。 | | | | | | |

**实验一**

约瑟夫环问题

问题定义及需求分析

Josephus排列问题定义如下：假设n个竞赛者排成一个环形。给定一个正整数m≤n，从第1人开始，沿环计数，第m人出列。这个过程一直进行到所有人都出列为止。最后出列者为优胜者。全部出列次序定义了1，2，…n的一个排列。称为（n，m）Josephus排列。例如，（7，3）Josephus排列为3,6,2,7,5,1,4。

设计求解Josephus排列问题程序。

（1）采用顺序表、单链表或双向循环链表等数据结构。

（2）采用双向循环链表实现Josephus排列问题，且奇数次顺时针轮转，偶数次逆时针轮转。

（3）推荐采用静态链表实现Josephus排列问题。

（4）根据程序提示要求输入两个整数，分别代表参加游戏的人数和每次淘汰的计数序号。输出Josephus排列。

概要设计

1. 用静态双向链表表示人员组成的环形队列。设计两个静态链表表示未淘汰人员和淘汰人员。通过人员所在的队列表示人员的状态。利用未淘汰人员链表的人员数量表示游戏状态（继续进行或结束）。
2. 程序运行后，首先根据人员数量创造静态双向链表。默认人员全部存在。默认方向逆时针。
3. 游戏开始程序运行，遍历未淘汰人员链表，遇到基数序号就将人员淘汰，即不断调用删除人员函数，不断把淘汰的人员加到表示淘汰人员的链表里。直到表示未淘汰人员的链表为空。其中每次删除人员之后，都会改变遍历链表的方向，表示改变顺（逆）的改变。
4. 最后输出淘汰人员链表，表示Josephus排列

详细设计

1. 人员结点（Jose）组成：自身在静态链表中位置（id），以及前驱（pre）后继（next）结点。
2. 约瑟夫环（Josephus）组成：总人员数量（m），淘汰序号（n），未淘汰人员（in）和淘汰人员（out）的链表头指针，人员结点组成的静态链表（Jose[]）。
3. 静态链表初始化，约瑟夫环初始化。
4. 删除人员：根据删除人员的下标（index）在未淘汰人员的链表中删除此节点，在淘汰人员的链表中插入此节点。最后返回删除结点的前驱。采用头插法插入。

if (index<1 || index>Jo.n) return -1;

Jose \*temp1 = &Jo.Jose\_array[index];

int mn = temp1->next;

删除(temp,Jo.in);

Jose \*temp2 = &Jo.Jose\_array[Jo.out];

插入（temp2,Jo.out）;

return mn;

1. 游戏开始程序：建立计数器（num），建立方向标志（direct），后从约瑟夫环（Jo）的未淘汰人员链表开始遍历，根据方向标志决定遍历防线，计数器和淘汰序号相等就进行删除操作，根据方向的变化，进行不同的操作。知道未淘汰人员链表为空。

int 计数器 = 1;

int 方向 = 顺时针;

int m = Jo.m;

int index = Jo.in;

while (Jo.in不为0) {

while (方向为顺时针) {

if (计数器 == m) {

在Jo.in中删除 index;

改变方向;

计数器归一;

break;

}

++num;

index变为后继结点;

}

if (Jo.in为0) break;

while (方向为逆时针) {

if (计数器 == m) {

同顺时针过程;

}

++num;

index变为前驱结点;

}

}

1. 遍历（打印）操作，根据静态链表头指针，逆序遍历链表。

void print(Josephus\* jo)

{

int t = 链表头;

while (t) {

打印t;

t = t前驱;

}

}

调试分析

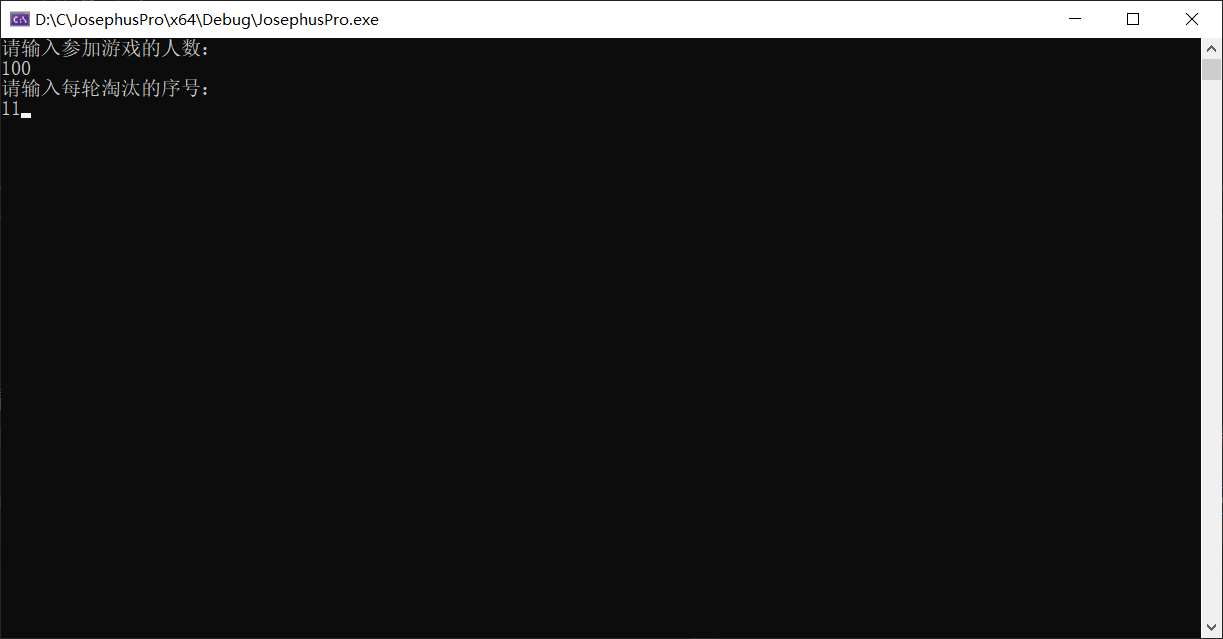
1. 调试过程中对于总人数过少的测试数据会出现程序终止运行的情况，后来发现是在初始化过程中，由于静态链表从下标1开始表示人员，人数过少时，不会进入初始化的循环过程。后采取第一个元素手动初始化的方法解决了问题。
2. 空间复杂度分析：程序运行过程中，会根据人员总数开辟相应空间的静态链表来进行模拟淘汰过程。所以空间复杂度为O(n)。
3. 时间复杂度分析：开始全部人员结点都在静态链表中，不断遍历链表，直到链表为空。计数序号到达m时，淘汰一人，一共淘汰n人，所以遍历的总次数为m\*n次，时间复杂度为O(mn)。
4. 改进设想：可以增加页面设计，提高美感。

使用说明

程序开始运行后，根据提示输入对应的数据，后程序开始计算，给出Josephus排列。

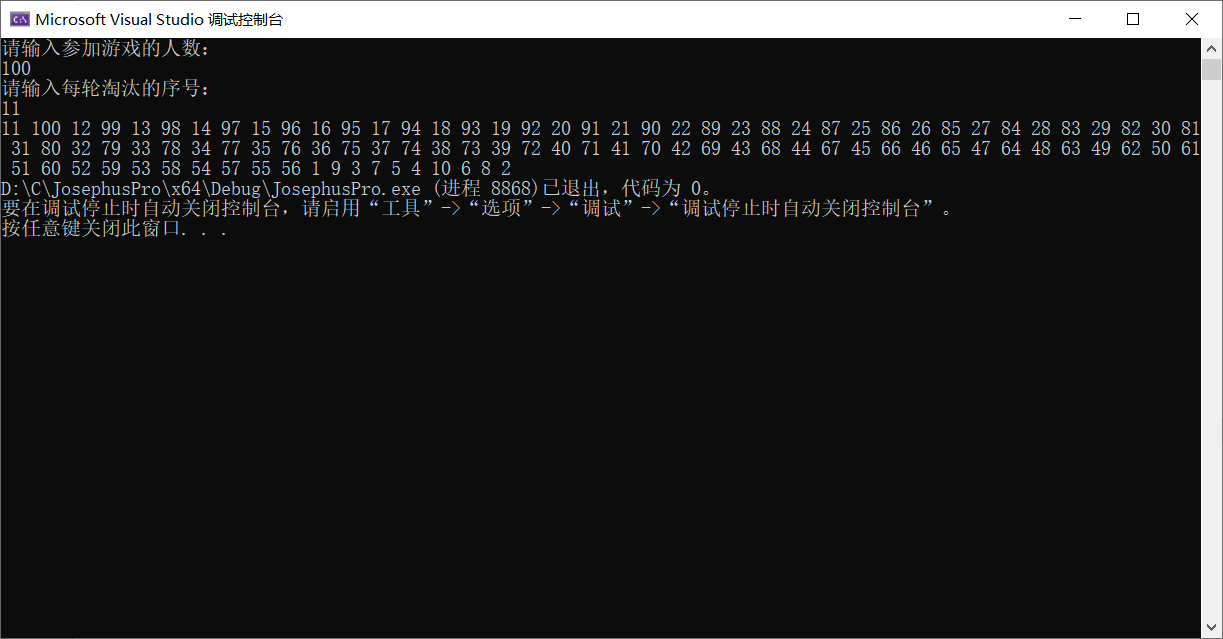
测试结果

1. 输入数据：100 11



1. 输出结果：

11 100 12 99 13 98 14 97 15 96 16 95 17 94 18 93 19 92 20 91 21 90 22 89 23 88 24 87 25 86 26 85 27 84 28 83 29 82 30 81 31 80 32 79 33 78 34 77 35 76 36 75 37 74 38 73 39 72 40 71 41 70 42 69 43 68 44 67 45 66 46 65 47 64 48 63 49 62 50 61 51 60 52 59 53 58 54 57 55 56 1 9 3 7 5 4 10 6 8 2



附录

1. /\*\*\*\*\*\*\*\*struct.h\*\*\*\*\*\*/
3. #pragma once
5. //约瑟夫环的每个结点
6. **typedef** **struct** Jose{
7. **int** id;
8. **int** pre;
9. **int** next;
10. }Jose;
12. //约瑟夫环的规模（总结点数）：n
13. //数到第几个被淘汰：m
14. //仍然剩余的人员所组成的静态链表的头结点：in
15. //已经被淘汰的人员所组成的静态链表的头结点（即为 Josephus排列）：out
16. **typedef** **struct** Josephus{
17. **int** n,m;
18. **int** in,out;
19. Jose\* Jose\_array;
20. }Josephus;
21. /\*\*\*\*\*\*\*\*\*\*\*delete.cpp\*\*\*\*\*\*\*\*\*\*\*\*/
23. #include "struct.h"
24. #include <stdlib.h>
26. **int** delete\_people(Josephus &Jo,**int** index)
27. {
28. **if** (index<1 || index>Jo.n) **return** -1;
29. Jose \*temp1=&Jo.Jose\_array[index];
30. **int** mn = temp1->next;
31. Jo.Jose\_array[temp1->pre].next = temp1->next;
32. Jo.Jose\_array[temp1->next].pre = temp1->pre;
33. **if** (Jo.in == index)Jo.in = mn;
34. **if** (Jo.in == index) Jo.in = 0;
35. Jose \*temp2 = &Jo.Jose\_array[Jo.out];
36. temp1->pre = temp2->pre;
37. temp2->pre = temp1->id;
38. temp1->next = temp2->id;
39. Jo.Jose\_array[temp1->pre].next = temp1->id;
40. Jo.out = temp1->id;
41. **return** mn;
42. }
44. **void** start\_up(Josephus &Jo)
45. {
46. **int** num = 1;
47. **int** direct = 1;
48. **int** m = Jo.m;
49. **int** index = Jo.in;
50. **while** (Jo.in) {
51. **while** (direct) {
52. **if** (num == m) {
53. index = delete\_people(Jo, index);
54. **if** (index == -1) exit(-1);
55. **else** index = Jo.Jose\_array[index].pre;
56. direct = 0;
57. num = 1;
58. **break**;
59. }
60. ++num;
61. index = Jo.Jose\_array[index].next;
62. }
63. **if** (!Jo.in) **break**;
64. **while** (!direct) {
65. **if** (num == m) {
66. index = delete\_people(Jo, index);
67. direct = 1;
68. num = 1;
69. **break**;
70. }
71. ++num;
72. index = Jo.Jose\_array[index].pre;
73. }
74. }
75. }
76. /\*\*\*\*\*\*\*\*\*initJose.cpp\*\*\*\*\*\*\*\*\*\*\*/
78. #include "struct.h"
79. #include <stdio.h>
80. #include <string.h>
81. #include <stdlib.h>
83. //约瑟夫环Jose[]数组的初始化
84. Jose\* initJose(**int** n)
85. {
86. **if** (n < 1)**return** 0;
87. Jose\* head = (Jose\*)malloc(**sizeof**(Jose)\*(**static\_cast**<**\_\_int64**>(n) +1));
88. **if** (!head) **return** 0;
89. head[0].pre = 0; head[0].id = 0; head[0].next = 0;
90. head[1].pre = n; head[1].id = 1; head[1].next = 2;
91. **for** (**int** i = 2; i < n; ++i) {
92. head[i].pre = i - 1; head[i].id = i; head[i].next = i + 1;
93. }
94. head[n].pre = n - 1; head[n].id = n; head[n].next = 1;
95. **return** head;
96. }
98. Josephus\* initJosephus(**int** n, **int** m)
99. {
100. Josephus\* jo = (Josephus\*)malloc(**sizeof**(Josephus));
101. **if** (!jo) exit(1);
102. jo->in = 1;
103. jo->out = 0;
104. jo->n = n;
105. jo->m = m;
106. jo->Jose\_array = initJose(n);
107. **return** jo;
108. }
109. /\*\*\*\*\*\*\*\*\*\*\*main.cpp\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
111. #include "struct.h"
112. #include <stdio.h>
114. Josephus\* initJosephus(**int**, **int**);
115. **void** start\_up(Josephus&);
117. **void** print(Josephus\* jo)
118. {
119. //int vnt = 1;
120. **int** t = jo->Jose\_array[0].pre;
121. **while** (t) {
122. //printf("%d:%d\n", vnt++, t);
123. printf("%d ", t);
124. t = jo->Jose\_array[t].pre;
125. }
126. }
127. **int** main()
128. {
129. printf("请输入参加游戏的人数：\n");
130. **int** n;
131. scanf\_s("%d", &n);
132. printf("请输入每轮淘汰的序号：\n");
133. **int** m;
134. scanf\_s("%d", &m);
135. Josephus\* game = initJosephus(n, m);
136. start\_up(\*game);
137. print(game);
138. **return** 0;
139. }

**实验二**

停车场管理

问题定义及需求分析

设停车场采用南北方向的双口，每个口都有一个入口和出口。另外停车场入口处各有一个单车道的等候通道，并允许等候的车辆因急事从等候通道直接开走。

设计停车场模拟管理程序。

（1）采用栈或队列等数据结构。

（2）等候车辆的管理。

（3）停车位的管理。

（4）模拟大量车辆涌入的情况。

概要设计

1. 定义数据类型
2. 定义车辆（Car）。数据的最小单元，表示车辆的属性。
3. 定义结点（Node）。由车辆和其前驱后继结点构成，停车场和等候区的元素。
4. 定义停车场（ParkQueue）。由南北两个停车场构成，每个停车场都是一个队列，存储车辆结点。
5. 定义等候区（WaitList）。由车辆结点构成的队列。
6. 主程序流程
7. 程序开始运行，开始自动进入大量车辆，根据车辆出现顺序给车辆编号。首先调用等候区判满函数（isFull\_WaitList），等候区不满，调用车辆进入等候区函数（enter\_WaitList），车辆进入等候区。后调用停车场判满函数（isFull），停车场不满，车辆进入停车场。停车场满后，车辆只进入等候区不进入停车场。等候区满后，等候区不再接收车辆。
8. 调用车辆离开函数（isFull\_WaitList），模拟等候区车辆突然离开的情况。
9. 调用停车场车辆离开函数（PQpop），模拟停车场车辆离开的情况。后程序自动调用等候区出车函数（From\_Wait\_To\_Park），使对应相同方向等候区的车辆进入停车场。

详细设计

1. 数据结构：用双向链表实现队列先进先出的功能。分别用两个队列模拟南北停车场，南北停车场双队列封装在总停车场里。用一个队列加一个标识符模拟南北等候区。由“车辆突然离开”要求，增加此队列中间元素可先出的特殊情况。
2. 定义数据类型：
3. 车辆（Car）：包含车辆ID，以及车辆所在停车场。
4. 结点（Node）：包含车辆（Car），以及前驱结点（pre），后继结点（next）。
5. 停车场（ParkQueue）：包含停车场最大容量（limitNum），停车场先停有车辆数（nowNum），南停车场的队列头（SouthHead）和队列尾（SouthTail），北停车场的队列头（NorthHead）和队列尾（NorthTail）。
6. 等候区（WaitList）：包含等候区最大容量（Length），等候区现有车辆数目（Car\_Number），南北标识符（Location），等候区队列头（head），队列尾（rear）。
7. 结点初始化，停车场初始化，等候区初始化。
8. 停车场判满，等候区判满。
9. 车辆进入等候区：

enter\_WaitList：

if(等候区为满) 返回；

if(是北等候区)｛

北等候区车辆数量++；

将顺序编号附给车辆ID；

头插法插入表示北等候区的队列（链表）。

｝

else

车辆进入南等候区，操作同北等候区；

1. 车辆离开函数：模拟车辆的突然离开

leave\_Suddenly：

输入离开车辆ID；

遍历表示等候区的链表，找到被删除车辆的位置；

在链表中删除目标车辆结点；

等候区车辆数量减一；

1. 车辆从等候区进入停车场：

From\_Wait\_To\_Park：

if(停车场满) 返回；

if(等候区为空) 出错；

if(目标停车场为南停车场)

在表示南等候区的链表尾删除此车辆；

南等候区车辆数量减一；

调用停车场南口进入函数；

else

在表示北等候区的链表尾删除此车辆；

北等候区车辆数量减一；

调用提车场北口进入函数；

1. 停车场车辆进入函数：

PQpush：

if(停车场为空) 返回；

判断目标停车场，确定车辆插入目标队列尾；

尾插法插入目标队列；

停车场车辆数量++；

1. 停车场车辆移出函数：

PQpop：

判断出车停车场，确定删除车辆目标队列头；

队列头部删除目标车辆；

停车场车辆数量减一；

返回被删除车辆；

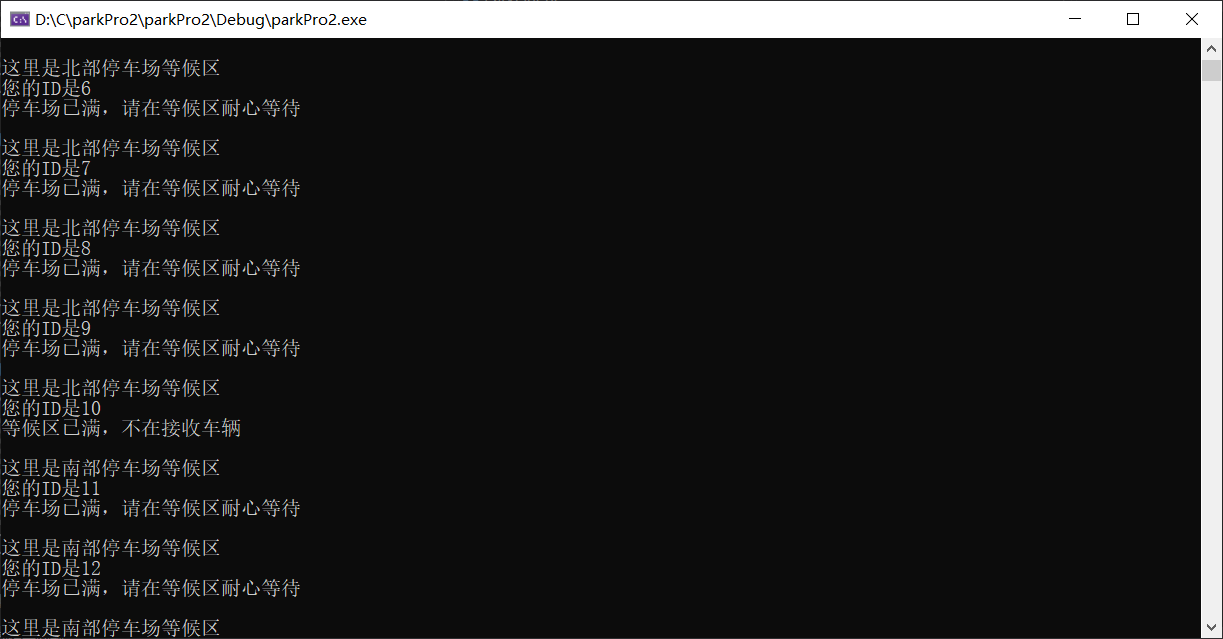
1. 等候区，停车场车辆打印。

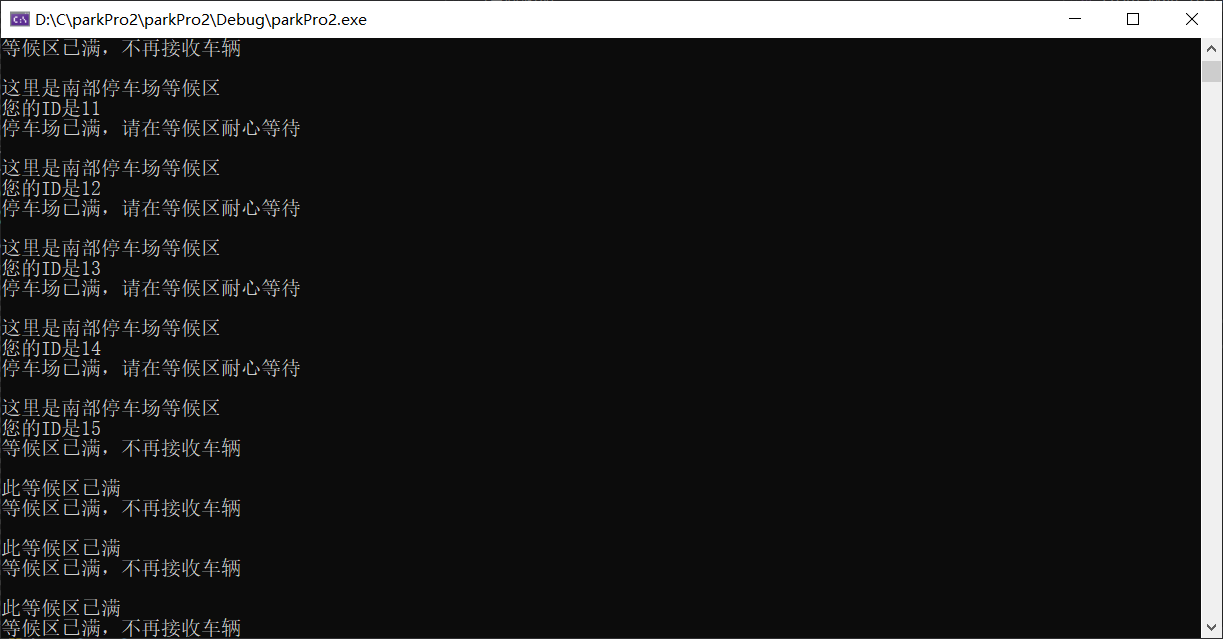
调试分析

* 1. 时间复杂度分析：车辆进入离开等候区，车辆进入离开停车场，都是队列的插入和删除，由于头尾指针的引进，时间复杂度为O(1)。但是在车辆离开函数中，需要遍历表示等候区的队列，时间复杂度为O(n)。所以总的时间复杂度为为O(n)。
  2. 空间复杂度分析：对于表示车辆的数据，需要在等候区或者停车场开辟空间，不需要同时开辟，所以空间复杂度为O(n)。
  3. 改进设想：增加界面设计和动态图，使过程更加直观。

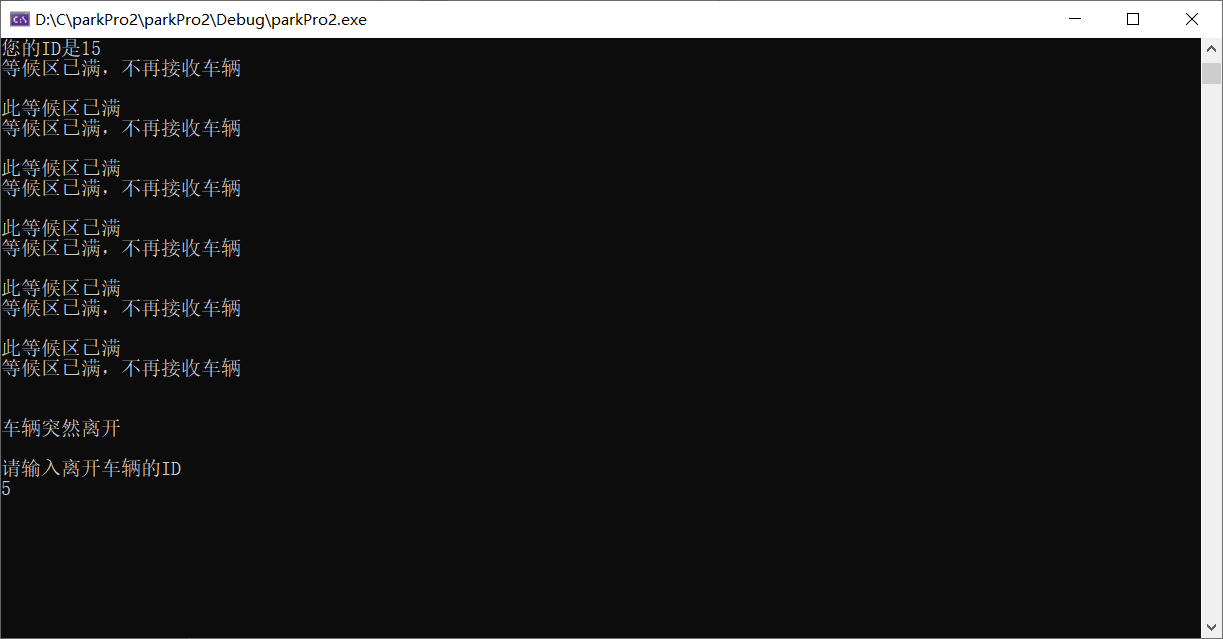
测试结果

1. 开始自动进入20辆车

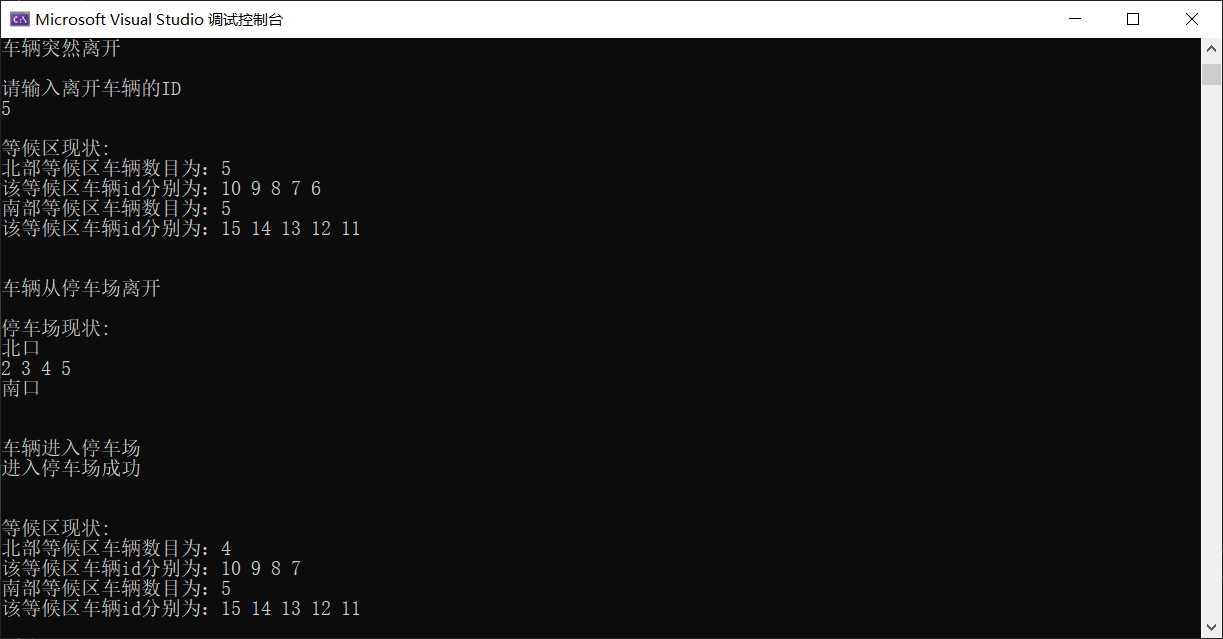


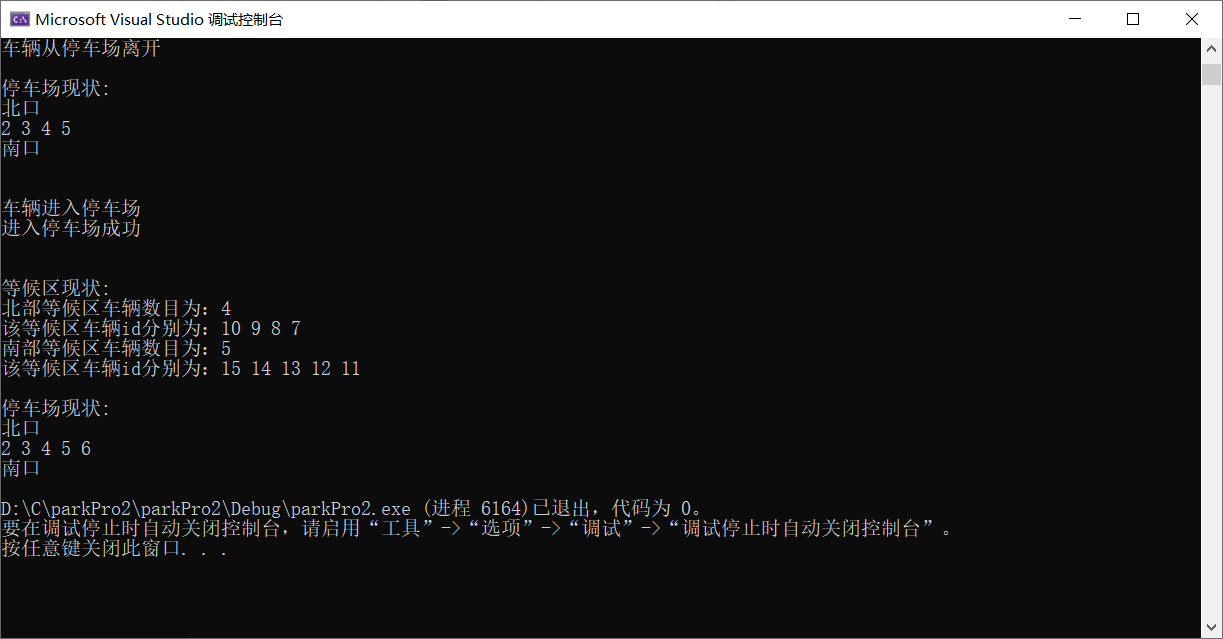


1. 模拟车辆突然离开情况



1. 车辆离开，等候区车辆进入





附录

1. /\*\*\*\*\*\*\*\*struct.h\*\*\*\*\*\*\*\*/
3. #ifndef UNTITLED17\_STRUCT\_H
4. #define UNTITLED17\_STRUCT\_H
6. **static** **int** order = 1; //用来赋给每个等候车辆ID
8. **typedef** **struct** Car {
9. **int** ID;
10. **int** Pos;
11. }Car;
13. **typedef** **struct** Node {
14. Car ca;
15. **struct** Node\* next;
16. **struct** Node\* pre;
17. }Node;
19. **typedef** **struct** WaitList {
20. Node\* head;
21. Node\* rear;
22. **int** Length;
23. **int** Car\_Number;   //等候的车辆数量
24. **int** Location;
25. }WaitList;
27. **typedef** **struct** ParkQueue
28. {
29. **int** limitNum;
30. **int** nowNum;
31. Node\* SouthHead, \* SouthTail;
32. Node\* NorthHead, \* NorthTail;
33. }ParkQueue;
35. #endif //UNTITLED17\_STRUCT\_H
36. /\*\*\*\*\*\*\*\*main.cpp\*\*\*\*\*\*\*\*\*/
38. #include <stdio.h>
39. #include <math.h>
40. #include <malloc.h>
41. #include "struct.h"
42. #define MaxSize\_WaitList 5
44. Node\* initNode();
45. **void** Init\_WaitList(WaitList\*\* W, **int** Loc);
46. **int** isEmpty\_WaitList(WaitList\* W);
47. **int** isFull\_WaitList(WaitList\* W);
48. **int** enter\_WaitList(WaitList\*\* W);
49. **int** leave\_Suddenly(WaitList\*\* W);
50. **void** print\_WaitList(WaitList\*\* W);
51. ParkQueue\* initPQ();
52. **int** isFull(ParkQueue\* PQ);
53. **int** PQpush(ParkQueue\* PQ, Car car, **int** num\_of\_exit);
54. **int** PQpop(ParkQueue\* PQ, Car &car, **int** num\_of\_exit);
55. **void** print(ParkQueue\* pq);
56. Car From\_Wait\_To\_Park(WaitList\*\* W, ParkQueue\* PQ);
58. ///////////////////////////////////////////////////////////////////////////
59. **int** main() {
60. //初始化
61. WaitList\* Wait\_South, \* Wait\_North;
62. Init\_WaitList(&Wait\_South, 0);
63. Init\_WaitList(&Wait\_North, 1);
64. ParkQueue\* Park\_Queue = initPQ();
66. printf("\n车辆进入\n\n");
67. **int** i = 0;
68. **for** (i = 0; i < 20; ++i) {
69. **if** (!isFull\_WaitList(Wait\_North)) {
70. enter\_WaitList(&Wait\_North);
71. //if(!isFull\_WaitList(Wait\_North))
72. From\_Wait\_To\_Park(&Wait\_North, Park\_Queue);
73. }
74. **else** {
75. enter\_WaitList(&Wait\_South);
76. //if (!isFull\_WaitList(Wait\_South))
77. From\_Wait\_To\_Park(&Wait\_South, Park\_Queue);
78. }
79. }
80. printf("\n车辆突然离开\n\n");
81. leave\_Suddenly(&Wait\_North);
82. //leave\_Suddenly(&Wait\_South);
83. printf("\n等候区现状:\n");
84. print\_WaitList(&Wait\_North);
85. print\_WaitList(&Wait\_South);
87. //车辆从停车场离开
88. Car car;
89. printf("\n\n车辆从停车场离开\n");
90. PQpop(Park\_Queue, car, 1);
91. printf("\n停车场现状:");
92. print(Park\_Queue);
93. putchar('\n');
95. //车辆从等候区进入停车区
96. printf("\n车辆进入停车场\n");
97. From\_Wait\_To\_Park(&Wait\_North, Park\_Queue);
98. printf("\n等候区现状:\n");
99. print\_WaitList(&Wait\_North);
100. print\_WaitList(&Wait\_South);
101. printf("\n停车场现状:");
102. print(Park\_Queue);
104. **return** 0;
105. }
106. /\*\*\*\*\*\*\*\*\*\*fuction.cpp\*\*\*\*\*\*\*\*\*\*\*\*/
108. #include <stdio.h>
109. #include <math.h>
110. #include <malloc.h>
111. #include "struct.h"
112. #include <stdlib.h>
114. #define MaxSize\_WaitList 5
116. Node\* initNode()
117. {
118. Node\* newNode = (Node\*)malloc(**sizeof**(Node));
119. **if** (!newNode) exit(0);
120. newNode->ca.ID = 0;
121. newNode->ca.Pos = 0;
122. newNode->next = NULL;
123. newNode->pre = NULL;
124. **return** newNode;
125. }
127. **void** Init\_WaitList(WaitList\*\* W, **int** Loc)
128. {
129. (\*W) = (WaitList\*)malloc(**sizeof**(WaitList));
130. **if** (!(\*W)) exit(0);
131. (\*W)->head = (Node\*)malloc(**sizeof**(Node));
132. **if** (!((\*W)->head)) exit(0);
133. (\*W)->head->next = (\*W)->head->pre = NULL;
134. (\*W)->rear = (\*W)->head;
135. (\*W)->Length = MaxSize\_WaitList;
136. (\*W)->Car\_Number = 0;
137. (\*W)->Location = Loc;
138. }
140. **int** isEmpty\_WaitList(WaitList\* W)
141. {
142. **if** (W->Car\_Number == 0)
143. **return** 1;
144. **else** **return** 0;
145. }

148. **int** isFull\_WaitList(WaitList\* W)
149. {
150. **if** (W->Car\_Number == MaxSize\_WaitList)
151. **return** 1;
152. **else** **return** 0;
153. }
155. **int** enter\_WaitList(WaitList\*\* W)
156. {
158. **if** (!isFull\_WaitList(\*W)) {
159. **if** ((\*W)->Location == 0) {
160. printf("这里是南部停车场等候区\n");
161. }
162. **if** ((\*W)->Location == 1) {
163. printf("这里是北部停车场等候区\n");
164. }
165. }
166. **else** { printf("此等候区已满\n"); **return** 0; }
168. Node\* p = initNode();
169. **if** ((\*W)->rear == (\*W)->head) {
171. (\*W)->Car\_Number++; //等候车数加一
172. p->ca.ID = order++;  //赋给车辆ID
173. printf("您的ID是%d\n", p->ca.ID);
175. (\*W)->rear = p;
176. (\*W)->head->next = (\*W)->rear;
177. (\*W)->head->pre = NULL;
178. (\*W)->rear->pre = (\*W)->head;
179. (\*W)->rear->next = NULL;
180. }
181. **else** {
182. (\*W)->Car\_Number++;
183. p->ca.ID = order++;
184. printf("您的ID是%d\n", p->ca.ID);
186. p->next = (\*W)->head->next;
187. p->pre = (\*W)->head;
188. (\*W)->head->next->pre = p;
189. (\*W)->head->next = p;
190. }
192. **return** 1;
193. }
195. **int** leave\_Suddenly(WaitList\*\* W)
196. {
197. **int** id;
198. printf("请输入离开车辆的ID\n");
199. scanf\_s("%d", &id);
201. Node\* p = (\*W)->head->next;
202. **while** (p != NULL) {
203. **if** (p->ca.ID == id) **break**;
204. **else** p = p->next;
205. }
206. **if** (p != NULL) {
207. printf("%d号车离开等候区\n", id);
208. (\*W)->Car\_Number--;
209. p->pre->next = p->next;
210. p->next->pre = p->pre;
211. p->next = NULL;
212. p->pre = NULL;
213. free(p);
214. **return** 1;
215. }
216. **return** 0;
217. }
219. **void** print\_WaitList(WaitList\*\* W) {
220. **if** ((\*W)->Location == 0) {
221. printf("南部等候区车辆数目为：%d\n", (\*W)->Car\_Number);
222. }
223. **else** printf("北部等候区车辆数目为：%d\n", (\*W)->Car\_Number);
224. Node\* p = (\*W)->head->next;
225. **if** ((\*W)->Car\_Number > 0) printf("该等候区车辆id分别为：");
226. **while** (p != NULL) {
227. printf("%d ", p->ca.ID);
228. p = p->next;
229. }
230. printf("\n");
231. }
232. //////////////////////////////////////////////////////////////
233. ParkQueue\* initPQ() {
234. ParkQueue\* newPQ = (ParkQueue\*)malloc(**sizeof**(ParkQueue));
235. **if** (!newPQ) exit(0);
236. newPQ->limitNum = 5;
237. newPQ->nowNum = 0;
238. newPQ->NorthHead = initNode();
239. newPQ->NorthTail = newPQ->NorthHead;
240. newPQ->SouthHead = initNode();
241. newPQ->SouthTail = newPQ->SouthHead;
242. **return** newPQ;
243. }
245. **int** isFull(ParkQueue\* PQ) {
246. //出错
247. **if** (PQ->limitNum < PQ->nowNum) **return** -1;
248. **return** PQ->limitNum == PQ->nowNum;
249. }
251. **int** PQpush(ParkQueue\* PQ, Car car, **int** num\_of\_exit)
252. {
253. **if** (isFull(PQ)) {
254. printf("停车场已满。\n");
255. **return** 0;
256. }
257. Node\* newNode = initNode();
258. newNode->ca = car;
259. Node\* temp = (num\_of\_exit ? PQ->NorthTail : PQ->SouthTail);
260. **if** (!temp) exit(0);
261. newNode->next = temp->next;
262. temp->next = newNode;
263. **if** (num\_of\_exit) PQ->NorthTail = newNode;
264. **else** PQ->SouthTail = newNode;
265. PQ->nowNum++;
266. **return** 1;
267. }
269. **int** PQpop(ParkQueue\* PQ, Car &car, **int** num\_of\_exit)
270. {
271. Node\* head = (num\_of\_exit ? PQ->NorthHead : PQ->SouthHead);
272. Node\* temp = head->next;
273. **if** (!temp) {
274. printf("停车场没有车");
275. **return** 0;
276. }
277. head->next = temp->next;
278. temp->next = 0;
279. car = temp->ca;
280. PQ->nowNum--;
281. **return** 1;
282. }
284. **void** print(ParkQueue\* pq)
285. {
286. Node\* head = pq->NorthHead->next;
287. **if** (head != NULL) printf("\n北口\n");
288. **while** (head) {
289. printf("%d ", head->ca.ID);
290. head = head->next;
291. }
292. printf("\n南口\n");
293. head = pq->SouthHead->next;
294. **if** (head != NULL) printf("\n南口\n");;
295. **while** (head) {
296. printf("%d ", head->ca.ID);
297. head = head->next;
298. }
299. }
300. ///////////////////////////////////////////////////////////////////////
301. Car From\_Wait\_To\_Park(WaitList\*\* W, ParkQueue\* PQ)
302. {
303. **if** (isFull(PQ)) {
304. **if** (!isFull\_WaitList(\*W)) printf("停车场已满，请在等候区耐心等待\n\n");
305. **else** printf("等候区已满，不再接收车辆\n\n");
306. **return** { 0,0 };
307. }
308. **if** (isEmpty\_WaitList(\*W)) {
309. printf("\n等候区为空，无法进车\n");
310. **return** { 0,0 };
311. }
312. Node\* p;
313. **if** ((\*W)->Location == 0) //南部进车
314. {
315. (\*W)->Car\_Number--;
316. p = (\*W)->rear;
317. (\*W)->rear = p->pre;
318. p->pre = NULL;
319. (\*W)->rear->next = NULL;
320. PQpush(PQ, p->ca, 0);
321. free(p);
322. }
323. **else** {
324. (\*W)->Car\_Number--;
325. p = (\*W)->rear;
326. (\*W)->rear = p->pre;
327. p->pre = NULL;
328. (\*W)->rear->next = NULL;
329. PQpush(PQ, p->ca, 1);
330. free(p);
331. }
332. printf("进入停车场成功\n\n");
333. **return** p->ca;
334. }

**实验三**

哈夫曼编码

问题定义及需求分析

利用哈夫曼树求得用于通信的二进制编码称为哈夫曼编码。以N中字符出现的频率作为权值，设计电文总长度最短的二进制前缀编码（哈夫曼编码）。

设计哈夫曼编码及解码程序。

（1）采用二叉树等存储结构。

（2）创建哈夫曼树，生成哈夫曼编码。

（3）编码文件的译码，能够完成编码问价和译码文件的相互转换。

（4）可尝试位图文件的压缩问题。

概要设计

* 1. 定义数据类型

1. 定义字母（alph）。建立哈夫曼树用到的字母表即alpha的数组。
2. 定义哈夫曼结点（HTNode），表示设计的哈夫曼树的结点，用哈夫曼结点组成的静态链表表示哈弗曼树。
3. 定义哈夫曼编码表（HuffmanCode），用二维指针实现。
   1. 利用QT进行用户界面设计，将各个操作同界面按钮绑定。
   2. 主程序流程
4. 用户在程序界面点击“选择文件”按钮，选择需要翻译的文件，程序自动辨别是原码文件（.txt）或译码（.y）文件，由此进行不同的操作。
5. 对于原码文件，程序调用文本文件读取函数（read\_txtFile）生成字母表，调用文件转换函数（transFileToStr），获得转换后的字符串，并将其在程序界面输入框内显示。
6. 对于译码文件，程序调用译码文件读取函数（read\_yFile），生成字母表，获得转换字符串，并将其在程序界面的输入框内显示。
7. 程序根据获得的字母表调用哈弗曼树生成函数（HuffmanCoding）生成哈夫曼树（HT）和哈夫曼表（HC）。
8. 用户点击“转换”按钮，程序根据步骤1）的文件类型，进行不同的转换操作。
9. 对于原码(.txt)文件，程序调用编码函数（Huffman\_codeStr），利用生成的哈夫曼编码表（HC），将输入框内的字符（原码文件内容）转换成译码，并在程序界面的输出框内展示。
10. 对于译码(.y)文件，程序调用解码函数（Huffman\_decodeStr），利用生成的哈夫曼数和对应的字母表，将输入框内的字符（译码文件内容）转换成原码，并在程序界面的输出框内展示。
11. 用户点击“保存”按钮，选择文件路径，编辑文件命名，程序根据步骤1）的文件类型判断将输入框内的字符保存为译码文件(.y)或原码文件(.txt)，并进行不同的操作。
12. 对于原码文件（.txt），程序调用字符串保存函数（saveStrToFile），直接将或的原码写入要保存的文件里。
13. 对于译码文件（.y）,程序用头数据写入函数（write\_headData\_decodeFile），将字母表等信息写入要保存的文件里，后字符串保存函数（saveStrToFile），将获得的译码写入要保存的文件里。

详细设计

1. 数据结构

以静态链表形成的二叉树表示哈弗曼树。

1. 数据类型
2. 结构体字母（alph）包含：字符（letter），字符出现的频率（fre）。
3. 结构体哈夫曼结点（HTNode）包含：结点权重（weight），结点的双亲结点（parent），结点的左孩子（lchild），结点的右孩子（rchild）。
4. 哈夫曼数（HuffmanTree）：哈夫曼结点类型的指针，指向哈夫曼结点组成的数组，即静态链表的存储结构。
5. 哈夫曼编码表（HuffmanCode）:字符类型的二维指针，存储每个字符对应的01串。
6. 利用QT自带库函数（QFileDialog::getSaveFileName），完成用户选择文件，程序获取文件路径的功能设计。后根据获得的文件名后缀，判断文件类型。
7. 建立字符的哈希表hash[]，映射到是否在字母表中出现，以及在字母表中的索引，哈希函数为：字符的ASCII码。
8. 读取原码文件操作：依次读取文件中的字符，通过hash[]映射到在字母表中的索引，映射到-1，说明为在字母表中出现过，则在字母表中增加此字符的位置，初始频率设为1，字母表中的索引附给hash[]。再次过程中，应用了堆的思想，出现字符数量超过开始字母表的预设长度时（10），则开辟新的内存，建立更长的字母表。

read\_txtFile:

打开文件filename；

开辟字母表初始空间

(n) = 0;

while (依次读取文件字符) {

哈希函数将读到的字符转换为索引inx

if (hash[inx] == -1) {

(n)++;

if(读取字符个数超出字母表大小){

开辟新的字母表;

更新字母表大小;

}

将读取的字符种类和频率附给字母表；

增加新的哈希映射；

}//if

else {

更新字母表中此字符的频率;

}//else

}//while

返回开辟的字母表;

1. 读取译码文件操作：首先读取译码文件中的头数据，头数据中包含生成此译码的字母表以及表中字母的数量，后根据字母数量，开辟同等大小的字母表空间，后依次读取头数据中包含的字母种类和对应的出现频率。重新形成字母表。原数据读取完成后，依次读取文件字符获得译码。

read\_yFile:

打开文件filename;

读取字母数量n；

开辟字母表，长度为n；

while（n!=0）{

读取字符以及字符出现次数；

将字符放到字符表里；

n--;

}

while(没到文件末尾)｛

读取文件字符，赋值给字符串str

｝

返回字母表和str

1. 在哈夫曼树中选择权重最小的两个结点：遍历表示哈夫曼树的数组，开始设置两个哨兵，利用打擂台算法取得最小的两个结点。

Select：

设置哨兵以及对应的索引；

for(i = 1;i < 数组长度n; ++i){

if(HT[i]的双亲不为0) continue；

打擂台算法进行哨兵更新；

}

返回最后的哨兵及其索引；

1. 生成哈夫曼树和哈夫曼编码表：

生成哈弗曼树：有n个字母，开辟长度为m=2\*n+1的哈夫曼结点数组，作为哈夫曼树。每次在未挑选过的结点里挑选两个权重最小的结点，合并成一个结点，加入到哈夫曼数组里，同时完成左右孩子的连接。进行n+1次，建树成功，最终数组中第m-1个元素为哈夫曼树的根节点。

生成哈夫曼编码表：从叶子结点到根逆向求每个字符的哈夫曼编码。

HuffmanCoding：

//生成哈夫曼树

if (n <= 1) return;

m = 2 \* n + 1;

开辟长度为(m+1)的哈夫曼树HT（不使用0号单元）；

前n个元素初始化；

后n+1元素初始化；

for (i = 1; i<n; ++i) {

在HT[1..i-1]中挑选双亲结点为0，且权重最小的两个结点s1,s2；

将s1,s2合并成一个结点s;

s同s1,s2建立起双亲孩子关系，并加入到哈夫曼树里;

}

//生成哈夫曼编码表

开辟长度为n+1的指针数组HC；

开辟长度为n的字符数组cd;

for (i = 1; i < n; ++i){

f = 叶子结点c=HC[i]的双亲结点；

while（f不为0）｛

if(c是f的左孩子) “0”加入cd

else “1”加入cd

｝

为第i个字符编码分配空间；

将cd逆转赋值给HC[i];

}

1. 原码文件转换（Huffman\_codeStr）：依次读取从原码文件获得的字符串的每个字符，利用哈希函数找到字符在hash[]中的索引，并映射到在哈夫曼编码表（HT）中的索引，从而找到每个字符对应的译码（01串），并将形成的各个字符串连接，组成译码。
2. 译码文件转换：依次读取从译码文件获得的字符串的每个字符，从哈夫曼树的根节点向下寻找，根据字符类型（“0”或“1”）即路径上的编码，决定继续找左子树还是右子树，直到结点的左右孩子都为0，说明找到了叶子结点。此时，结点在哈弗曼树（HC）中的索引，即在字母表（table）中的索引，便找到此01串对应的字符。将字符全部连接，即获得译码对应的原码。

Huffman\_decodeStr：

设p为数的根节点；

遍历原码字符串s的每个字符s[i]｛

由s[i]的值（“0”或“1”）判断继续向左（右）子树寻找；

if(p是叶子结点)｛

将字母表中对应索引的字符取出；

p重新变为根节点，从头开始寻找。

｝

｝

1. 原码文件保存：读取输出框字符，由QT库函数（QFileDialog::getSaveFileName）获得保存路径和保存文件名，后将原码直接写入文件。
2. 译码文件保存：读取输出框字符，由QT库函数（QFileDialog::getSaveFileName）获得保存路径和保存文件名。后先向文件中写入头数据，后写入译码。

write\_headData\_decodeFile：

打开目标文件；

遍历字母表｛

将字母表元素存入目标文件；

｝

1. 界面按钮功能绑定：

点击“选择文件”：

获取文件路径和文件名；

if(是原码文件)｛

读取原码文件，获得文件内容；

｝

else{

读取译码文件，获得文件内容；

}

建树，建表；

输出获得文件内容；

点击“转换”：

获得输入框字符串；

if(选择文件为原码文件)

原码文件装换；

else if（是译码文件）

译码文件转化;

输出框显示字符串；

点击“保存”：

读取保存文件路径和保存文件名字；

if(选择文件时原码文件)

写入头数据；

写入字符串；

调试分析

* 1. 调试过程中遇到形参为指针，但是条用函数后实参值没有变化的问题。经过分析，原因是C语言传值的特性，当实参也是指针时，在函数内对新开辟空间的指向会在函数执行结束后释放收回，造成指针变为野指针的现象。解决方案为将函数的返回值设为同类型的指针。
  2. 时间复杂度分析：

程序主要分为：核心部分，哈夫曼树和哈夫曼编码表的建立；文件读写部分，文件的读取和写入；翻译部分，编码和译码。

1. 核心部分：建立哈夫曼树的过程中，挑选两个权重最小的元素的时间复杂度为O(n)，一共挑选（n-1）次，所以挑选总的时间复杂度为O(n^2)。树的建立过程复杂度为O(nlogn),所以建树的总时间复杂度为O(n^2)。哈夫曼编码表建立过程中，对每个字符都要遍历哈弗曼树，所以时间复杂度为O(n^2)。总的时间复杂度为O(n^2)。
2. 文件读写部分：遍历文件获取原（译）码字符，时间复杂度为O(n)。
3. 翻译部分：编码部分，由于完美哈希函数的选取使得在哈夫曼查找表中查找字符对应编码的时间复杂度为O(1)，所以编码总的时间复杂度为O(n)。译码部分：对于每个译码串，都需要遍历哈夫曼树取得在哈夫曼树中的索引下标O(n^2)，后通过哈希函数映射到字母表中的索引O(1)。所以总时间复杂度为O(n^2)。

所以程序总的时间复杂度为O(n^2)。

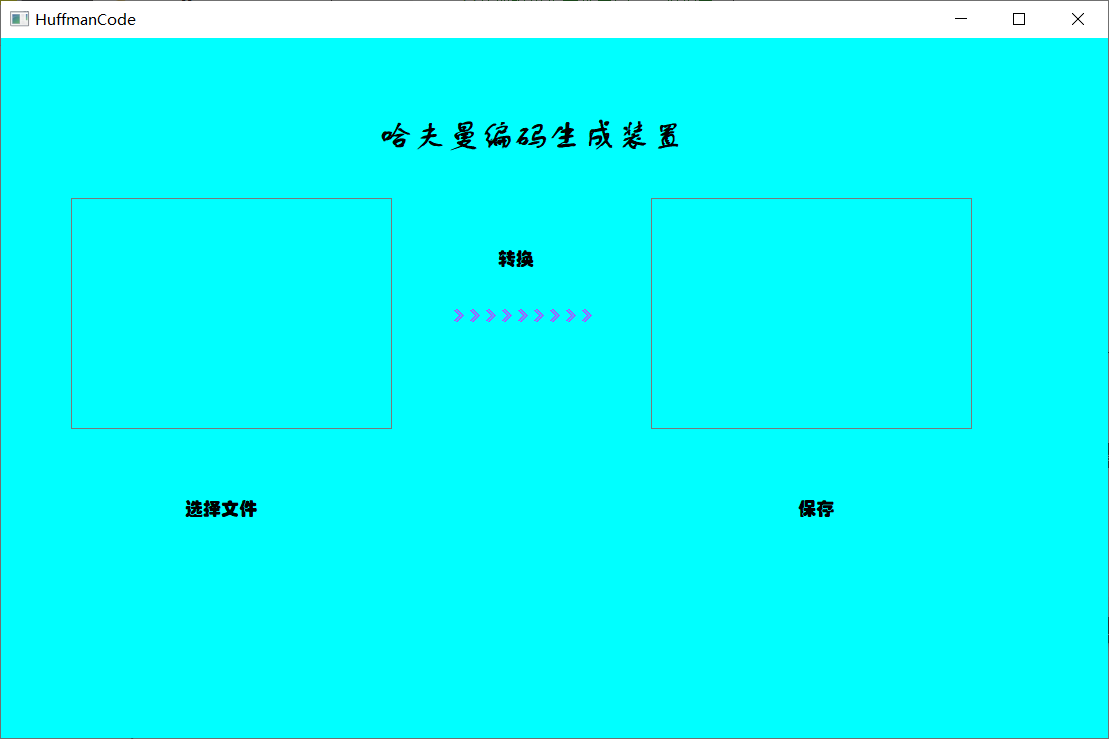
* 1. 空间复杂度分析：建立字母表需要开辟0(n)的空间，建立哈弗曼树需要开辟0(2\*n+1)的空间，建立哈夫曼编码表需要开辟0(n^2)的空间。所以总的空间复杂度为O(n^2)。
  2. 改进设想：

可以将生成的01译码每八位变成一个字节，通过这种方式降低空间复杂度。

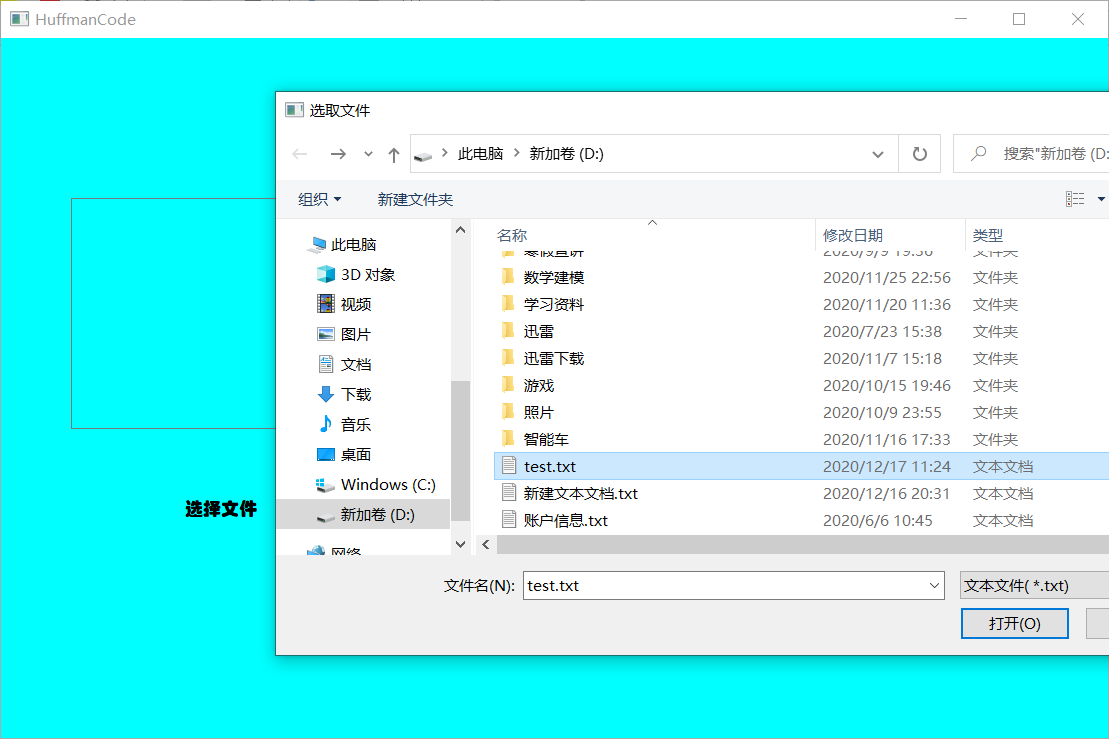
使用说明

进入程序界面，点击“选择文件”按钮，选择要编码或解码的文件。后文件内容会显示到界面的输入框内。后点击“转换”按钮，文件的译码或原码就会显示在界面的输出框内。点击“保存”按钮，选择文件保存路径，编辑文件命名，点击“确定”，会提示文件“保存成功”，到此完成一个文件的编码（解码）。

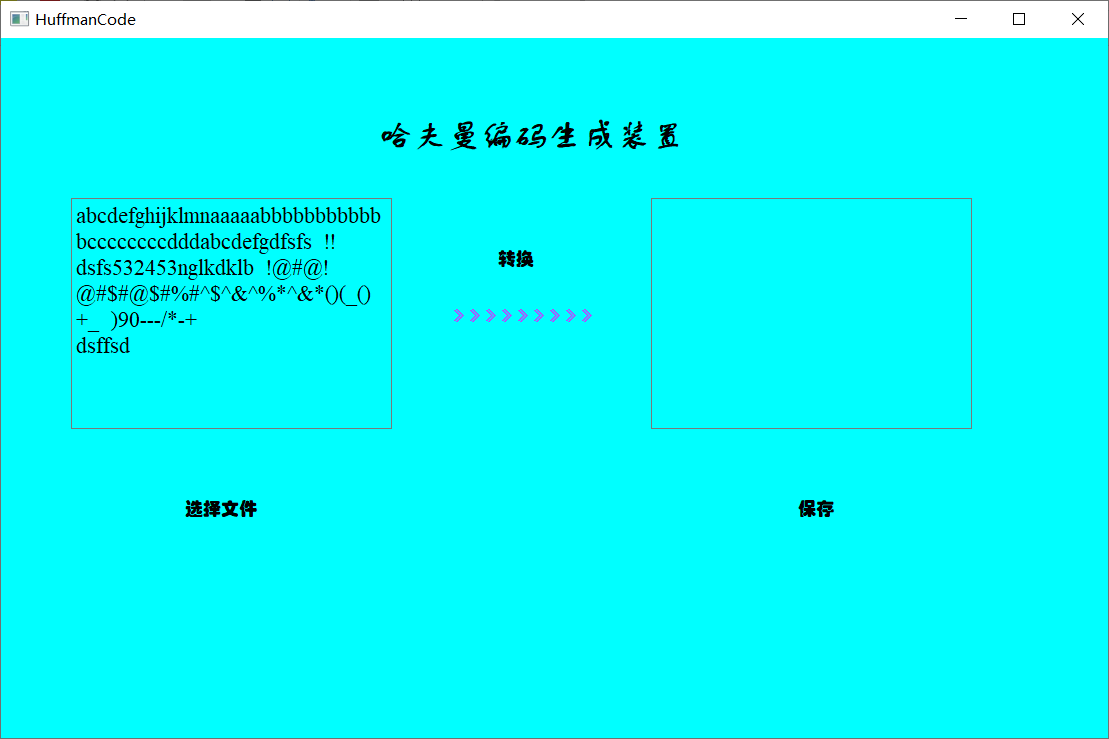
测试结果



程序初始界面



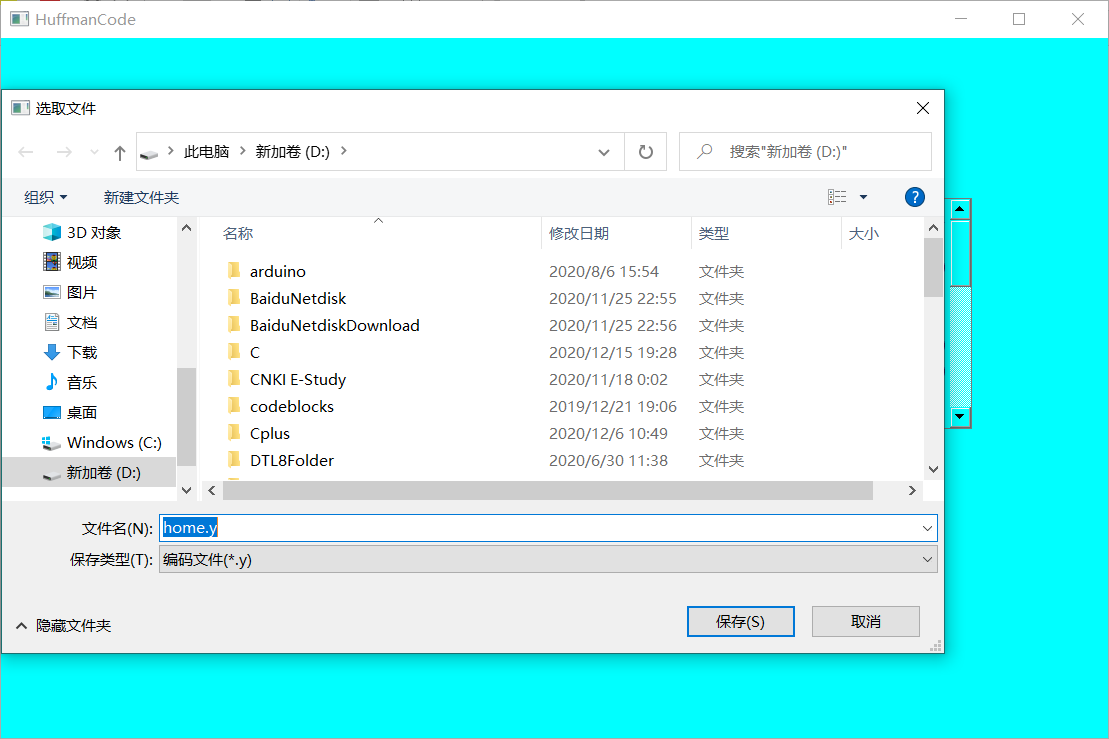
选择转换文件



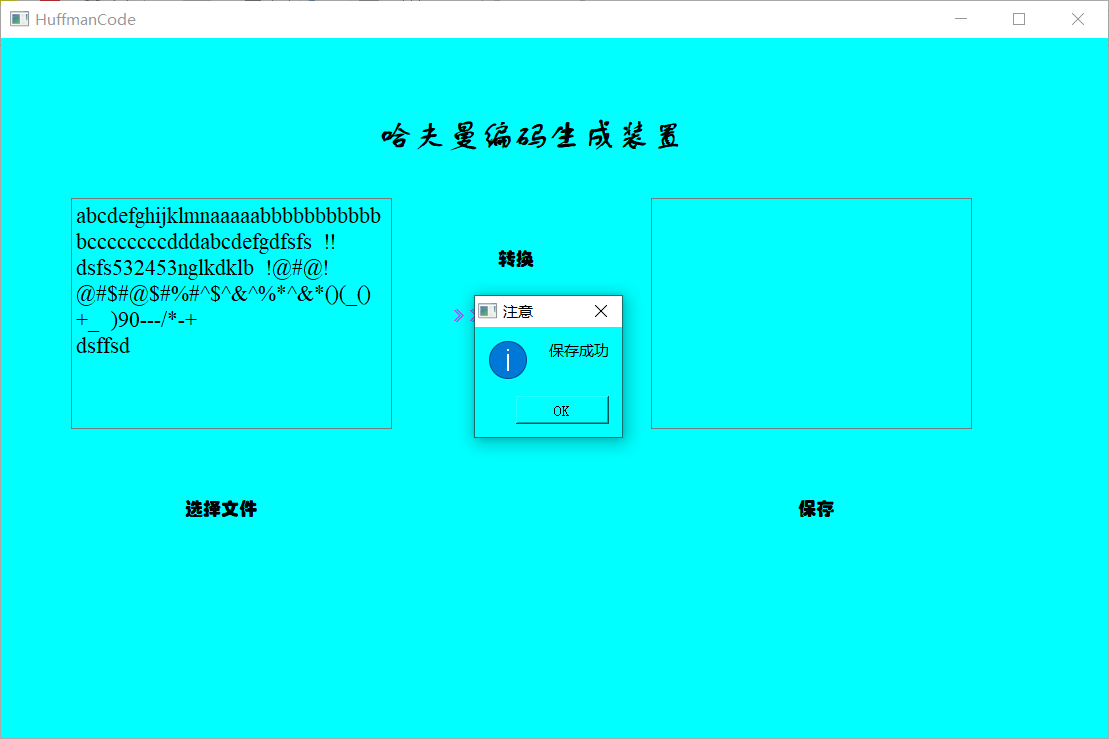
原码文件内容展示



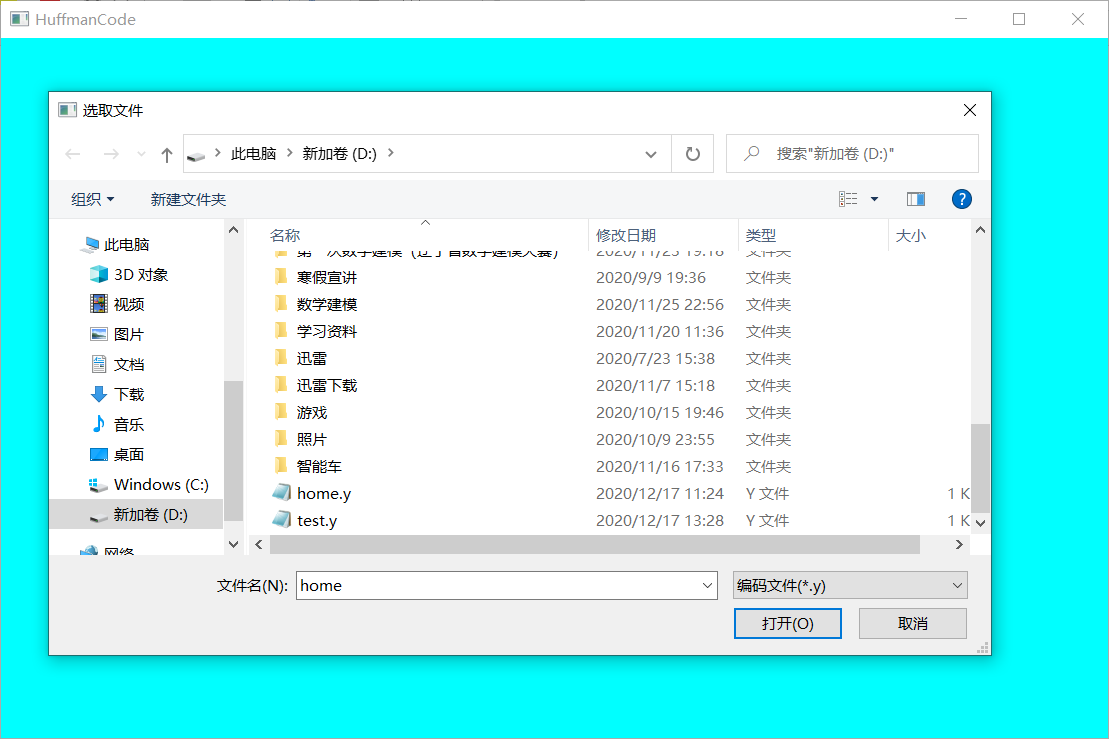
编码转换



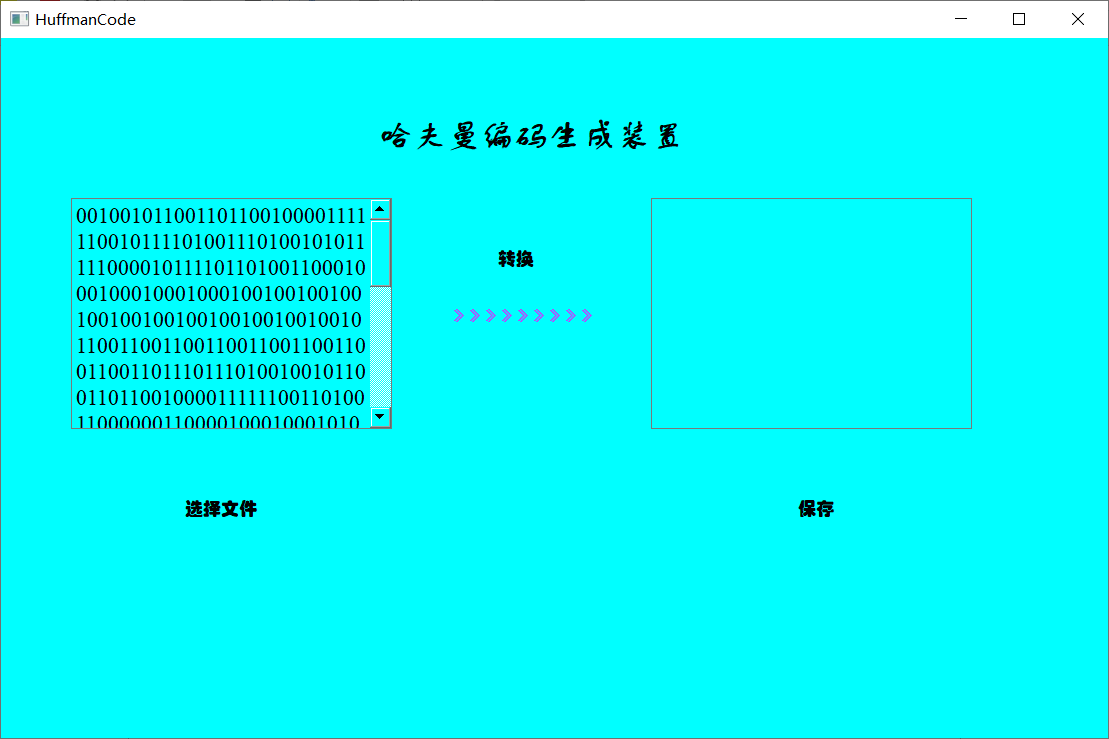
选择文件保存路径，编辑文件名



文件保存成功提示



选择译码文件



译码文件内容展示



译码转换

附录

1. /\*\*\*\*\*\*\*\*\*main.cpp\*\*\*\*\*\*\*\*\*\*\*/
2. //主函数：界面进入
4. #include "mainwindow.h"
6. #include <QApplication>
8. **int** main(**int** argc, **char** \*argv[])
9. {
10. QApplication a(argc, argv);
11. MainWindow w;
12. w.show();
13. **return** a.exec();
14. }
15. /\*\*\*\*\*\*\*\*\*\*\*\*mainwindow.h\*\*\*\*\*\*\*\*\*\*/
16. //界面类定义
18. #ifndef MAINWINDOW\_H
19. #define MAINWINDOW\_H
21. #include <QMainWindow>
22. #include "struct.h"
24. QT\_BEGIN\_NAMESPACE
25. **namespace** Ui { **class** MainWindow; }
26. QT\_END\_NAMESPACE
28. **class** MainWindow : **public** QMainWindow
29. {
30. Q\_OBJECT
32. **public**:
33. MainWindow(QWidget \*parent = nullptr);
34. ~MainWindow();
36. **private** slots:
37. **void** on\_saveButton\_clicked();
39. **void** on\_choseButton\_clicked();
41. **void** on\_transButton\_clicked();
43. **private**:
44. Ui::MainWindow \*ui;
45. HuffmanTree HT;
46. HuffmanCode HC;
47. alph \*atable;
48. **int** num\_char;
49. **int** hash[257];
50. **bool** is\_yFile;
51. **bool** have\_chosedFile;
52. };
53. #endif // MAINWINDOW\_H
54. /\*\*\*\*\*\*\*\*\*\*\*mainwindow.cpp\*\*\*\*\*\*\*\*\*\*\*\*/
55. //界面类的实现，按钮功能的实现
57. #include "mainwindow.h"
58. #include "ui\_mainwindow.h"
59. #include <QFileDialog>
60. #include "struct.h"
61. #include <QMessageBox>
62. #include <iostream>
63. #include <string>

66. MainWindow::MainWindow(QWidget \*parent)
67. : QMainWindow(parent)
68. , ui(**new** Ui::MainWindow)
69. {
70. ui->setupUi(**this**);
71. **this**->setWindowTitle(tr("HuffmanCode"));
72. is\_yFile = **false**;
73. have\_chosedFile = **false**;
74. //建立哈夫曼编码树
75. // HC = initHuffmanCode(26);
76. // HT = HuffmanCoding(HC, alphtable, 26);
77. }
79. MainWindow::~MainWindow()
80. {
81. **delete** ui;
82. }
84. **void** MainWindow::on\_saveButton\_clicked()
85. {
86. //获得存储位置和文件名
87. QString fileName;
88. **if**(!is\_yFile) fileName = QFileDialog::getSaveFileName(**this**,tr("选取文件"),"/home",tr("编码文件(\*.y)"));
89. **else** fileName = QFileDialog::getSaveFileName(**this**,tr("选取文件"),"/home",tr("文本文件(\*.txt)"));
91. //获取输出框文件内容，并根据文件位置进行储存
92. **if** (!fileName.isEmpty())
93. {
94. QString str = ui->to\_textEdit->toPlainText();
95. //如果是编码文件，就写入头数据，即字母表
96. **if**(!is\_yFile){
97. //QMessageBox::information(this,tr("注意"),tr("开始写入头数据"));
98. **if**(!write\_headData\_decodeFile((**char**\*)fileName.toStdString().c\_str(), atable, num\_char)){
99. QMessageBox::information(**this**,tr("注意"),tr("头数据写入失败"));
100. **return**;
101. }
102. }
103. **if**(saveStrToFile(fileName.toStdString().c\_str(),str.toStdString().c\_str())){
104. ui->to\_textEdit->clear();
105. QMessageBox::information(**this**,tr("注意"),tr("保存成功"));
106. }
107. **else**
108. QMessageBox::warning(**this**,tr("注意"),tr("保存失败"));
109. }
110. }

113. **void** MainWindow::on\_choseButton\_clicked()
114. {
115. //读取文件名
116. memset(hash,-1,**sizeof**(hash));
117. QString fileName = QFileDialog::getOpenFileName(**this**, tr("选取文件"), "/home", tr("文本文件( \*.txt);;编码文件(\*.y)"));
118. **if**(fileName.isNull()){
119. **return**;
120. }
121. std::string filename = fileName.toStdString();
122. **int** len = fileName.length();
123. QString str;
124. alph\* table;
125. **int** n = 0;
127. //区分文件类型
128. **if**(fileName[len - 1] == 't'){
129. is\_yFile = **false**;
130. //读取原始文件，获得字符串种类
131. table = read\_txtFile(filename.c\_str(), hash, n);
132. //QMessageBox::warning(this,tr("注意"),tr(filename.c\_str()));
133. //获得文件内容
134. str = transFileToStr(filename.c\_str());
135. }
136. **else**{
137. is\_yFile = **true**;
138. //读取y文件
139. table = read\_yFile(filename.c\_str(), str, &n);
140. }
141. **if**(n == 0){
142. QMessageBox::warning(**this**,tr("注意"),tr("n==0"));
143. **return**;
144. }
146. //获取字符种类数和字母表
147. atable = table;
148. num\_char = n;
149. //建树，建表
150. HC = initHuffmanCode(n);
151. HT = HuffmanCoding(HC, table, n);
153. //将从文件获得的字符串在界面上输出
154. **if**(str.isEmpty()){
155. QMessageBox::warning(**this**,tr("注意"),tr("文件不存在或文件内容为空"));
156. }
157. **else**{
158. have\_chosedFile = **true**;
159. ui->from\_textEdit->setText(str);
160. }
161. }
163. **void** MainWindow::on\_transButton\_clicked()
164. {
165. //获得输入框的字符串
166. ui->to\_textEdit->clear();
167. QString from\_str = ui->from\_textEdit->toPlainText();
168. //if(!have\_chosedFile) atable = read\_txt(from\_str.toStdString().c\_str(), hash, num\_char);
170. //求取输出框的字符串
171. QString to\_str;
172. //根据输入文件类型判断输入字符类型，后根据上面获得的哈夫曼树和表，进行转换
173. **if**(from\_str.isEmpty()){
174. QMessageBox::warning(**this**,tr("警告"),tr("输入框为空或输入格式错误"));
175. **return**;
176. }
177. **else** **if**(is\_yFile)
178. to\_str = Huffman\_decodeStr((**char**\*)from\_str.toStdString().c\_str(), from\_str.length(), HT, num\_char, atable);
179. **else**
180. to\_str = Huffman\_codeStr((**char**\*)from\_str.toStdString().c\_str(), HC, hash);
182. //输出转换后的字符串
183. **if**(to\_str.isEmpty())
184. QMessageBox::warning(**this**,tr("警告"),tr("输出为空"));
185. **else** ui->to\_textEdit->setText(to\_str);
186. }
187. /\*\*\*\*\*\*\*\*\*\*\*\*codejie.cpp\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
188. //建树，建表。字符串转换

191. #include "struct.h"
192. #include <string.h>
193. #include <stdlib.h>
194. #include <stdio.h>
195. #include<share.h>
196. #include <QString>
198. //下面三个函数为建树(编码)函数
199. HuffmanCode initHuffmanCode(**int** n)
200. {
201. HuffmanCode HC = (HuffmanCode)malloc((n + 1) \* **sizeof**(**char**\*));
202. **if** (!HC) exit(0);
203. **return** HC;
204. }
206. **void** Select(HuffmanTree HT,**int** n,unsigned **int** \*s1,unsigned **int** \*s2)
207. {
208. **int** mininx = 0, minweight = 100000000, min2inx = 0, min2weight = 100000000;
209. **for** (**int** i = 1; i <= n; ++i) {
210. **if** (HT[i].parent != 0) **continue**;
211. **int** weight = HT[i].weight;
212. **if** (weight > min2weight) **continue**;
213. **if** (weight < minweight) {
214. min2inx = mininx, min2weight = minweight;
215. mininx = i, minweight = weight;
216. }
217. **else** min2inx = i, min2weight = weight;
218. }
219. \*s1 = mininx; \*s2 = min2inx;
220. }
222. HuffmanTree HuffmanCoding(HuffmanCode HC, alph w[], **int** n)
223. {
224. **int** m;
225. **if** (n < 1) **return** NULL;
226. m = 2 \* n - 1;
227. HuffmanTree HT = (HuffmanTree)malloc((m + 1) \* **sizeof**(HTNode));
228. **if** (!HT) exit(0);
229. **int** i;
230. HuffmanTree p = HT+1;
231. **for** ( i = 1; i <= n; ++i,++p, ++w) {
232. p->weight = w->fre;
233. p->parent = p->lchild = p->rchild = 0;
234. }
235. **if**(n == 1){
236. HC[1] = (**char**\*)("1");
237. **return** HT;
238. }
239. **for** (; i <= m; ++i, ++p) {
240. p->lchild = p->parent = p->rchild = p->weight = 0;
241. }
242. **for** (i = n + 1; i <= m; ++i) {
243. unsigned **int** s1, s2;
244. Select(HT, i-1,&s1,&s2);
245. HT[s1].parent = i; HT[s2].parent = i;
246. HT[i].lchild = s1; HT[i].rchild = s2;
247. HT[i].weight = HT[s1].weight + HT[s2].weight;
248. }
249. **char** \*cd = (**char**\*)malloc(n \* **sizeof**(**char**));
250. **if** (!cd) exit(0);
251. cd[n - 1] = '\0';
252. **for** (**int** i = 1; i <= n; ++i) {
253. **int** start = n - 1;
254. **for** (**int** c = i, f = HT[i].parent; f != 0; c = f, f = HT[f].parent) {
255. **if** (HT[f].lchild == (unsigned)c) cd[--start] = '0';
256. **else** cd[--start] = '1';
257. }
258. HC[i] = (**char**\*)malloc((n - start) \* **sizeof**(**char**));
259. **if** (!HC[i]) exit(0);
260. strcpy\_s(HC[i], n - start, &cd[start]);
261. }
262. free(cd);
263. **return** HT;
264. }
266. //下面两个为解码函数
267. //字符串变字符串（01字符串变字母字符串，返回解码后的字母串）
268. QString Huffman\_decodeStr(**char** s[],**int** lens, HuffmanTree HT,**int** n,alph a[]) {
269. QString str;
270. **int** p = 2\*n-1;
271. **int** i = 0;
272. **for** (i = 0; i < lens; i++) {
273. **if** (s[i] == '0') {
274. **if** (HT[p].lchild != 0) {
275. p = HT[p].lchild;
276. }
277. }
278. **else** **if** (s[i] == '1'){
279. **if** (HT[p].rchild != 0) {
280. p = HT[p].rchild;
281. }
282. }
283. **if** (HT[p].lchild == 0 && HT[p].rchild == 0){
284. // printf("%c", a[p-1].letter);
285. str+=a[p-1].letter;
286. p = 2\*n-1;
287. }
288. }
289. **return** str;
290. }
292. //下面为编码生成函数
293. //字符串变字符串（字母字符串变01字符串。返回编码后的字符串）
294. QString Huffman\_codeStr(**char** alstr[], HuffmanCode HC, **int** hash[])
295. {
296. QString str;
297. **for**(**int** i = 0; alstr[i]; ++i){
298. **int** ch = alstr[i];
299. //if (ch < 'a' || ch>'z') continue;
300. str += HC[hash[ch]+1];
301. }
302. **return** str;
303. }
305. //下面为字符串和文件相互转化
306. //将字符串（无所谓是字母串还是01串）直接变成文件
307. **bool** saveStrToFile(**const** **char**\* filename,**const** **char** \*str)
308. {
309. **FILE** \*fp = fopen(filename,"a+");
310. **if**(!fp) **return** **false**;
311. **for**(**int** i = 0; str[i]; ++i){
312. fputc(str[i],fp);
313. }
314. fclose(fp);
315. **return** **true**;
316. }
318. //将文件内容直接变成字符串
319. QString transFileToStr(**const** **char**\* filename)
320. {
321. QString str;
322. **FILE** \*fp = fopen(filename,"r");
323. **if**(!fp) **return** str;
324. **char** ch;
325. **while**((ch = fgetc(fp)) != EOF)
326. str += ch;
327. fclose(fp);
328. **return** str;
329. }
330. /\*\*\*\*\*\*\*\*\*\*\*\*read\_char.cpp\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
331. //文件读取和写入
333. #include "struct.h"
334. #include <stdio.h>
335. #include <QMessageBox>
337. //读取原始文件，得到字符串的种类和出现频率，储存在table[]中
338. //后利用table[]生成对应的哈弗曼树和哈夫曼表
339. alph\* read\_txtFile(**const** **char**\* filename, **int** hash[], **int** &n)
340. {
341. **FILE** \*fp = fopen(filename,"r");
342. **if** (!fp) **return** NULL;
343. **char** ch;
344. **int** size = 10;
345. alph\* table = (alph\*)malloc(**sizeof**(alph)\*size);
346. (n) = 0;
347. **while** ((ch = fgetc(fp)) != EOF) {
348. **int** inx = ch;
349. **if**(inx < 0) **continue**;
350. **if** (hash[inx] == -1) {
351. (n)++;
352. **if**(n >= size){
353. table = (alph\*)realloc(table, **sizeof**(alph) \* (size+10));
354. size += 10;
355. }
356. **if** (!table) exit(0);
357. table[(n) - 1].letter = ch;
358. table[(n) - 1].fre = 1;
359. hash[inx] = (n)-1;
360. }
361. **else** {
362. table[hash[inx]].fre++;
363. }
364. }
365. fclose(fp);
366. **return** table;
367. }
369. alph\* read\_yFile(**const** **char**\* filename, QString &str, **int** \*n)
370. {
371. **FILE**\* fp = fopen(filename, "r");
372. **int** numoftable = 0;
373. fscanf(fp, "%d", &numoftable);
374. \*n = numoftable;
375. alph\* table = (alph\*)malloc(**sizeof**(alph)\*numoftable);
376. **for**(**int** i = 0; i < numoftable; ++i){
377. **int** m, n;
378. fscanf(fp,"%d %d", &m, &n);
379. table[i].letter = (**char**)m;
380. table[i].fre = n;
381. }
382. **char** ch;
383. str = "\0";
384. ch = fgetc(fp);
385. **while**((ch = fgetc(fp)) != EOF){
386. str += ch;
387. }
388. fclose(fp);
389. **return** table;
390. }
392. //将从原始文件获得的字符信息table[]（头数据）写到编码文件里面，
393. **bool** write\_headData\_decodeFile(**char**\* filename,alph table[],**int** numoftable)
394. {
395. **FILE**\* fp = fopen(filename,"w");
396. **if**(!fp) **return** **false**;
397. fprintf(fp, "%d\n", numoftable);
398. **for** (**int** i = 0; i < numoftable; ++i) {
399. fprintf(fp, "%d %d\n", table[i].letter, table[i].fre);
400. }
401. fclose(fp);
402. **return** **true**;
403. }

**实验四**

哈希表应用

问题定义及需求分析

针对学生的手机号码，设计一个哈希表，使得平均查找长度不超过给定值R。

设计哈希表应用程序。

（1）采用顺序或链式存储结构。

（2）设计哈希函数。

（3）分析平均查找长度特性。

概要设计

1. 定义数据类型：
2. 定义哈希表（HashTable）。本次实验采用顺序存储结构，哈希表由三个数组（散列表存储数组、状态数组、探测次数数组）构成。
3. 定义元素结构（HElemType）.数据最小单元，用来描述元素属性。
4. 主程序流程：
5. 将哈希表初始化（initHashTable），收集电话号码等信息，使用CreateHashTable函数将信息存入哈希表中.
6. 存入过程中，首先将字符型电话号码预处理（converse），转换成数字，将每条电话号码通过地址插入散列表中（Insert）。
7. 插入过程中，需要找到关键码对应地址并通过返回的条件判断能否进行插入操作（FindPos）,若查找成功，则返回地址。
8. 当数据集存放完毕后，可以进行查找（FindPos），添加(Insert)，删除(Remove)等操作

详细设计

（1）元素结构组成HElemType：关键字key[11]（存储手机号码）

（2）哈希表结构组成HashTable；用来进行求余运算的整数divisor，已用地址数n和最大地址数m，散列表存储数组data，状态数组KindOfState，探测次数数组count

（3）哈希表的初始化，对data数组和count数组的内存分配，以及各个指标的初始化

（4）对字符串的预处理，通过converse函数实现，主要思想利用“折叠法”，每次取三个字符，将其转化成数字，然后进行相加

int converse(char A[]){

char\* p = A;

char cont[3];

int cont\_num\_all = 0; //转换后的数字的和

int i;

for (i = 0; i < 11; i++) {

cont[i%2] = \*(p+i);

if (i%2 == 0){

cont\_num\_all += atoi(cont);

}

}

return cont\_num\_all%1000;

}

1. 创建哈希表过程中调用了寻址函数和插入函数，其中插入函数需要调用寻址函数

FindPos:

将电话号码转化为数字

将数字处理为地址（线性探测法哈希函数）

If状态为“活动”且关键码相同

返回true

else do

If状态为“活动”且关键码相同

返回true

else if 状态不是“活动”

返回false

对地址使用线性探测法，加一取模

While 地址和所选地址不同

返回 false

1. 插入函数

Insert：

建立一个flag和i

令flag等于FindPos返回值

If flag//表中有此元素，不能插入

return false

else if i == -1 //表已满，不能插入

return false

将elem、num赋给哈希表，同时将状态改为“活动”

return true

1. ASL函数

ASL：

将count数组中所有值相加，除以总地址数

调试分析

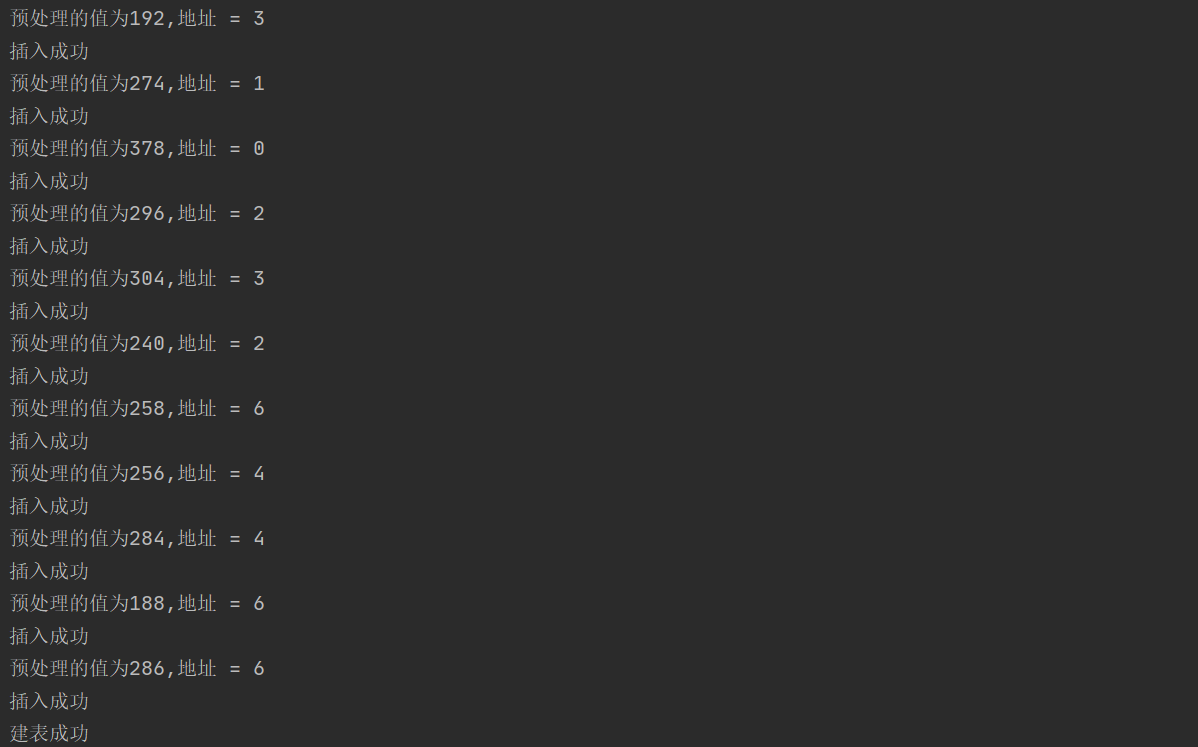
1. 散列表的关键码和存储位置之间直接建立了映像，当选择的散列函数能够得到均匀的地址分布时，在查找过程中可以不做多次检测，查找率较高，但是由于很难避免冲突，这就增加了查找时间。

（2）根据实验分析可得，冲突的出现和散列函数的选取（地址分布是否均匀）、处理冲突的方法（是否产生堆积）有关。从理论上可以认为，各种均匀的散列函数对于同样以组从关键码集合中随机选取的关键码，造成冲突的程度是相同的，多以散列表的性能主要取决于处理冲突的方法。但是在实际应用中，情况并非如此。如在用作关键码的许多标识符具有相同的前缀和后缀，或者是相同字符的不同排列的场合，不同的散列函数往往导致散列表具有不同的查找性能。大量实验表明，链地址法优于开地址法；在散列函数中，除留余数法优于其他类型的散列函数，最差的是折叠法。当装填因子较高时，因选择的散列函数不同，散列表的查找性能差别很大。因此，一般情况下选用除留余数法。

测试结果

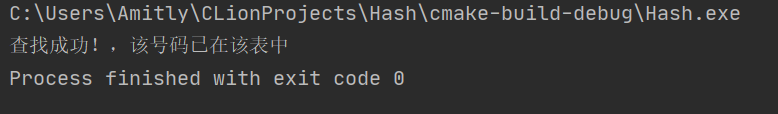
（1）使用11个电话号码进行构建散列表

KeyType \*key\_1[11] = {"15642323166","13342256887","18842286978","18927364283","13345256887","13342254567","13342636887","13342634887","13343256887","13343432147","13234556887"};



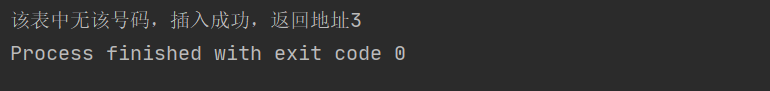
（2）使用其中一则号码作为例子，进行查找

FindPos(HashTable\*\* HT, KeyType x[], int\* i, int\* num);



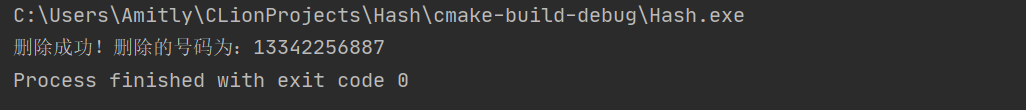
1. 将其他号码插入散列表中

Insert(HashTable\*\* HT, HElemType elem);



1. 删除其中一个号码

Remove(HashTable\*\* HT, KeyType k, HElemType\* elem);



附录

1. #include <stdlib.h>
2. #include <stdio.h>
3. #define defaultSize 15
4. **enum** KindOfState {Active, Blank, Deleted};
5. **typedef** **char** KeyType;
6. **typedef** **struct**{
7. KeyType key[13];
8. }HElemType;
9. **typedef** **struct** {
10. **int** divisor;            //除留余数
11. **int** n,m;                //已用地址数与最大地址数
12. HElemType \*data;        //散列表存储数组
13. **enum** KindOfState \*state;     //状态数组
14. **int** \*count;             //探测次数数组
15. }HashTable;
16. //telephone number 11
17. #define defaultSize 15
18. **void** initHashTable(HashTable\*\* HT, **int** d);
19. **int** FindPos(HashTable\*\* HT, KeyType x[], **int**\* i, **int**\* num);
20. **bool** Insert(HashTable\*\* HT, HElemType elem);
21. **void** createHashTable(HashTable\*\* HT, KeyType \*\*A, **int** n);
22. **bool** Remove(HashTable\*\* HT, KeyType k, HElemType\* elem);
23. **void** printHashTable(HashTable\*\* HT);
24. **int** converse(**char** A[]);
26. #include <stdlib.h>
27. #include <stdio.h>
28. #include <string.h>
29. #include <stdbool.h>
30. #include "HashTable.h"
32. **void** initHashTable(HashTable\*\* HT, **int** d){
33. (\*HT) = (HashTable\*)malloc(**sizeof**(HashTable));
34. (\*HT)->divisor = d;
35. (\*HT)->m = defaultSize; (\*HT)->n = 0;
36. (\*HT)->data = (HElemType\* )malloc((\*HT)->m \* **sizeof**(HElemType));
37. (\*HT)->count = (**int**\*)malloc((\*HT)->m \* **sizeof**(**int**));
39. **for** (**int** i = 0; i < (\*HT)->m; ++i) {
40. (\*HT)->state[i] = Blank;
41. (\*HT)->count[i] = 0;
42. }
43. }
45. **int** converse(**char** A[]){
46. **char**\* p = A;
47. **char** cont[3];
48. **int** cont\_num\_all = 0;   //转换后的数字的和
49. **int** i;
50. **for** (i = 0; i < 11; i++) {
51. cont[i%2] = \*(p+i);
52. **if** (i%2 == 0){
53. cont\_num\_all += atoi(cont);
54. }
55. }
56. **return** cont\_num\_all%1000;
57. }
59. **int** FindPos(HashTable\*\* HT, KeyType x[], **int**\* i, **int**\* num){
60. **int** y = converse(x);
61. \*i = y % (\*HT)->divisor;     printf("预处理的值为%d,地址 = %d\n",y,\*i);
62. \*num = 0;
63. //printf("%d\n",strcmp((\*HT)->data[\*i].key,x));
64. **if** ((\*HT)->state[\*i] == Active&& (strcmp((\*HT)->data[\*i].key,x)==0) ){**return** **true**;}
65. **else**{//printf("%d ",\*i);
66. **int** j = \*i;
67. **do** {
68. (\*num)++;
69. **if** ((\*HT)->state[\*i] == Active && strcmp((\*HT)->data[\*i].key,x)==0 ) {**return** **true**;}
70. **else** **if**((\*HT)->state[\*i] != Active) {**return** **false**;}
71. \*i = (\*i + 1) % (\*HT)->m;
72. }**while** (j != \*i);
74. \*i = -1; **return** **false**;
75. }
76. }
77. **bool** Insert(HashTable\*\* HT, HElemType elem){
78. **int** i, num; **bool** flag;
79. flag = FindPos(HT, elem.key, &i, &num);
80. **if**(flag) {printf("表中有此元素，不能插入\n"); **return** **false**;}
81. **else** **if**(i == -1){printf("表已满，不能插入\n");**return** **false**;}
82. (\*HT)->data[i] = elem; (\*HT)->state[i] = Active;
83. (\*HT)->count[i] = num;
84. (\*HT)->n++;
85. printf("插入成功\n");
86. **return** **true**;
87. }
89. **void** createHashTable(HashTable\*\* HT, KeyType \*\*A, **int** n){
91. HElemType elem ;
92. **int** i;
93. **for** (i = 0; i < n; i++) {
94. strcpy(elem.key,\*(A+i));
95. Insert(HT, elem);
96. }
97. printf("建表成功\n");
98. }
100. **bool** Remove(HashTable\*\* HT, KeyType k, HElemType\* elem){
101. **int** i,num; **bool** flag;
102. flag = FindPos(HT, &k, &i, &num);
103. **if** (!flag) {printf("表中没有找到被删元素\n"); **return** **false**;}
104. **else** {
105. (\*HT)->state[i] = Deleted; (\*HT)->n--;
106. **return** **true**;
107. }
108. }
110. **void** printHashTable(HashTable\*\* HT){
111. **int** i;
112. **for** (i = 0; i < (\*HT)->m; ++i) {
113. **if** ((\*HT)->state[i] == Active ){
114. printf("[%d] %s %d\n",i,(\*HT)->data[i].key,(\*HT)->count[i]);
115. }
116. }
117. }

**实验五**

问题定义和需求分析

STL的栈stack类

（1）实现STL的栈stacke类。

（2）栈stacke类的简单应用。

概要设计

1. 定义栈的模板类（Stack）。
2. 模板类要具有元素出栈函数（Pop），元素入栈函数（Push），判满函数（IsFull），判空函数（IsEmpty），求取长度函数（Length），清空函数（Clear）。

详细设计

1. 模板类（Stack）的保护数据成员有：栈指针（buffer），用来指向开辟的顺序表。栈顶元素指针（top），栈的最大容量（SIZE）。
2. 元素出栈函数：在栈不为空的情况下，返回栈顶元素的值，栈顶指针下移。

Pop:

if(栈空) 返回；

返回栈元素的值；

栈顶指针下移；

1. 元素入栈函数：在栈不满的情况下，将元素加入栈顶指针指向的位置，栈顶指针上移。在栈满的情况下，利用堆的思想，开辟出二倍空间，后将元素深拷贝，栈指针指向新开辟的空间，将元素加入栈顶指针指向的位置，栈顶指针上移。

Push:

If(栈满)｛

新开辟2\*SIZE大小的空间。

将原栈指针（buffer）指向长度为SIZE空间里的元素附给新开辟的空间。

原栈指针指向新开辟的空间。

栈最大容量\*2；

｝

栈顶位置 = 新加入元素；

栈顶指针++;

1. 判满函数，判断栈顶指针的值是否和最大容量相同。
2. 判空函数，判断栈顶指针是否为-1。
3. 求取长度函数：返回栈顶指针的值。
4. 清空函数：栈顶指针值变为-1。

调试分析

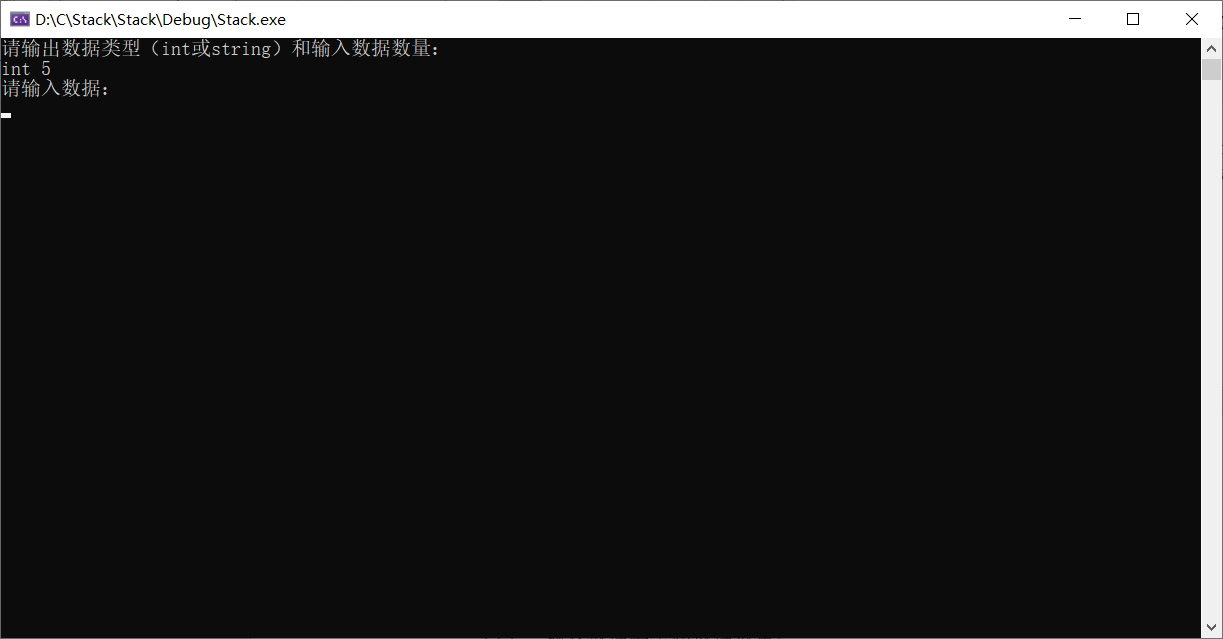
1. 调试过程中，曾发现对于string类型的字符串无法实现功能，通过调试分析，最开始构造函数无法为string类型开辟出空间。经过查阅资料，发现原始是开始使用的malloc函数无法调用string的构造函数，对于难以确认最开始需要字节数的string就无法开辟空间。解决方案是，调用C++的new函数解决，new会调用string的构造函数，实现后续功能。
2. 时空复杂度分析：入栈出栈的时间复杂度都为O(1),但是入栈时处理栈满过程中会出现遍历栈中已有元素的情况，此时复杂度为O(n)。总的时间复杂度为O(n)。空间复杂度为O(n)。
3. 实现栈的类模板过程中发现，对于一个通用的模板的实现是很不容易的，不仅需要考虑到，容器本身功能的实现和内存的相关问题，还要考虑不同数据类型的性质，以获得其通用性。C++的STL模板类本身是很伟大的。

使用说明

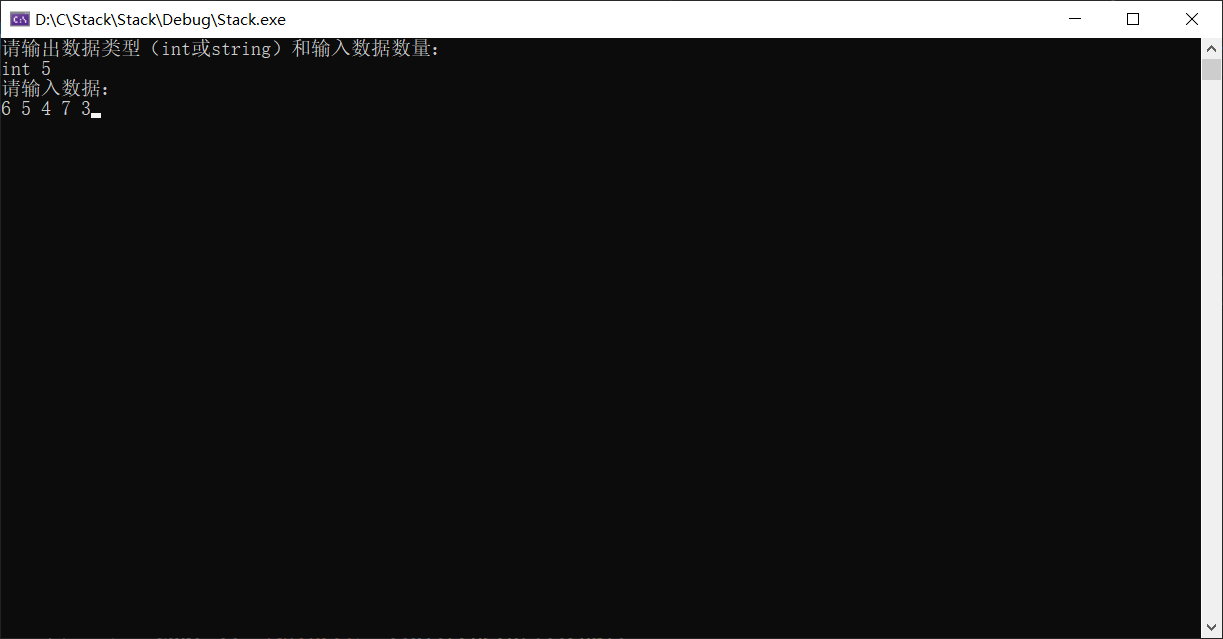
程序运行开始后，输入字符串表示要演示的栈模板的数据类型。这里只取整形（int）和字符串（string）演示。后输入将入栈的元素数量n，后输入n个指定的元素。最后程序给出栈中元素的数量，栈中元素按出栈顺序组成的排列。后可重复上述过程。

测试结果

* 1. 输入数据类型和数据数量



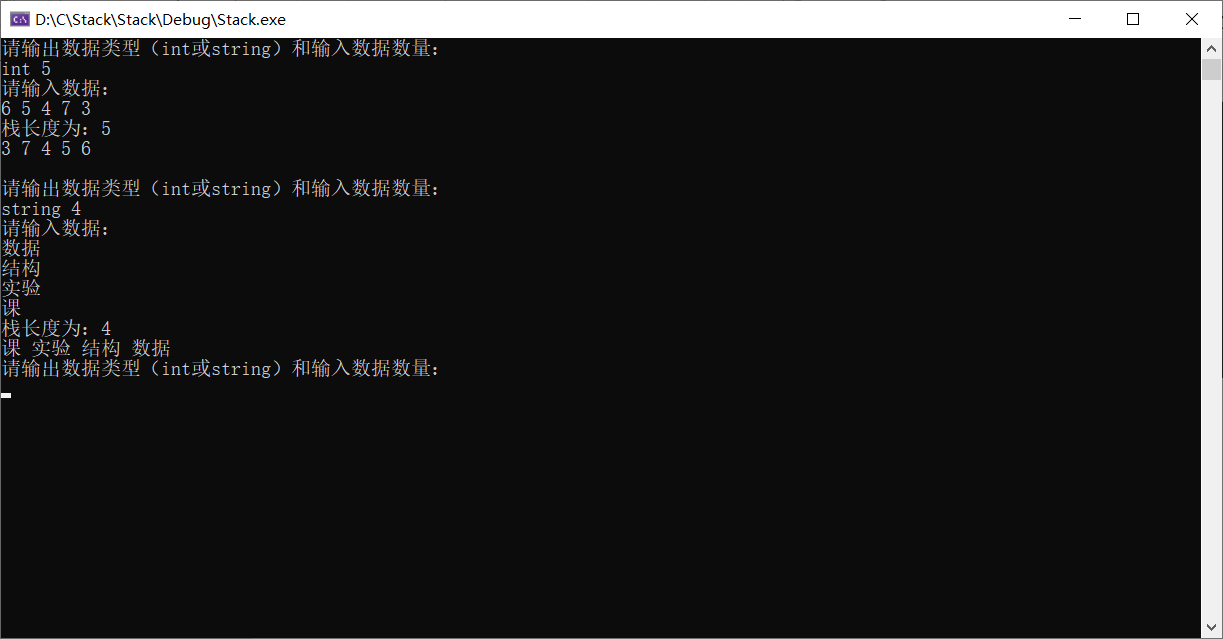
* 1. 输入数据



* 1. 程序给出元素排列



* 1. 字符串类型测试



附录

1. #include<iostream>
2. #include<cassert>
4. **using** **namespace** std;
6. **template**<**typename** T>
7. **class** Stack
8. {
9. **protected**:
10. T \*buffer;
11. **int** top;
12. **int** SIZE;
13. **public**:
14. Stack():top(-1)
15. {
16. SIZE = 50;
17. buffer = **new** T[SIZE];
18. **if** (!buffer) exit(0);
19. //memset(buffer, 0, sizeof(buffer));
20. }
21. **int** Length() { **return** top + 1; }
22. **bool** IsEmpty()**const** { **return** top == -1 ? **true** : **false**; }
23. **bool** IsFull()**const** { **return** top == SIZE - 1 ? **true** : **false**; }
24. **void** Push(**const** T& value)
25. {
26. **if** (!buffer) **return**;
27. **if** (top + 1 >= SIZE && !buffer) {
28. T\* buffer2 = **new** T[SIZE \* 2];
29. **for** (**int** i = 0; i < SIZE; ++i)
30. buffer2[i] = buffer[i];
31. **delete**(buffer);
32. buffer = buffer2;
33. SIZE = SIZE \* 2;
34. }
35. //if(IsFull()) return;
36. buffer[++top] = value;
37. }
38. T Pop()
39. {
40. assert(!IsEmpty());
41. **return** buffer[top--];
42. }
43. **void** Clear()
44. {
45. top = -1;
46. }
48. **const** T& Top()**const**
49. {
50. assert(!IsEmpty());
51. **return** buffer[top];
52. }
53. };
55. **int** main()
56. {
57. string Typename;
58. **int** num;
59. **while**(1){
60. cout << "请输出数据类型（int或string）和输入数据数量：" << endl;
61. cin >> Typename >> num;
62. cout << "请输入数据：" << endl;
63. **if**(Typename=="int")
64. {
65. Stack <**int**> X1;
66. **for**(**int** i=1;i<=num;i++)
67. {
68. **int** x;
69. cin>>x;
70. X1.Push(x);
71. }
72. cout << "栈长度为："<<X1.Length()<<endl;
73. **while**(!X1.IsEmpty())
74. {
75. cout<<X1.Top()<<" ";
76. X1.Pop();
77. }
78. cout<<endl;
79. cout<<endl;
80. }
81. **else**{
82. Stack <string> Y1;
83. **for**(**int** i=1;i<=num;i++)
84. {
85. string y;
86. cin>>y;
87. Y1.Push(y);
88. }
89. cout << "栈长度为：" << Y1.Length() << endl;
90. **while**(Y1.IsEmpty()==0)
91. {
92. cout<<Y1.Top()<<" ";
93. Y1.Pop();
94. }
95. cout<<endl;
96. }
97. }
98. **return** 0;
99. }