

栈与队列

本文档需要用到的编程相关知识点：

C语言：struct（结构体）、typedef、malloc（free）、函数做参数。

本文档的参考教材：

数据结构（C语言版） 严蔚敏

一. 栈的实现

1. 顺序栈的基本操作实现

顺序栈的实现参照教材46页。

```
1  /**
2   * 顺序栈的基本操作实现
3   * 参照教材46页
4   * */
5
6  #include <stdio.h>
7  #include <stdlib.h>
8
9  /**
10   * 宏定义及类型定义
11   * */
12  #define ERROR 0
13  #define OK 1
14  #define TRUE 1
15  #define FALSE 0
16  #define STACK_INIT_SIZE 100 // 栈存储空间初始分配量
17  #define STACKINCREMENT 10 // 栈容量每次增加的值
18
19  typedef int SElemType;
20  typedef int Status;
21
22  /**
23   * 定义顺序栈
24   * */
25  typedef struct {
26      SElemType *base; // 基指针，在栈构造前和销毁之后，base的值为NULL
27      SElemType *top; // 指向栈顶的指针
28      int stacksize; // 当前已分配的存储空间，以元素为单位
29  } SqStack;
30
31  /**
32   * 构造一个空栈S
33   * */
34  Status InitStack(SqStack &S)
35  {
36      S.base = (SElemType *)malloc(STACK_INIT_SIZE * sizeof(SElemType));
```

```

37     if (!S.base)
38         return ERROR; // 存储分配失败
39     S.top = S.base;
40     S.stacksize = STACK_INIT_SIZE;
41     return OK;
42 } // InitStack
43
44 /**
45  * 销毁栈S，S不再存在
46  */
47 Status DestroyStack(SqStack &S)
48 {
49     free(S.base);
50     S.base = NULL;
51     return 0;
52 } // DestroyStack
53
54 /**
55  * 把S置为空栈
56  */
57 Status ClearStack(SqStack &S)
58 {
59     S.top = S.base;
60     return 0;
61 } // ClearStack
62
63 /**
64  * 若栈S为空，返回TRUE， 否则返回FALSE
65  */
66 Status StackEmpty(SqStack S) { return (S.base == S.top) ? TRUE : FALSE; }
67
68 /**
69  * 返回S的元素个数，即栈的长度
70  */
71 int StackLength(SqStack S) { return (S.top - S.base); }
72
73 /**
74  * 若栈不空，则用e返回S的栈顶元素，并返回OK，否则返回ERROR
75  */
76 Status GetTop(SqStack S, SElemType &e)
77 {
78     if (S.base == S.top) // 栈为空
79         return ERROR;
80     e = *(S.top - 1); // 栈顶元素赋给e
81     return OK;
82 } // GetTop
83
84 /**
85  * 若栈不空，返回S的栈顶元素
86  */
87 int GetTop(SqStack S) { return *(S.top - 1); }
88
89 /**
90  * 插入元素e为新的栈顶元素
91  */
92 Status Push(SqStack &S, SElemType e)
93 {
94     if (S.stacksize == StackLength(S)) { // 栈达到了最大容量，自动增加容量

```

```

95     S.base = (SElemType *)realloc(
96         S.base, (S.stacksize + STACKINCREMENT) * sizeof(SElemType));
97     if (!S.base)
98         exit(OVERFLOW); // 存储分配失败
99     S.top = S.base + S.stacksize;
100    S.stacksize += STACKINCREMENT;
101 }
102 *S.top++ = e;
103 return OK;
104 } // Push
105
106 /**
107  * 若栈不空, 则删除S的栈顶元素, 用e返回其值, 并返回OK, 否则返回ERROR
108  */
109 Status Pop(SqStack &S, SElemType &e)
110 {
111     if (S.top == S.base) // 栈为空
112         return ERROR;
113     e = *(--S.top);
114     return OK;
115 } // Pop
116
117 /**
118  * 从栈底到栈顶每个元素依次调用visit(), 且visit失败, 则操作失败
119  */
120 Status StackTraverse(SqStack S, Status (*visit)(SElemType))
121 {
122     SElemType *p;
123     p = S.base; // p指向栈底
124     while (p != S.top) { // 遍历栈的元素
125         visit(*p);
126         p++;
127     }
128     return OK;
129 } // StackTraverse
130
131 /**
132  * 打印元素
133  */
134 Status visit_display(SElemType e)
135 {
136     printf("%d ", e);
137     return OK;
138 }
139
140 Status display_char(SElemType e)
141 {
142     printf("%c ", e);
143     return OK;
144 }

```

顺序栈测试程序：（测试程序代码放在实现代码下方即可）

```

1 int main()
2 {

```

```

3   SqStack stack1;
4   InitStack(stack1);
5   int arr1[8] = {33, 12, 75, 0, 49, 67, 8, 999};
6   int arr2[3] = {24, 81, 100};
7   for (int i = 0; i < sizeof(arr1) / sizeof(int); i++)
8       Push(stack1, arr1[i]);
9
10  printf("顺序栈stack1中有%d个元素，这%d个元素为: \n", StackLength(stack1),
11        StackLength(stack1));
12  StackTraverse(stack1, visit_display);
13
14  // 将stack1的栈顶元素删除，并将arr2数组中的元素一次放入栈中，最后返回栈顶元素有
15  int a1 = 0;
16  Pop(stack1, a1);
17  for (int i = 0; i < sizeof(arr2) / sizeof(int); i++)
18      Push(stack1, arr2[i]);
19  printf("\nstack1更新后有%d个元素\n其中栈顶元素为: %d", StackLength(stack1),
20        GetTop(stack1));
21
22  DestroyStack(stack1);
23
24  return 0;
25 }

```

测试程序运行结果为:

顺序栈stack1中有8个元素，这8个元素为:

33 12 75 0 49 67 8 999

stack1更新后有10个元素

其中栈顶元素为: 100

2. (顺序) 栈的应用

- 括号匹配的检验

在已实现顺序栈的情况下，定义一个用于括号匹配检验的函数，若左右括号匹配则返回OK，不匹配返回ERROR。

```

1  /**
2   * 教材49页，3.2.2 括号匹配的检验，匹配返回OK，不匹配返回ERROR
3   */
4  Status checkBracket(char *expr)
5  {
6      int i = 0;
7      SqStack S;
8      InitStack(S);
9      SElemType e;
10     while ('\0' != *(expr + i)) {
11         switch (*(expr + i)) {
12             case '(':
13                 Push(S, -1);
14                 break;
15             case '[':

```

```

16         Push(S, -2);
17         break;
18     case '{':
19         Push(S, -3);
20         break;
21     case ')':
22         if (OK == GetTop(S, e)) { // 栈非空, 得到栈顶元素
23             if (-1 == e) // 栈顶元素与'{'匹配
24                 Pop(S, e); // 删除栈顶元素
25             else // 不匹配, 返回ERROR
26                 return ERROR;
27         } else
28             return ERROR;
29         break;
30     case ']':
31         if (OK == GetTop(S, e)) {
32             if (-2 == e)
33                 Pop(S, e);
34             else
35                 return ERROR;
36         } else
37             return ERROR;
38         break;
39     case '}':
40         if (OK == GetTop(S, e)) {
41             if (-3 == e)
42                 Pop(S, e);
43             else
44                 return ERROR;
45         } else
46             return ERROR;
47         break;
48     default:
49         break;
50 } // switch
51
52     i++;
53 } // while
54
55 if (StackEmpty(S))
56     return OK;
57 else
58     return ERROR;
59 }
60
61 /**
62  * 输出括号匹配的检验结果
63  */
64 void outputResult(Status i, char *k)
65 {
66     OK == i ? printf("表达式 \"%s\" 左右括号数匹配。\\n", k)
67             : printf("表达式 \"%s\" 左右括号数不匹配。\\n", k);
68 }
69
70 // 测试程序
71 int main()
72 {

```

```

73     char a[] = "class A {private: char a; int b[]; public: A(){} ~A()  
    {} }";  
74     char c[] = "int main() { float a[] = {1.0, 5.2}; return 0;}";  
75     outputResult(checkBracket(a), a);  
76     outputResult(checkBracket(c), c);  
77     return 0;  
78 }

```

测试程序运行结果:

表达式 "class A {private: char a; int b[]; public: A(){} ~A(){} }" 左右括号数不匹配。

表达式 "int main() { float a[] = {1.0, 5.2}; return 0;} " 左右括号数匹配。

• 迷宫问题求解

迷宫问题求解算法的实现参照教材51页。

在迷宫地图中，用1表示墙，用0表示通道，地图四周都为墙，即地图矩阵的外围元素都为1。左上角为入口，右下角为出口。

```

1  /**
2   * 迷宫问题求解算法的实现
3   * 部分内容参照教材51页
4   */
5
6  #include <stdio.h>
7  #include <stdlib.h>
8
9  #define ERROR 0
10 #define OK 1
11 #define TRUE 1
12 #define FALSE 0
13 #define OVERFLOW -1
14 #define STACK_INIT_SIZE 100 // 栈初始化值
15 #define STACKINCREMENT 10 // 栈容量每次增加的值
16 #define SIZE 10 // 定义迷宫地图大小
17
18 typedef struct {
19     int row; // 行, 0起始
20     int col; // 列, 0起始
21 } PostType;
22
23 typedef struct {
24     int ord; // 通道块在路径上的序号
25     PostType seat; // 通道块在迷宫中的坐标位置
26     int di; // 从此通道走向下一通道块的方向
27 } SElemType;
28
29 typedef int Status;
30
31 /**
32 * 定义顺序栈
33 */
34 typedef struct {
35     SElemType *base; // 栈底指针, 在栈构造前和销毁之后, base的值为NULL
36     SElemType *top; // 栈顶指针
37     int stacksize; // 当前已分配的存储空间, 以元素为单位

```

```

38 } SqStack;
39
40 /**
41  * 构造一个空栈S
42  */
43 Status InitStack(SqStack &S)
44 {
45     S.base = (SElemType *)malloc(STACK_INIT_SIZE * sizeof(SElemType));
46     if (!S.base)
47         return ERROR; // 存储分配失败
48     S.top = S.base;
49     S.stacksize = STACK_INIT_SIZE;
50     return OK;
51 } // InitStack
52
53 /**
54  * 销毁栈S，S不再存在
55  */
56 Status DestroyStack(SqStack &S)
57 {
58     free(S.base);
59     S.base = NULL;
60     return 0;
61 } // DestroyStack
62
63 /**
64  * 把S置为空栈
65  */
66 Status ClearStack(SqStack &S)
67 {
68     S.top = S.base;
69     return 0;
70 } // ClearStack
71
72 /**
73  * 若栈S为空，返回TRUE， 否则返回FALSE
74  */
75 Status StackEmpty(SqStack S) { return (S.base == S.top) ? TRUE : FALSE; }
76
77 /**
78  * 返回S的元素个数，即栈的长度
79  */
80 int StackLength(SqStack S) { return (S.top - S.base); }
81
82 /**
83  * 若栈不空，则用e返回S的栈顶元素，并返回OK，否则返回ERROR
84  */
85 Status GetTop(SqStack S, SElemType &e)
86 {
87     if (S.base == S.top) // 栈为空
88         return ERROR;
89     e = *(S.top - 1); // 栈顶元素赋给e
90     return OK;
91 } // GetTop
92
93 /**
94  * 插入元素e为新的栈顶元素

```

```

95  */
96  Status Push(SqStack &S, SElemType e)
97  {
98      if (S.stacksize == StackLength(S)) { // 栈达到了最大容量, 自动增加容量
99          S.base = (SElemType *)realloc(
100              S.base, (S.stacksize + STACKINCREMENT) *
sizeof(SElemType));
101          if (!S.base)
102              exit(OVERFLOW); // 存储分配失败
103          S.top = S.base + S.stacksize;
104          S.stacksize += STACKINCREMENT;
105      }
106      *S.top++ = e;
107      return OK;
108  } // Push
109
110  /**
111   * 若栈不空, 则删除S的栈顶元素, 用e返回其值, 并返回OK, 否则返回ERROR
112   */
113  Status Pop(SqStack &S, SElemType &e)
114  {
115      if (S.top == S.base) // 栈为空
116          return ERROR;
117      e = *(--S.top);
118      return OK;
119  } // Pop
120
121  /**
122   * 从栈底到栈顶每个元素依次调用visit(), 且visit失败, 则操作失败
123   */
124  Status StackTraverse(SqStack S, Status (*visit)(SElemType))
125  {
126      SElemType *p;
127      p = S.base; // p指向栈底
128      while (p != S.top) { // 遍历栈的元素
129          visit(*p);
130          p++;
131      }
132      return OK;
133  } // StackTraverse
134
135  /**
136   * 打印元素
137   */
138  Status visit_display(SElemType e)
139  {
140      printf("%d ", e);
141      return OK;
142  }
143
144  Status display_char(SElemType e)
145  {
146      printf("%c ", e);
147      return OK;
148  }
149
150  extern SqStack S;
151  extern int Map[SIZE][SIZE];

```



```

152
153 /**
154  * 判断当前位置能不能通过，当前位置可通过是指该位置未曾到达过
155  */
156 bool Pass(PosType curpos)
157 {
158     if (1 == Map[curpos.row][curpos.col])
159         return false;
160     return true;
161 }
162
163 /**
164  * 在走过的地方留下足迹
165  */
166 void FootPrint(PosType curpos) { Map[curpos.row][curpos.col] = 1; }
167
168 /**
169  * 下一个位置，di从1到4表示从东到北
170  */
171 PosType NextPos(PosType pos, int di)
172 {
173     PosType curpos;
174     switch (di) {
175         case 1: // 东，列加1
176             curpos.row = pos.row;
177             curpos.col = pos.col + 1;
178             break;
179         case 2: // 南，行加1
180             curpos.row = pos.row + 1;
181             curpos.col = pos.col;
182             break;
183         case 3: // 西，列减1
184             curpos.row = pos.row;
185             curpos.col = pos.col - 1;
186             break;
187         case 4: // 北，行减1
188             curpos.row = pos.row - 1;
189             curpos.col = pos.col;
190             break;
191     }
192     return curpos;
193 }
194
195 /**
196  * 算法3.3，迷宫算法
197  */
198 Status MazePath(PosType start, PosType end)
199 {
200     PosType curpos;
201     int curstep = 1; // 探索第一步
202     curpos = start;
203     SElemType e; // 通道块
204     do {
205         if (Pass(curpos)) { // 如果当前位置能通过
206             FootPrint(curpos); // 留下足迹
207             e.ord = curstep;
208             e.seat = curpos;
209             e.di = 1;

```

```

210     Push(S, e); // 加入路径
211     if (curpos.row == end.row && curpos.col == end.col)
212         return (OK); // 到达终点
213     curpos = NextPos(curpos, 1); // 将当前位置的东邻设为下一位置
214     curstep++; // 探索下一步
215 } else { // 如果当前位置不能通过
216     if (!StackEmpty(S)) {
217         Pop(S, e);
218         while (4 == e.di && !StackEmpty(S))
219             Pop(S, e); // 即该位置的4方向都已经探索完成, 就再退回一步
220         if (e.di < 4) { // 换一个方向探索
221             e.di++;
222             Push(S, e);
223             curpos = NextPos(e.seat, e.di);
224         }
225     }
226 }
227 } while (!StackEmpty(S));
228
229 return ERROR;
230 }
231
232 Status display_path(SElemType e)
233 {
234     printf("(%d, %d) ", e.seat.row, e.seat.col);
235     switch (e.di) {
236     case 1:
237         printf("东\n");
238         break;
239     case 2:
240         printf("南\n");
241         break;
242     case 3:
243         printf("西\n");
244         break;
245     case 4:
246         printf("北\n");
247         break;
248     }
249     return OK;
250 }
251
252 // 测试程序
253 int Map[SIZE][SIZE] = {{1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
254                        {1, 0, 0, 1, 0, 0, 0, 1, 0, 1},
255                        {1, 0, 0, 1, 0, 0, 0, 1, 0, 1},
256                        {1, 0, 0, 0, 0, 1, 1, 0, 0, 1},
257                        {1, 0, 1, 1, 1, 0, 0, 0, 0, 1},
258                        {1, 0, 0, 0, 1, 0, 0, 0, 0, 1},
259                        {1, 0, 1, 0, 0, 0, 1, 0, 0, 1},
260                        {1, 0, 1, 1, 1, 0, 1, 1, 0, 1},
261                        {1, 1, 0, 0, 0, 0, 0, 0, 0, 1},
262                        {1, 1, 1, 1, 1, 1, 1, 1, 1, 1}}; // 迷宫地图
263 SqStack S; // 路径栈
264
265 int main()
266 {

```

```
267     InitStack(S);
268     PostType start;
269     PostType end;
270     start.row = 1;
271     start.col = 1;
272
273     end.row = 8;
274     end.col = 8;
275
276     MazePath(start, end);
277     StackTraverse(S, display_path);
278     return 0;
279 }
```

测试程序运行结果：

(1, 1) 东
(1, 2) 南
(2, 2) 南
(3, 2) 西
(3, 1) 南
(4, 1) 南
(5, 1) 东
(5, 2) 东
(5, 3) 南
(6, 3) 东
(6, 4) 东
(6, 5) 南
(7, 5) 南
(8, 5) 东
(8, 6) 东
(8, 7) 东
(8, 8) 东

二. 队列的实现

1. 链队列的基本操作实现

链队列的实现参照教材61页。

```
1  /**
2   * 链队列的基本操作实现
3   * 参照教材61页
4   * */
5
6  #include <stdio.h>
7  #include <stdlib.h>
8
9  /**
10   * 宏定义及类型定义
11   * */
12  #define ERROR 0
13  #define OK 1
14  #define TRUE 1
15  #define FALSE 0
16
17  typedef int QElemType;
18  typedef int Status;
19
20  typedef struct QNode {
21      QElemType data;
22      struct QNode *next;
23  } QNode, *QueuePtr;
24
25  /**
26   * 定义链队列
27   * */
28  typedef struct {
29      QueuePtr front; // 队头指针
30      QueuePtr rear;  // 队尾指针
31  } LinkQueue;
32
33  /**
34   * 构造一个空队列Q
35   * */
36  Status InitQueue(LinkQueue &Q)
37  {
38      Q.front = (QNode *)malloc(sizeof(QNode));
39      if (!Q.front)
40          return ERROR;
41      Q.front->next = NULL;
42      Q.rear = Q.front;
43      return OK;
44  } // InitQueue
45
46  /**
47   * 销毁队列Q, Q不再存在
48   * */
49  Status DestroyQueue(LinkQueue &Q)
50  {
```

```

51     while (Q.front) {
52         Q.rear = Q.front->next;
53         free(Q.front);
54         Q.front = Q.rear;
55     }
56     return OK;
57 } // DestroyQueue
58
59 /**
60  * 将Q清为空队列，并释放结点空间
61  */
62 Status ClearQueue(LinkQueue &Q)
63 {
64     QueuePtr p, temp;
65     p = Q.front->next;
66     Q.front->next = NULL;
67     Q.rear = Q.front;
68     while (p) {
69         temp = p->next;
70         free(p);
71         p = temp;
72     }
73     return OK;
74 } // ClearQueue
75
76 /**
77  * 判断Q是否为空队列，是返回TRUE，否则返回FALSE
78  */
79 Status QueueEmpty(LinkQueue Q)
80 {
81     if (Q.front == Q.rear)
82         return TRUE;
83     else
84         return FALSE;
85 } // QueueEmpty
86
87 /**
88  * 返回Q的元素个数，即队列的长度
89  */
90 int QueueLength(LinkQueue Q)
91 {
92     QueuePtr p;
93     int len = 0;
94     p = Q.front->next;
95     while (p) {
96         len++;
97         p = p->next;
98     }
99     return len;
100 } // QueueLength
101
102 /**
103  * 若队列不空，则用e返回Q的队头元素，并返回OK；否则返回ERROR
104  */
105 Status GetHead(LinkQueue Q, QElemType &e)
106 {
107     if (Q.front == Q.rear)
108         return ERROR;

```

```

109     e = Q.front->next->data;
110     return OK;
111 } // GetHead
112
113 /**
114  * 插入元素e为Q的新的队尾元素
115  */
116 Status EnQueue(LinkQueue &Q, QElemType e)
117 {
118     QueuePtr p;
119     p = (QNode *)malloc(sizeof(QNode)); // 生成新的结点
120     if (!p)
121         return ERROR;
122     p->data = e;
123     p->next = NULL;
124     Q.rear->next = p;
125     Q.rear = p;
126     return OK;
127 } // EnQueue
128
129 /**
130  * 若队列不空，删除队列Q的队头元素，用e返回其值，返回OK，否则返回ERROR
131  */
132 Status DeQueue(LinkQueue &Q, QElemType &e)
133 {
134     QueuePtr p;
135     p = Q.front->next;
136     if (Q.front == Q.rear)
137         return ERROR;
138     e = p->data;
139     if (Q.front->next == Q.rear) // 队列中只有一个元素时，还要修改尾指针
140         Q.rear = Q.front;
141     Q.front->next = p->next;
142     free(p);
143     p = NULL;
144     return OK;
145 } // DeQueue
146
147 /**
148  * 从队头到队尾依次对Q中每个元素调用visit()，visit()失败则操作失败
149  */
150 Status QueueTraverse(LinkQueue Q, Status (*visit)(QElemType &e))
151 {
152     QueuePtr p;
153     p = Q.front->next;
154     while (p) {
155         if (!(*visit)(p->data))
156             return ERROR;
157         p = p->next;
158     }
159     return OK;
160 } // QueueTraverse
161
162 /**
163  * 打印元素
164  */
165 Status visit_display(QElemType &e)
166 {

```

```
167     printf("%d ", e);
168     return OK;
169 }
```

顺序栈测试程序：（测试程序代码放在实现代码下方即可）

```
1  int main()
2  {
3      LinkQueue queue1;
4      InitQueue(queue1);
5      int arr1[8] = {33, 12, 75, 0, 49, 67, 8, 999};
6      for (int i = 0; i < sizeof(arr1) / sizeof(int); i++)
7          EnQueue(queue1, arr1[i]);
8
9      printf("链队列queue1中有%d个元素，分别为：\n", QueueLength(queue1));
10     QueueTraverse(queue1, visit_display);
11
12     int head;
13     GetHead(queue1, head);
14     printf("\n第一个元素的值为： %d", head);
15
16     // 删除队头元素
17     DeQueue(queue1, head);
18     GetHead(queue1, head);
19     printf("\n删除队头元素后，第一个元素的值为： %d", head);
20
21     // 在队尾插入元素52，并遍历
22     int tail = 52;
23     EnQueue(queue1, tail);
24     printf("\n插入新元素后的queue1为：\n");
25     QueueTraverse(queue1, visit_display);
26
27     DestroyQueue(queue1);
28
29     return 0;
30 }
```

测试程序运行结果：

链队列queue1中有8个元素, 分别为:

33 12 75 0 49 67 8 999

第一个元素的值为: 33

删除队头元素后, 第一个元素的值为: 12

插入新元素后的queue1为:

12 75 0 49 67 8 999 52

2. 循环队列的基本操作实现

循环队列的实现参照教材64页。

```

1  /**
2   * 循环队列的顺序存储结构及基本操作实现
3   * 参照教材64页
4   * */
5
6  #include <stdio.h>
7  #include <stdlib.h>
8
9  /**
10   * 宏定义及类型定义
11   * */
12  #define ERROR 0
13  #define OK 1
14  #define TRUE 1
15  #define FALSE 0
16  #define MAXQSIZE 100 // 最大队列长度
17
18  typedef int QElemType;
19  typedef int Status;
20
21  /**
22   * 定义循环队列
23   * */
24  typedef struct {
25      QElemType *base; // 初始化的动态分配存储空间
26      int front; // 头指针，若队列不空，指向队列头元素
27      int rear; // 尾指针，若队列不空，指向队列尾元素的下一个位置
28  } CyclicQueue;
29
30  /**
31   * 构造一个空的循环队列Q
32   * */
33  Status InitQueue(CyclicQueue &Q)
34  {
35      Q.base = (QElemType *)malloc(MAXQSIZE * sizeof(QElemType));
36      if (!Q.base)
37          return ERROR;
38      Q.front = 0;
39      Q.rear = 0;
40      return OK;
41  } // InitQueue
42
43  /**
44   * 销毁队列
45   * */
46  Status DestroyQueue(CyclicQueue &Q)
47  {
48      free(Q.base);
49      Q.base = NULL;
50      return OK;
51  } // DestroyQueue
52
53  /**
54   * 清空队列
55   * */
56  Status ClearQueue(CyclicQueue &Q)
57  {
58      Q.front = Q.rear = 0;

```



```

59     return OK;
60 } // ClearQueue
61
62 /**
63  * 判断队列是否为空
64  */
65 Status QueueEmpty(CyclicQueue &Q)
66 {
67     if (Q.rear == Q.front)
68         return TRUE;
69     return FALSE;
70 } // QueueEmpty
71
72 /**
73  * 返回队列Q的元素个数，即队列的长度
74  */
75 Status QueueLength(CyclicQueue Q)
76 {
77     return (Q.rear - Q.front + MAXQSIZE) % MAXQSIZE;
78 } // QueueLength
79
80 /**
81  * 得到队列头元素，用e返回，队列为空返回ERROR，否则返回OK
82  */
83 Status GetHead(CyclicQueue Q, QElemType &e)
84 {
85     if (Q.rear == Q.front)
86         return ERROR;
87     e = *(Q.base + Q.front);
88     return OK;
89 } // GetHead
90
91 /**
92  * 插入元素e为Q的新的队尾元素，队列已满返回ERROR，否则返回OK
93  */
94 Status EnQueue(CyclicQueue &Q, QElemType e)
95 {
96     if ((Q.rear + 1) % MAXQSIZE == Q.front) // 队列满
97         return ERROR;
98     *(Q.base + Q.rear) = e;
99     Q.rear = (Q.rear + 1) % MAXQSIZE;
100     return OK;
101 } // EnQueue
102
103 /**
104  * 若队列不空，则删除队头元素，用e返回其值，并返回OK，否则返回ERROR
105  */
106 Status DeQueue(CyclicQueue &Q, QElemType &e)
107 {
108     if (Q.front == Q.rear)
109         return ERROR;
110     e = *(Q.base + Q.front);
111     Q.front = (Q.front + 1) % MAXQSIZE;
112     return OK;
113 } // DeQueue
114
115 /**
116  * 从队头到队尾遍历Q中每个元素调用visit()，visit()失败则操作失败

```

```

117  */
118  Status QueueTraverse(CyclicQueue Q, Status (*visit)(QElemType &e))
119  {
120      int p;
121      p = Q.front;
122      while (p != Q.rear) {
123          if (!(*visit)(*(&Q.base[p])))
124              return ERROR;
125          p = (p + 1) % MAXQSIZE;
126      }
127      return OK;
128  } // QueueTraverse
129
130  /**
131   * 打印元素
132   */
133  Status visit_display(QElemType &e)
134  {
135      printf("%d ", e);
136      return OK;
137  }

```

Author: XF

Finished time: 2019/11/11