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。

```c
/**
 * 顺序栈的基本操作实现
 * 参照教材46页
 * */

#include <stdio.h>
#include <stdlib.h>

/**
 * 宏定义及类型定义
 * */
#define ERROR 0
#define OK 1
#define TRUE 1
#define FALSE 0
#define STACK_INIT_SIZE 100  // 栈存储空间初始分配量
#define STACKINCREMENT 10    // 栈容量每次增加的值

typedef int SElemType;
typedef int Status;

/**
 * 定义顺序栈
 * */
typedef struct {
    SElemType *base;  // 基指针，在栈构造前和销毁之后，base的值为NULL
    SElemType *top;   // 指向栈顶的指针
    int stacksize;    // 当前已分配的存储空间，以元素为单位
} SqStack;

/**
 * 构造一个空栈S
 */
Status InitStack(SqStack &S)
{
    S.base = (SElemType *)malloc(STACK_INIT_SIZE * sizeof(SElemType));
```

```
    if (!S.base)
        return ERROR;  // 存储分配失败
    S.top = S.base;
    S.stacksize = STACK_INIT_SIZE;
    return OK;
}  // InitStack

/**
 * 销毁栈S，S不再存在
 */
Status DestroyStack(SqStack &S)
{
    free(S.base);
    S.base = NULL;
    return 0;
}  // DestroyStack

/**
 * 把S置为空栈
 */
Status ClearStack(SqStack &S)
{
    S.top = S.base;
    return 0;
}  // ClearStack

/**
 * 若栈S为空，返回TRUE，  否则返回FALSE
 */
Status StackEmpty(SqStack S) { return (S.base == S.top) ? TRUE : FALSE; }

/**
 * 返回S的元素个数，即栈的长度
 */
int StackLength(SqStack S) { return (S.top - S.base); }

/**
 * 若栈不空，则用e返回S的栈顶元素，并返回OK，否则返回ERROR
 */
Status GetTop(SqStack S, SElemType &e)
{
    if (S.base == S.top)  // 栈为空
        return ERROR;
    e = *(S.top - 1);  // 栈顶元素赋给e
    return OK;
}  // GetTop

/**
 * 若栈不空，返回S的栈顶元素
 */
int GetTop(SqStack S) { return *(S.top - 1); }

/**
 * 插入元素e为新的栈顶元素
 */
Status Push(SqStack &S, SElemType e)
{
    if (S.stacksize == StackLength(S)) {  // 栈达到了最大容量，自动增加容量
```

```
        S.base = (SElemType *)realloc(
            S.base, (S.stacksize + STACKINCREMENT) * sizeof(SElemType));
        if (!S.base)
            exit(OVERFLOW);  // 存储分配失败
        S.top = S.base + S.stacksize;
        S.stacksize += STACKINCREMENT;
    }
    *S.top++ = e;
    return OK;
}  // Push

/**
 * 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
 */
Status Pop(SqStack &S, SElemType &e)
{
    if (S.top == S.base)  // 栈为空
        return ERROR;
    e = *(--S.top);
    return OK;
}  // Pop

/**
 * 从栈底到栈顶每个元素依次调用visit()，且visit失败，则操作失败
 */
Status StackTraverse(SqStack S, Status (*visit)(SElemType))
{
    SElemType *p;
    p = S.base;            // p指向栈底
    while (p != S.top) {  // 遍历栈的元素
        visit(*p);
        p++;
    }
    return OK;
}  // StackTraverse

/**
 * 打印元素
 */
Status visit_display(SElemType e)
{
    printf("%d ", e);
    return OK;
}

Status display_char(SElemType e)
{
    printf("%c ", e);
    return OK;
}
```

顺序栈测试程序：（测试程序代码放在实现代码下方即可）

```
int main()
{
```

```
    SqStack stack1;
    InitStack(stack1);
    int arr1[8] = {33, 12, 75, 0, 49, 67, 8, 999};
    int arr2[3] = {24, 81, 100};
    for (int i = 0; i < sizeof(arr1) / sizeof(int); i++)
        Push(stack1, arr1[i]);

    printf("顺序栈stack1中有%d个元素，这%d个元素为: \n", StackLength(stack1),
           StackLength(stack1));
    StackTraverse(stack1, visit_display);

    // 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
    int a1 = 0;
    Pop(stack1, a1);
    for (int i = 0; i < sizeof(arr2) / sizeof(int); i++)
        Push(stack1, arr2[i]);
    printf("\nstack1更新后有%d个元素\n其中栈顶元素为: %d", StackLength(stack1),
           GetTop(stack1));

    DestroyStack(stack1);

    return 0;
}
```

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。

```
/**
 * 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
 */
Status checkBracket(char *expr)
{
    int i = 0;
    SqStack S;
    InitStack(S);
    SElemType e;
    while ('\0' != *(expr + i)) {
        switch (*(expr + i)) {
            case '(':
                Push(S, -1);
                break;
            case '[':
```

```
                Push(S, -2);
                break;
            case '{':
                Push(S, -3);
                break;
            case ')':
                if (OK == GetTop(S, e)) {  // 栈非空，得到栈顶元素
                    if (-1 == e)           // 栈顶元素与‘）’匹配
                        Pop(S, e);         // 删除栈顶元素
                    else                   // 不匹配，返回ERROR
                        return ERROR;
                } else
                    return ERROR;
                break;
            case ']':
                if (OK == GetTop(S, e)) {
                    if (-2 == e)
                        Pop(S, e);
                    else
                        return ERROR;
                } else
                    return ERROR;
                break;
            case '}':
                if (OK == GetTop(S, e)) {
                    if (-3 == e)
                        Pop(S, e);
                    else
                        return ERROR;
                } else
                    return ERROR;
                break;
            default:
                break;
        }  // switch

        i++;
    }  // while

    if (StackEmpty(S))
        return OK;
    else
        return ERROR;
}

/**
 * 输出括号匹配的检验结果
 */
void outputResult(Status i, char *k)
{
    OK == i ? printf("表达式 \"%s\" 左右括号数匹配。\n", k)
            : printf("表达式 \"%s\" 左右括号数不匹配。\n", k);
}

// 测试程序
int main()
{
```

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

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。

```
/**
 * 迷宫问题求解算法的实现
 * 部分内容参照教材51页
 */

#include <stdio.h>
#include <stdlib.h>

#define ERROR 0
#define OK 1
#define TRUE 1
#define FALSE 0
#define OVERFLOW -1
#define STACK_INIT_SIZE 100  // 栈初始化值
#define STACKINCREMENT 10    // 栈容量每次增加的值
#define SIZE 10              // 定义迷宫地图大小

typedef struct {
    int row;  // 行，0起始
    int col;  // 列，0起始
} PosType;

typedef struct {
    int ord;       // 通道块在路径上的序号
    PosType seat;  // 通道块在迷宫中的坐标位置
    int di;        // 从此通道走向下一通道块的方向
} SElemType;

typedef int Status;

/**
 * 定义顺序栈
 * */
typedef struct {
    SElemType *base;  // 栈底指针，在栈构造前和销毁之后，base的值为NULL
    SElemType *top;  // 栈顶指针
    int stacksize;   // 当前已分配的存储空间，以元素为单位
```

```
} SqStack;

/**
 * 构造一个空栈S
 */
Status InitStack(SqStack &S)
{
    S.base = (SElemType *)malloc(STACK_INIT_SIZE * sizeof(SElemType));
    if (!S.base)
        return ERROR;  // 存储分配失败
    S.top = S.base;
    S.stacksize = STACK_INIT_SIZE;
    return OK;
}  // InitStack

/**
 * 销毁栈S，S不再存在
 */
Status DestroyStack(SqStack &S)
{
    free(S.base);
    S.base = NULL;
    return 0;
}  // DestroyStack

/**
 * 把S置为空栈
 */
Status ClearStack(SqStack &S)
{
    S.top = S.base;
    return 0;
}  // ClearStack

/**
 * 若栈S为空，返回TRUE， 否则返回FALSE
 */
Status StackEmpty(SqStack S) { return (S.base == S.top) ? TRUE :
FALSE; }

/**
 * 返回S的元素个数，即栈的长度
 */
int StackLength(SqStack S) { return (S.top - S.base); }

/**
 * 若栈不空，则用e返回S的栈顶元素，并返回OK，否则返回ERROR
 */
Status GetTop(SqStack S, SElemType &e)
{
    if (S.base == S.top)  // 栈为空
        return ERROR;
    e = *(S.top - 1);  // 栈顶元素赋给e
    return OK;
}  // GetTop

/**
 * 插入元素e为新的栈顶元素
```

```
 */
Status Push(SqStack &S, SElemType e)
{
    if (S.stacksize == StackLength(S)) {  // 栈达到了最大容量，自动增加容量
        S.base = (SElemType *)realloc(
            S.base, (S.stacksize + STACKINCREMENT) *
sizeof(SElemType));
        if (!S.base)
            exit(OVERFLOW);  // 存储分配失败
        S.top = S.base + S.stacksize;
        S.stacksize += STACKINCREMENT;
    }
    *S.top++ = e;
    return OK;
}  // Push

/**
 * 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
 */
Status Pop(SqStack &S, SElemType &e)
{
    if (S.top == S.base)  // 栈为空
        return ERROR;
    e = *(--S.top);
    return OK;
}  // Pop

/**
 * 从栈底到栈顶每个元素依次调用visit()，旦visit失败，则操作失败
 */
Status StackTraverse(SqStack S, Status (*visit)(SElemType))
{
    SElemType *p;
    p = S.base;            // p指向栈底
    while (p != S.top) {  // 遍历栈的元素
        visit(*p);
        p++;
    }
    return OK;
}  // StackTraverse

/**
 * 打印元素
 */
Status visit_display(SElemType e)
{
    printf("%d ", e);
    return OK;
}

Status display_char(SElemType e)
{
    printf("%c ", e);
    return OK;
}

extern SqStack S;
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;
```

```
            Push(S, e);  // 加入路径
            if (curpos.row == end.row && curpos.col == end.col)
                return (OK);                // 到达终点
            curpos = NextPos(curpos, 1);  // 将当前位置的东邻设为下一位置
            curstep++;                      // 探索下一步
        } else {                            // 如果当前位置不能通过
            if (!StackEmpty(S)) {
                Pop(S, e);
                while (4 == e.di && !StackEmpty(S))
                    Pop(S, e);  // 即该位置的4方向都已经探索完成，就再退回一步
                if (e.di < 4) {  // 换一个方向探索
                    e.di++;
                    Push(S, e);
                    curpos = NextPos(e.seat, e.di);
                }
            }
        }
    } while (!StackEmpty(S));

    return ERROR;
}

Status display_path(SElemType e)
{
    printf("(%d, %d) ", e.seat.row, e.seat.col);
    switch (e.di) {
        case 1:
            printf("东\n");
            break;
        case 2:
            printf("南\n");
            break;
        case 3:
            printf("西\n");
            break;
        case 4:
            printf("北\n");
            break;
    }
    return OK;
}

// 测试程序
int Map[SIZE][SIZE] = {{1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
                       {1, 0, 0, 1, 0, 0, 0, 1, 0, 1},
                       {1, 0, 0, 1, 0, 0, 0, 1, 0, 1},
                       {1, 0, 0, 0, 0, 1, 1, 0, 0, 1},
                       {1, 0, 1, 1, 1, 0, 0, 0, 0, 1},
                       {1, 0, 0, 0, 1, 0, 0, 0, 0, 1},
                       {1, 0, 1, 0, 0, 0, 1, 0, 0, 1},
                       {1, 0, 1, 1, 1, 0, 1, 1, 0, 1},
                       {1, 1, 0, 0, 0, 0, 0, 0, 0, 1},
                       {1, 1, 1, 1, 1, 1, 1, 1, 1, 1}};  // 迷宫地图
SqStack S;                                               // 路径栈

int main()
{
```

```
    InitStack(S);
    PosType start;
    PosType end;
    start.row = 1;
    start.col = 1;

    end.row = 8;
    end.col = 8;

    MazePath(start, end);
    StackTraverse(S, display_path);
    return 0;
}
```

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。

```
/**
 * 链队列的基本操作实现
 * 参照教材61页
 * */

#include <stdio.h>
#include <stdlib.h>

/**
 * 宏定义及类型定义
 * */
#define ERROR 0
#define OK 1
#define TRUE 1
#define FALSE 0

typedef int QElemType;
typedef int Status;

typedef struct QNode {
    QElemType data;
    struct QNode *next;
} QNode, *QueuePtr;

/**
 * 定义链队列
 * */
typedef struct {
    QueuePtr front;  // 队头指针
    QueuePtr rear;   // 队尾指针
} LinkQueue;

/**
 * 构造一个空队列Q
 */
Status InitQueue(LinkQueue &Q)
{
    Q.front = (QNode *)malloc(sizeof(QNode));
    if (!Q.front)
        return ERROR;
    Q.front->next = NULL;
    Q.rear = Q.front;
    return OK;
}  // InitQueue

/**
 * 销毁队列Q，Q不再存在
 */
Status DestroyQueue(LinkQueue &Q)
{
```

```
    while (Q.front) {
        Q.rear = Q.front->next;
        free(Q.front);
        Q.front = Q.rear;
    }
    return OK;
}  // DestroyQueue

/**
 * 将Q清为空队列，并释放结点空间
 */
Status ClearQueue(LinkQueue &Q)
{
    QueuePtr p, temp;
    p = Q.front->next;
    Q.front->next = NULL;
    Q.rear = Q.front;
    while (p) {
        temp = p->next;
        free(p);
        p = temp;
    }
    return OK;
}  // ClearQueue

/**
 * 判断Q是否为空队列，是返回TRUE，否则返回FALSE
 */
Status QueueEmpty(LinkQueue Q)
{
    if (Q.front == Q.rear)
        return TRUE;
    else
        return FALSE;
}  // QueueEmpty

/**
 * 返回Q的元素个数，即队列的长度
 */
int QueueLength(LinkQueue Q)
{
    QueuePtr p;
    int len = 0;
    p = Q.front->next;
    while (p) {
        len++;
        p = p->next;
    }
    return len;
}  // QueueLength

/**
 * 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
 */
Status GetHead(LinkQueue Q, QElemType &e)
{
    if (Q.front == Q.rear)
        return ERROR;
```

```
    e = Q.front->next->data;
    return OK;
}  // GetHead

/**
 * 插入元素e为Q的新的队尾元素
 */
Status EnQueue(LinkQueue &Q, QElemType e)
{
    QueuePtr p;
    p = (QNode *)malloc(sizeof(QNode));  // 生成新的结点
    if (!p)
        return ERROR;
    p->data = e;
    p->next = NULL;
    Q.rear->next = p;
    Q.rear = p;
    return OK;
}  // EnQueue

/**
 * 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
 */
Status DeQueue(LinkQueue &Q, QElemType &e)
{
    QueuePtr p;
    p = Q.front->next;
    if (Q.front == Q.rear)
        return ERROR;
    e = p->data;
    if (Q.front->next == Q.rear)  // 队列中只有一个元素时，还要修改尾指针
        Q.rear = Q.front;
    Q.front->next = p->next;
    free(p);
    p = NULL;
    return OK;
}  // DeQueue

/**
 * 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
 */
Status QueueTraverse(LinkQueue Q, Status (*visit)(QElemType &e))
{
    QueuePtr p;
    p = Q.front->next;
    while (p) {
        if (!(*visit)(p->data))
            return ERROR;
        p = p->next;
    }
    return OK;
}  // QueueTraverse

/**
 * 打印元素
 */
Status visit_display(QElemType &e)
{
```

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

顺序栈测试程序：（测试程序代码放在实现代码下方即可）

```
int main()
{
    LinkQueue queue1;
    InitQueue(queue1);
    int arr1[8] = {33, 12, 75, 0, 49, 67, 8, 999};
    for (int i = 0; i < sizeof(arr1) / sizeof(int); i++)
        EnQueue(queue1, arr1[i]);

    printf("链队列queue1中有%d个元素, 分别为: \n", QueueLength(queue1));
    QueueTraverse(queue1, visit_display);

    int head;
    GetHead(queue1, head);
    printf("\n第一个元素的值为: %d", head);

    // 删除队头元素
    DeQueue(queue1, head);
    GetHead(queue1, head);
    printf("\n删除队头元素后, 第一个元素的值为: %d", head);

    // 在队尾插入元素52, 并遍历
    int tail = 52;
    EnQueue(queue1, tail);
    printf("\n插入新元素后的queue1为: \n");
    QueueTraverse(queue1, visit_display);

    DestroyQueue(queue1);

    return 0;
}
```

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。

```c
/**
 * 循环队列的顺序存储结构及基本操作实现
 * 参照教材64页
 * */

#include <stdio.h>
#include <stdlib.h>

/**
 * 宏定义及类型定义
 * */
#define ERROR 0
#define OK 1
#define TRUE 1
#define FALSE 0
#define MAXQSIZE 100  // 最大队列长度

typedef int QElemType;
typedef int Status;

/**
 * 定义循环队列
 * */
typedef struct {
    QElemType *base;  // 初始化的动态分配存储空间
    int front;        // 头指针，若队列不空，指向队列头元素
    int rear;  // 尾指针，若队列不空，指向队列尾元素的下一个位置
} CyclicQueue;

/**
 * 构造一个空的循环队列Q
 */
Status InitQueue(CyclicQueue &Q)
{
    Q.base = (QElemType *)malloc(MAXQSIZE * sizeof(QElemType));
    if (!Q.base)
        return ERROR;
    Q.front = 0;
    Q.rear = 0;
    return OK;
}  // InitQueue

/**
 * 销毁队列
 */
Status DestroyQueue(CyclicQueue &Q)
{
    free(Q.base);
    Q.base = NULL;
    return OK;
}  // DestroyQueue

/**
 * 清空队列
 */
Status ClearQueue(CyclicQueue &Q)
{
    Q.front = Q.rear = 0;
```

```
    return OK;
}  // ClearQueue

/**
 * 判断队列是否为空
 */
Status QueueEmpty(CyclicQueue &Q)
{
    if (Q.rear == Q.front)
        return TRUE;
    return FALSE;
}  // QueueEmpty

/**
 * 返回队列Q的元素个数，  即队列的长度
 */
Status QueueLength(CyclicQueue Q)
{
    return (Q.rear - Q.front + MAXQSIZE) % MAXQSIZE;
}  // QueueLength

/**
 * 得到队列头元素，用e返回，队列为空返回ERROR，否则返回OK
 */
Status GetHead(CyclicQueue Q, QElemType &e)
{
    if (Q.rear == Q.front)
        return ERROR;
    e = *(Q.base + Q.front);
    return OK;
}  // GetHead

/**
 * 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
 */
Status EnQueue(CyclicQueue &Q, QElemType e)
{
    if ((Q.rear + 1) % MAXQSIZE == Q.front)  // 队列满
        return ERROR;
    *(Q.base + Q.rear) = e;
    Q.rear = (Q.rear + 1) % MAXQSIZE;
    return OK;
}  // EnQueue

/**
 * 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
 */
Status DeQueue(CyclicQueue &Q, QElemType &e)
{
    if (Q.front == Q.rear)
        return ERROR;
    e = *(Q.base + Q.front);
    Q.front = (Q.front + 1) % MAXQSIZE;
    return OK;
}  // DeQueue

/**
 * 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
```

```
 */
Status QueueTraverse(CyclicQueue Q, Status (*visit)(QElemType &e))
{
    int p;
    p = Q.front;
    while (p != Q.rear) {
        if (!(*visit)(*(Q.base + p)))
            return ERROR;
        p = (p + 1) % MAXQSIZE;
    }
    return OK;
}  // QueueTraverse

/**
 * 打印元素
 */
Status visit_display(QElemType &e)
{
    printf("%d ", e);
    return OK;
}
```

Author: XF

Finished time: 2019/11/11