廣東工業大學

设 计 性 实 验

课程名称　　数据结构

题目名称　　栈和队列

学生学院　　计算机学院

专业班级　　计算机科学与技术 1901

学　　号　　3119004760

学生姓名　　叶嘉轩

指导教师　　李杨

2020 年 12 月 23 日

1.0 **题目**

采用字符型为元素类型和顺序存储结构以及链式存储结构来实现抽象数据类型栈和队列

使用的软件环境和工具：Visual Studio Code（mac 电脑）

使用的语言：C

数据对象：$D = \{ a_i \mid a_i \in ElemSet, i=1,2,...,n, \ n \geqslant 0 \}$

数据关系：$R1 = \{ <ai-1, ai> | ai-1, ai \in D, i=2,...,n \}$

基本操作：

顺序栈：

InitStack_Sq(SqStack *S,int size,int inc)

操作结果：构造一个空的顺序栈。

Push_Sq(SqStack *S, char e)

初始条件：顺序栈 S 已存在。

操作结果：元素 e 入栈。

DestroyStack_Sq(SqStack *s)

初始条件：顺序栈 s 已存在。

操作结果：销毁顺序栈 s。

StackEmpty(SqStack *s)

初始条件：顺序栈 s 已存在。

操作结果：判断顺序栈是否为空，若为空则返回 1，否则返回 0

ClearStack_Sq(SqStack *s)

初始条件：顺序栈 s 已存在。

操作结果：清空顺序栈 s，清空成功则返回 1。

Pop_Sq(SqStack *s)

初始条件：顺序栈 s 已存在。

操作结果：返回栈顶元素，则栈为空则返回 0。

GetTop_Sq(SqStack *s)

初始条件：顺序栈 s 已存在。

操作结果：取出栈顶元素。

循环队列：

InitQueue_Sq(SqQueue *Q,int size)

操作结果：构造一个空的循环队列，构造成功则返回 1，否则返回 0。

DestroyQueue_Sq(SqQueue *Q)

初始条件：循环队列 Q 已存在。

操作结果：销毁循环队列 Q。

ClearQueue_Sq(SqQueue *Q)

初始条件：循环队列 Q 已存在。

操作结果：清空循环队列 Q。

QueueEmpty_Sq(SqQueue *Q)

初始条件：循环队列 Q 已存在。

操作结果：判断循环队列 Q 是否为空，空的话返回 1，否则返回 0。

QueueLength_Sq(SqQueue *Q)

初始条件：循环队列 Q 已存在。

操作结果：返回循环队列 Q 的长度。

GetHead_Sq(SqQueue *Q)

初始条件：循环队列 Q 已存在。

操作结果：返回循环队列 Q 的队头元素，队列为空则返回‘0’。

EnQueue_Sq(SqQueue *Q,char a)

初始条件：循环队列 Q 已存在。

操作结果：循环队列队尾插入元素 a，插入成功则返回 1，否则返回 0。

DeQueue_Sq(SqQueue *Q)

初始条件：循环队列 Q 已存在。

操作结果：删除循环队列 Q 的头元素，删除失败则返回‘0’。

链栈：

InitStack_LS()

操作结果：构造一个空的链栈。

ClearStack_LS(LStack s)

初始条件：链栈 s 已存在。

操作结果：清空链栈 s。

StackEmpty_LS(LStack s)

初始条件：链栈 s 已存在。

操作结果：判断链栈 s 是否为空，空的话返回 1，否则返回 0。

DestroyStack_LS(LStack s)

初始条件：链栈 s 已存在。

操作结果：销毁链栈 s。

Puah_LS(LStack s,char e)

初始条件：链栈 s 已存在。

操作结果：将元素 e 压入链栈 s。

Pop_LS(LStack s)

初始条件：链栈 s 已存在。

操作结果：将链栈栈顶元素出栈。

GetTop_LS(LStack s)

初始条件：链栈 s 已存在。

操作结果：返回栈顶元素，若栈为空则返回‘0’。

链队列：

InitQueue_LQ(LQueue *q)

操作结果：构造一个空的链队列。

QueueEmpty_LQ(LQueue *q)

初始条件：链队列 q 已存在。

操作结果：判断链队列 q 是否为空，空则返回 1，否则返回 0。

DestroyQueue_LQ(LQueue *q)

初始条件：链队列 q 已存在。

操作结果：销毁链队列 q。

QueueLength_LQ(LQueue *q)

初始条件：链队列 q 已存在。

操作结果：返回链队列的长度。

GetHead_LQ(LQueue *q)

初始条件：链队列 q 已存在。

操作结果：返回链队列的队头元素，若不存在则返回‘0’。

EnQueue_LQ(LQueue *q,char e)

初始条件：链队列 q 已存在。

操作结果：链队列 q 尾插入元素 e。

DeQueue_LQ(LQueue *q)

初始条件：链队列 q 已存在。

操作结果：删除链队列 s 的队头元素。

2.0 **存储结构定义**

公用头文件 hello.c：

(1) 顺序栈存储结构

```
typedef struct
    {
        char *elem；
         int top；
        int size；
        int increment；
    }SqStack；
```

(2) 循环队列存储结构

```
typedef struct
    {
        char *elem；
        int front；
         int rear；
        int maxSize；
    }SqQueue；
```

(3) 链栈存储结构

```
typedef struct LSNode
```

```
        {
                char data;
                struct LSNode *next;
        }LSNode,*LStack;
```

(4)链队列存储结构

```
    typedef struct LQNode
        {
                char data;
                struct    LQNode *next;
        }LQNode, *QueuePtr;
    typedef struct
        {
              QueuePtr front;
             QueuePtr rear;
        }LQueue;
```

3.0 **算法设计**

（1）顺序栈存储结构

#pragma mark - 顺序栈

```
typedef struct
{
char *elem;
```

```
int top;
int size;
int increment;

}SqStack;
//顺序栈的初始化
int InitStack_Sq(SqStack *S,int size,int inc)
{
S->elem = (char *)malloc(size *sizeof(char));
if (S->elem == NULL)
{
return 0;
}
S->top = 0;
S->size = size;
S->increment = inc;
return 1;
}
//顺序栈入栈
int Push_Sq(SqStack *S, char e)
{
char *temp;
```

```
if (S->top>S->size)
{
temp = (char *)realloc(S->elem,(S->size+S->increment)*sizeof(char));
if (temp == NULL)
{
return 0;
}
S->elem = temp;
S->size += S->increment;
}
S->elem[S->top++] = e;
return 1;
}
//销毁顺序栈
int DestroyStack_Sq(SqStack *s)
{
free(s->elem);
s->elem = NULL;
s->size = 0;
s->top = 0;
s->increment = 0;
return 1;
```

```
}
//判断栈是否为空
int StackEmpty(SqStack *s)
{
if (s->top == 0)
{
return 1；
}
return 0；
}
//清空栈
int ClearStack_Sq(SqStack *s)
{
s->top = 0；
return 1；
}
//将栈顶元素出栈
char Pop_Sq(SqStack *s)
{
if (s->top == 0)
{
return 0；
```

```
}
s->top = s->top-1；
return s->elem[s->top]；
}
//取出栈顶元素
char GetTop_Sq(SqStack *s)
{
if (s->top == 0)
{
return 'n'；
}
return s->elem[s->top-1]；
}
#pragma mark - 循环队列
typedef struct
{
char *elem；
int front；
int rear；
int maxSize；
}SqQueue；
//循环队列初始化
```

```
int InitQueue_Sq(SqQueue *Q,int size)
{
Q->elem = (char *)malloc(size * sizeof(char));
if (Q->elem == NULL)
{
return 0;
}
Q->maxSize = size;
Q->front = Q->rear = 0;
return 1;
}
//销毁队列
void DestroyQueue_Sq(SqQueue *Q)
{
free(Q->elem);
Q->elem = NULL;
Q->front = Q->rear = 0;
Q->maxSize = 0;
}
//将队列置为空
void ClearQueue_Sq(SqQueue *Q)
{
```

```
Q->front = Q->rear = 0；
}
//判断队列是否为空队列
int QueueEmpty_Sq(SqQueue *Q)
{
if (Q->front == Q->rear)
{
return 1；
}
return 0；
}
//返回队列元素个数
int QueueLength_Sq(SqQueue *Q)
{
return (Q->rear - Q->front + Q->maxSize) % Q->maxSize；
}
//返回队列头元素
char GetHead_Sq(SqQueue *Q)
{
if (Q->front == Q->rear)
{
return '0'；
```

```
}
return Q->elem[Q->front];
}
//队尾插入元素
int EnQueue_Sq(SqQueue *Q,char a)
{
if ((Q->rear+1)%Q->maxSize == Q->front)
{
return 0;
}
Q->elem[Q->rear] = a;
Q->rear = (Q->rear+1)%Q->maxSize;
return 1;
}
//删除队列的头元素
char DeQueue_Sq(SqQueue *Q)
{
if (Q->front == Q->rear)
{
return '0';
}
int temp = Q->front;
```

```
Q->front = (Q->front+1)%Q->maxSize；
return Q->elem[temp]；
}
```

（2）链式存储结构

```
#pragma mark - 链栈
typedef struct LSNode
{
char data；
struct LSNode *next；
}LSNode,*LStack；
//初始化链栈
LStack InitStack_LS()
{
LStack s = (LStack)malloc(sizeof(LSNode))；
if (s==NULL)
{
printf("初始化失败\n")；
}else
{
s->next = NULL；
printf("初始化成功\n")；
```

```
return s;
}
return s;
}
//清空栈
void ClearStack_LS(LStack s)
{
LStack t = s->next;
while (t)
{
s->next = t->next;
free(t);
t->next = NULL;
t = s->next;
}
}
//判断空栈
int StackEmpty_LS(LStack s)
{
if (s->next == NULL)
{
return 1;
```

```
}
return 0;
}
//销毁链栈
void DestroyStack_LS(LStack s)
{
if (StackEmpty_LS(s) == 1)
{
free(s);
s->next = NULL;
}else
{
ClearStack_LS(s);
free(s);
s->next = NULL;
}
}
//元素入栈
void Puah_LS(LStack s,char e)
{
if (s != NULL)
{
```

```
LStack t = (LStack)malloc(sizeof(LSNode));
t->data = e;
t->next = s->next;
s->next = t;
}
}
//栈顶元素出栈
char Pop_LS(LStack s)
{
char e;
if (s->next)
{
LStack l = s->next;
e = l->data;
s->next = l->next;
}
return e;
}
//取栈顶元素
char GetTop_LS(LStack s)
{
if (s->next)
```

```
{
return s->next->data；
}
return '0'；
}

#pragma mark - 链队列
typedef struct LQNode
{
char data；
struct LQNode *next；
}LQNode, *QueuePtr；

typedef struct
{
QueuePtr front；
QueuePtr rear；
}LQueue；
//初始化空队列
void InitQueue_LQ(LQueue *q)
{
q->front = q->rear = (QueuePtr)malloc(sizeof(LQNode))；
```

```
q->front = NULL；
}
//队列判空
int QueueEmpty_LQ(LQueue *q)
{
if (q->front == NULL)
{
return 1；
}else
{
return 0；
}
}
//销毁队列
void DestroyQueue_LQ(LQueue *q)
{
if (QueueEmpty_LQ(q) == 0)
{
if (q->front == q->rear)
{
free(q);
q->front = q->rear = NULL；
```

```
}else
{
while (q->front != NULL)
{
free(q->front);
q->front = q->front->next;
}
}
}
}
//队列长度
int QueueLength_LQ(LQueue *q)
{
int k=0;
QueuePtr p;
p = (QueuePtr)malloc(sizeof(LQNode));
p = q->front;
if (p == NULL)
{
return 0;
}
while (p != NULL)
```

```
{
k++;
p = p->next;
}
return k;
}
//返队列头元素
char GetHead_LQ(LQueue *q)
{
if (q->front != NULL)
{
return q->front->data;
}
return '0';
}
//队尾插入元素
void EnQueue_LQ(LQueue *q,char e)
{
LQNode *p;
p =(LQNode *)malloc(sizeof(LQNode));
if (p != NULL)
{
```

```
p->data = e;
p->next = NULL;
if (q->front == NULL)
{
q->front = p;
}else
{
q->rear->next = p;
}
q->rear = p;
}
}
//删除队头元素
void DeQueue_LQ(LQueue *q)
{
LQNode *p;
char e;
if (q->front != NULL)
{
p = q->front;
e = p->data;
q->front = p->next;
```

```
if (q->rear == p)
{
q->rear = NULL;
}
free(p);
}
}
```

4.0 **测试**

测试文件 hello.c

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

int main() {

#pragma 顺序栈
SqStack *s;
//初始化栈
InitStack_Sq(s,10,6);

//元素压入栈
```

```
Push_Sq(s,'a');
Push_Sq(s,'c');
printf("%c 压入栈\n", s->elem[0]);
printf("%c 压入栈\n", s->elem[1]);
printf("栈顶位标%d\n",s->top);

//判断栈空
if (StackEmpty(s) == 1)
{
printf("栈为空\n");
}else {
printf("栈不为空\n");
}

//栈顶元素出栈
printf("栈顶元素%c 出栈\n",Pop_Sq(s));


//取出栈顶元素
printf("取出栈顶元素%c\n",GetTop_Sq(s));

//清空栈
```

```
if (ClearStack_Sq(s) == 1)
{
printf("清空栈，当前栈顶位标为%d\n",s->top);
}

//销毁栈
if (DestroyStack_Sq(s) == 1)
{
printf("销毁栈\n");
}

//判断栈空
if (StackEmpty(s) == 1)
{
printf("栈为空\n");
}else {
printf("栈不为空\n");
}
```

输出结果：

```
[Running] cd "/Users/macbookpro/Desktop/c配置/" && gcc hello.c -o hello && "/Users/macbookpro/Desktop/c配置/"hello
a压入栈
c压入栈
栈顶位标2
栈不为空
栈顶元素c出栈
取出栈顶元素a
清空栈，当前栈顶位标为0
销毁栈
栈为空
```

#pragma 循环队列

```
//初始化循环队列
SqQueue *Q;
InitQueue_Sq(Q,10);

//队尾插入元素
EnQueue_Sq(Q,'a');
EnQueue_Sq(Q,'c');
printf("队尾插入元素%c\n",Q->elem[0]);
printf("队尾插入元素%c\n",Q->elem[1]);
//返回队列元素个数
printf("队列元素为%d 个\n",QueueLength_Sq(Q));
//返回队列头元素
printf("队头元素为%c\n",GetHead_Sq(Q));

//判断队列是否为空
if (QueueEmpty_Sq(Q) == 1)
{
```

```
printf("队列为空\n");
}else
{
printf("队列不为空\n");
}
//删除队列的头元素
printf("删除队头元素%c\n",DeQueue_Sq(Q));
printf("删除队头元素%c\n",GetHead_Sq(Q));
//将队列置为空
ClearQueue_Sq(Q);
if (QueueEmpty_Sq(Q) == 1)
{
printf("队列为空\n");
}else
{
printf("队列不为空\n");
}

//销毁队列
DestroyQueue_Sq(Q);
```

输出结果:

#pragma 链栈

```
//初始化链栈
LStack I = InitStack_LS();


//链栈判空
if (StackEmpty_LS(I) == 1)
{
printf("栈为空\n");
}else
{
printf("栈不空\n");
}


//元素入栈
Puah_LS(I,'a');
Puah_LS(I,'c');
printf("元素 ac 入栈\n");
//取栈顶元素
```

```
printf("取栈顶元素%c\n",GetTop_LS(l));

//栈顶元素出栈

printf("栈顶元素%c 出栈\n",Pop_LS(l));


//取栈顶元素

printf("再取栈顶元素%c\n",GetTop_LS(l));


//清空栈

ClearStack_LS(l);


//销毁栈

DestroyStack_LS(l);
```

输出结果：

#pragma 链队列

```
//初始化链队列

LQueue *q;

InitQueue_LQ(q);
```

```
//队尾插入元素
EnQueue_LQ(q,'a');
EnQueue_LQ(q,'c');
printf("插入元素 ac\n");

//返队列头元素
printf("队头元素%c\n",GetHead_LQ(q));

//队列判空
if (QueueEmpty_LQ(q) == 1)
{
printf("队列为空\n");
}else
{
printf("队列不为空\n");
}

//删除队头元素
DeQueue_LQ(q);
printf("删除队头元素%c\n",GetHead_LQ(q));
//队列长度
```

```
printf("队长%d\n",QueueLength_LQ(q));


//销毁队列
DestroyQueue_LQ(q);
return 0;
}
```

输出结果：

```
[Running] cd "/Users/macbookpro/Desktop/c配置/" && gcc hello.c -o hello && "/Users/macbookpro/Desktop/c配置/"hello
插入元素ac
队头元素a
队列不为空
删除队头元素c
队长1

[Done] exited with code=0 in 0.362 seconds
```

5.0 实验总结和体会

顺序栈和链栈：（1）存储结构方面

1.顺序栈的空间是静态分配的，固定大小的。

2.链栈的空间是动态分配的，容量可变，节省空间。

（2）使用方面

1.顺序栈的查询速度更快。

2.链栈的增删数据操作更快。

循环队列和链队列：（1）存储结构方面

1.循环队列要有一个固定的长度，可能会存在空间资源浪费的问题，除非是在已确定队列长度的情况下。

2.链队列的空间是动态分配的，可用指针来连接新分配的空间

3.相对于循环队列，链队列在地址存储的开销更大。