第三章 栈与队列

刘亮亮

2020 年 3 月 16 日

- 要点：本章主要介绍栈和队列，栈和队列也是线性结构，是操作受限的线性表，在计算机中的比较多的数据结构。主要内容包括：栈和队列的抽象数据类型、栈的表示与实现及应用举例、队列的表示与实现及应用。
- 重点：1.栈的进展与出栈算法；2.队列的入队与出队算法;
- 难点：1.栈的应用

栈与队列也是常用的线性结构，在计算机里面用的比较多的两种数据结构。栈和队列可以认为是操作受限的线性表，只允许在一端进行插入和删除。

1 栈的定义与实现

1.1 栈的定义

不同于线性表，栈(Stack)只能在一端进行插入和删除的线性表。由于一端固定，只能在一端进行插入和删除，先进入的元素后出来（或者后进去的先出来），因此，栈是一种先进后出(FILO,First In Last Out)或后进先出（LIFO,Last In First Out）的线性表。

先看栈的几个重要的概念：

定义 1. 栈底：固定的一端，该端不能进行删除和插入元素，相当于线性表的表头。

通常，我们设置一个栈底指针(bottom)指向栈底，处于栈底的元素称为栈底元素。

定义 2. 栈顶：能插入和删除的一端，相当于线性表的表尾。

处于栈顶的元素称为栈顶元素。一般地，设置一个栈顶指针(top)指向栈顶元素。

定义 3. 空栈：栈的长度为空。

在现实生活中，有很多例子是栈的结构，比如说火车调度站、弹夹等等都是栈的结构，一端是固定的，插入和删除只在一端进行，比如说弹夹，压入子弹都是从一端进行压入，后面压入的子弹开枪的时候先出来，等等，可以想想自己身边有什么结构是栈的结构。

定义 4. 进栈*(push)*：在栈的栈顶插入元素，也叫入栈。

定义 5. 出栈*(pop)*：在栈的栈顶删除元素，也叫退栈。

栈的抽象数据类型定义如下：

ADT Stack{

数据对象:$D = \{a_i | a_i \in DataType\}$

数据关系:$R = \{< a_i, a_{i+1} >\}$

基本操作：

$initStack(\&S)$:

操作结果：建立空栈S，创建成功返回true,并且用S返回，否则返回false

$destroyStack(\&S)$:

初始条件：栈S存在

操作结果：删除一个栈S,删除成功返回true,否则返回false

$clearStack(\&S)$:

初始条件：栈S存在

操作结果：清空栈S，清空成功返回true,否则返回false

$isEmptyStack(\&S)$:

初始条件：栈S存在

操作结果：栈S为空，返回true,否则返回false

$getLength(\&S)$:

初始条件：栈S存在

操作结果：返回栈S的长度

$getTop(\&S, \&e)$:

初始条件：S存在

操作结果：返回栈S的栈顶元素

$push(\&S, e)$:

初始条件：栈S存在

操作结果：在栈S 中插入元素e

$pop(\&S, \&e)$:

初始条件：栈S存在

操作结果：删除栈S的栈顶元素，用e返回

}End Stack.

1.2 栈的顺序存储结构与实现

栈的顺序存储结构称为顺序栈，开辟一组地址连续的存储单元依次存放栈的元素，同时设立两个指针bottom和top分别指向栈底元素和栈顶元

素。

但是在实际实现中，栈顶指针是指向栈顶元素的下一个位置。主要是为了很方便判断栈是否为空栈，因为如果指向栈顶元素的话，那么当$top == bottom$无法判断是空栈，还是有一个元素的栈，此时需要增加一个变量来标识。当指向栈顶元素的下一个元素的话，当$top == bottom$时，此时栈是空栈。因此，当这样设置的时候，栈满的实际上栈顶指针指向的地址空间没有存放元素。

1.2.1 存储表示

相比顺序表，栈的顺序存储结构要多开辟一个指针top 指向栈顶元素的下一个位置，bottom指针指向起始地址，也就是栈底指针，存储表示如下：

```
#define MAXSIZE 50   /*分配的最大值*/
#define INCREMENT 10 /*分配增量*/
typedef int ElemType;//根据具体应用来定义类型，这里定义成int
typedef struct Stack{
    ElemType* bottom; /*栈底指针*/
    ElemType* top;    /*栈顶指针*/
    int stackSize;    /*当前的容量*/
}Stack;
```

1.2.2 操作的实现

1. 初始化栈initStack

初始化栈就是开辟存储空间，构建一个空的栈。

```
Status initStack(Stack &S){
    //构造一个空的栈
    S.bottom =(ElemType*)malloc(MAXSIZE*sizeof(
        ElemType));
    if(!S.data){
```

```
        exit(OVERFLOW);//分配失败
    }
    S.top=s.bottom;//栈顶指针指向栈底指针，空栈
    //分配成功，初始化结构的基本属性
    S.stackSize=MAXSIZE;
    return OK;
}
```

2. 进栈操作push(S,e)

进栈操作只能在栈顶的位置进行操作，因此，进栈操作算法相对来说很简单，相当于在线性表的表尾进行插入操作。算法思想如下：

Step1：判断栈是否为满，如果栈满了需要重新分配存储空间；

Step2：将进栈的元素赋值给栈顶指针指向的存储单元；

Step3：栈顶指针后移。

具体算法描述如下：

```
Status push(Stack &S,ElemType e){
    if(S.top-S.bottom>=S. stacksize-1) {
    //栈满,追加存储空间
        S.bottom=(ElemType *)realloc((S.stackSize+
            INCREMENT)*sizeof(ElemType));
        if (!S.bottom) return ERROR;
    S.top=S.bottom+S.stackSize;//栈底的地址加上当前的容量
    S.stacksize+=INCREMENT;
    }
    *S.top=e;   //进栈，e成为新的栈顶
    S.top++ ; //栈顶指针加1,
    return OK;
}
```

算法分析：因为进栈是在栈顶位置进行插入，因此时间复杂度为O(1)。

3. 出栈操作pop(S,e)

出栈操作也是在栈顶位置进行，将栈顶元素进行删除，因此只需要移动栈顶指针就可以完成出栈操作。

算法思想如下：

Step1:判断栈是否为空，如果为空栈，则返回错误；

Step2:移动栈顶指针S.top−；

Step3:返回栈顶元素，完成出栈操作。

具体的算法描述如下：

```
Status push(Stack &S,ElemType e){
    //弹出栈顶元素
    //判断栈空
    if(S.top==S.bottom)
        return ERROR ;//栈空,返回失败标志
    S.top--;//移动栈顶指针
    e=*S.top;//返回栈顶元素
    return OK ;
}
```

算法分析：因为出栈也是在栈顶位置进行，因此时间复杂度为O(1)。

其它的操作和线性表的操作类似，这里就不一一介绍了。

1.3 栈的链式存储结构与实现

栈的链式存储结构称为链栈，栈的链式存储结构和单链表一样，结点分成两个域，一个是数据域(data)，一个是指针域(next),只不过栈的栈顶指针就是头结点。进栈和出栈都限定在头指针处进行操作。因此初始化操作等和单链表一样的。这里介绍一下链栈的进栈和出栈操作。

1. 进栈操作(push)

在栈顶指针（即：头结点）处进行插入，只需要将进栈的元素结点s的指针域指向栈顶指针的next指向结点($s \rightarrow next = top \rightarrow next$)，然后将栈顶指针结点指针域指向新进栈的结点。($top \rightarrow next = s$)。

具体的算法描述如下：

```
Status push(Stack_Node *top , ElemType e)
{
    Stack_Node *s;
```

```
    s=(Stack_Node*)malloc(sizeof(Stack_Node)) ;
    if(!s) return  ERROR;
    //申请新结点失败，返回错误标志
    s->data=e;
    s->next=top->next;
    top->next=p;
    return OK;
}
```

2. 退栈操作(pop)

退栈也是在栈顶指针处进行，只需要改变栈顶指针的指向就可以实现删除操作。

具体的算法描述如下：

```
Status pop(Stack_Node *top , ElemType  e)
{
    //判断栈是否为空
    if(top->next==NULL)return ERROR;
    //保存栈顶元素结点
    q=top->next;
    //栈顶指针指向栈顶元素的后继
    top->next=q->next;
    //返回栈顶元素
    e=q->data;
    //释放结点
    free(q);
    return OK;
}
```

2 栈的应用举例

2.1 进制数间的转换

如何将十进制转换成二进制、八进制、十进制数，比如：十进制数16转

换成二进制，采用模2取余的方法，得到的余数序列是从低位到高位，也就是说最先求得的余数是二进制数的最低位，最后得到的余数是二进制数的最高位，16一直除2取余的序列为“0,0,0,0,1，但是最后的二进制数为“10000”，这个顺序是不是像栈的先进后出的特点，最先的得到的余数最后出来放在末尾上。因此，可以充分利用栈的特点来存放取余数的序列，最后依次退栈的序列就是最终的二进制序列。利用栈的基本操作来实现这个算法如下：

```
void convert(int scale)
{
    //scale为要转换的进制
    //声明一个栈
    Stack s;
    //对栈进行初始化
    initStack(s);
    //输出一个数
    int number;
    cout<<"输入需要转换的数：";
    cin>>number;
    int n=number;
    while(n!=0)
    {
        //取余数
        int k=n%scale;
        //进栈操作
        s.push(k);
        //用商更新n
        n=n/scale;
    }
    //把转换的进制数打印出来
    while(!isEmptyStack(s))
    {
        int k;
```

```
26          push(s,k);
27          cout<<k;
28      }
29      cout<<endl;
30      return OK;
31  }
```

2.2 括号匹配算法

一般表达式都包含括号，可以有{},,

这三种括号，在做计算前需要验证一下括号是不是合法出现，所谓的合法出现，必须是成对出现，也不能出现交叉的情况，比如：([)],就不是合法的表达式。要写一个程序去判断括号是否匹配，还比较麻烦，而利用栈结构来实现，就比较简单了。它的基本思想是：依次扫描表达式，如果不是括号，就继续读入下一个。如果是左括号进栈；如果是右括号，如果合法的，那么前面进栈的应该是它对应的左括号，因此此时从栈中取出栈顶元素和当前的后括号进行匹配，如果匹配成功，继续读入下一个，如果匹配不成功，则返回错误。

算法描述如下：

```
1   bool bracketMatch(string expr)
2   {
3       //expr是带括号的表达式
4           Stack s;
5           initStack(s);
6           for (int i = 0; i < expr.size(); i++)
7           {
8           //左括号进栈
9                   if (expr[i] == '(' || expr[i] == '['
                        || expr[i] == '{')
10                  {
```

```
11                  s.push(expr[i]);
12              }
13              else if (expr[i] == ')' || expr[i] ==
                    ']' || expr[i] == '}')
14              {    //右括号则进行退栈操作匹配
15                  if (s.stackEmpty())
16                  {
17              return false;
18                  }
19                  char ch = expr[i];
20                  char e;
21          //退栈进行匹配
22          pop(s,e);
23                  switch (ch)
24                  {
25                  case ']':
26                          if(e!='[')
27                          {
28                  return false;
29                          }
30                          break;
31                  case ')':
32                          if (e != '(')
33                          {
34                                  return false;
35                          }
36                          break;
37                  case '}':
38                          if (e != '{')
39                          {
40                                  return false;
41                          }
42                          break;
```

```
43                              default:
44                                      break;
45                              }
46                      }
47              }
48              return true;
49      }
```

2.3 表达式求值

可以利用栈来实现表达式求值，一个表达式包括操作数、运算符及表达式开始和结束运算符，在实现表达式求值的时候，需要判断运算符的优先级，也就是当前的运算符和前面的运算符的优先级进行比较。我们需要设立两个栈来存放表达式中的各项，其中一个是操作数栈OPND，用来存放表达式中的操作数和运算结果，另一个是运算符栈OPTR，这个栈主要用于存放表达式中的运算符。为了理解方便，此处我们做简单的表达式，里面只包括二元运算符。算法思想是：

(1)依次读取表达式，如果是操作数则进操作数栈

(2)如果是运算符则需要比较当前的运算符和操作符栈的栈顶元素进行比较优先级：

21）如果当前运算符的优先级比栈顶元素要高，则进栈，不做运算，因为后面可能还有操作符优先级比它更高；

22）如果当前运算符的优先级比栈顶元素要低，则此时需要进行计算，分别从操作数栈退两个元素，第一个元素作为操作数2，第二个元素为操作数1，进行运算后，结果进入操作数栈，读入一下继续判断。

23）如果是右括号，栈顶元素是左括号，优先级相等，此时需要将左括号退栈，读入下一个继续判断。

24）如果是结束运算符，栈顶运算符是开始结束符，则表示运算完毕，退出结束运算符。

重复以上操作，直到运算符栈为空，将操作数栈中的栈顶元素取出就是最后的结果。

注：将#作为运算符的开始和结束标识符。

算法描述如下：

```
Status compute(){
    initStack(OPTR);
    Push(OPTR,'#');
    InitStack(OPND);
    c=getchar();
    while(c!='#'||GetTop(OPTR)!='#'){
        if(!In(c,OP)){//不是运算符，进OPND栈
            Push(OPND,c);
            c=getchar();
            }
            else{
                switch(Precede(GetTop(OPTR),c)){ //判断优先级
                    case '<':  //栈顶元素优先权低
                            Push(OPTR,c);
                        c=getchar();
                        break;
                    case '=':  //去掉括号
                            Pop(OPTR,c);
                        c=getchar();
                        break;
                    case '>':  //退栈，并且计算，结果进栈
                            Pop(OPTR,theta);
                        Pop(OPND,b);
                        Pop(OPND,a);
                        Push(OPND,Operate(a,theta,b));
                        break;
                }//end swich
            }//end else
        }//end while
        return GetTop(OPND);
}
```

3 队列的定义与实现

现实生活中有很多排队的例子，比如说去食堂打饭、买车票、看演唱会安检进场等等，大家观察这种结构的特点，首先，它是一种线性结构，满足一对一的特点，其次，插入到队列中的时候只能从队尾进行插入，要得到服务的时候，只能在队首进行，一般地，这种应用都是具有“先来先服务”、“先来先得”的观点。这种结构就是队列。

3.1 栈的定义

队列也是一种操作受限的线性表，它具有“先进先出（First In First Out,FIFO）特点，因此也称为先进先出的线性表。只能在线性表的一端进行插入，另一端进行删除。

定义 6. 队首：允许删除的一端。又称队头。

定义 7. 队首指针*(front)*：指向队首的指针。

定义 8. 队尾：允许插入的一端。

定义 9. 队尾指针*(rear)*：指向队尾的指针。

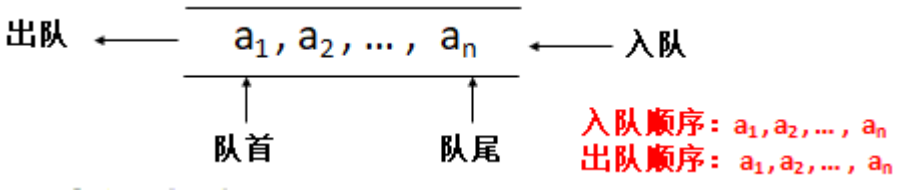

Figure 1: 队列示意图

栈的抽象数据类型定义如下：

ADT Queue{

数据对象:$D = \{a_i | a_i \in DataType\}$

数据关系:$R = \{< a_i, a_{i+1} >\}$

基本操作：

$initQueue(\&Q)$:

操作结果：建立空队列Q，创建成功返回true,并且用Q返回，否则返回false

$destroyQueue(\&Q)$:

初始条件：队列Q存在

操作结果：删除一个队列Q,删除成功返回true,否则返回false

$clearQueue(\&Q)$:

初始条件：栈Q存在

操作结果：清空队列Q，清空成功返回true,否则返回false

$isEmptyQueue(\&Q)$:

初始条件：队列Q存在

操作结果：队列Q为空，返回true,否则返回false

$getLength(\&Q)$:

初始条件：队列Q存在

操作结果：返回队列Q的长度

$getFront(\&S, \&e)$:

初始条件：Q存在

操作结果：返回队列Q的队首元素

$push(\&Q, e)$:

初始条件：队列Q存在

操作结果：在队列Q 的队尾插入元素e

$pop(\&Q, \&e)$:

初始条件：队列Q存在

操作结果：删除队列Q的队首元素，用e返回

}End Queue.

在队列的队尾插入元素，称为入队(push)操作，在队列的队头删除元素，称为出队(pop)操作。

3.2 队列的实现

队列也有两种存储结构，一种是顺序存储结构，另外一种是链式存储结构，链式存储结构相当于单链表，只不过多设立了一个指针，指向队尾，这样出队和入队操作会更简单。队首指针就是头指针，队尾指针指向队尾，入队时候再队首进行，出队在队尾进行。 因此在这里只介绍队列的顺序存储结构的实现。

3.2.1 队列的顺序存储结构表示与实现

队列的顺序存储结构的表示也称为顺序队列，用地址连续的存储单元来依次存放队列中的每一个元素。

```
#define  MAX_QUEUE_SIZE    100
typedef  struct  queue
{
    ElemType    Queue_array[MAX_QUEUE_SIZE];
    int    front;   //队首指针
    int    rear;    //队尾指针
}SqQueue;
```

从结构上看，设立一个队首指针$front$，一个队尾指针$rear$，分别指向队首和队尾元素，为了操作方便，我们约定队尾指针指向队尾元素的下一个位置。具有如下的特点：

- 初始化：$front = rear = 0$。
- 入队：将新元素插入rear所指的位置，然后rear加1。
- 出队：删去front所指的元素，然后加1并返回被删元素。
- 队列为空：$front = rear$。
- 队满：$rear = MAX_QUEUESIZE - 1$。

顺序队列“假溢出”的问题：不论是入队操作还是出队操作，rear指针和front指针都是往后移动，也就是增加，而入队操作只能在队尾进行，这样使得出队后的那些存储空间没有办法再继续使用，当队尾指针指向最大

的存储空间位置，此时无法再进行入队元素，而此时有元素出队，也就是说front¿0。由于无法进行入队操作，而实际空间是没有满的，因为有元素出队，那些存储空间没有办法使用，从而导致了“假溢出”现象。假溢出指满足空间为满的条件无法进行插入，但是空间实际却没有满。

如何解决这种“假溢出”现象呢？————采用循环队列。

3.2.2 循环队列

充分利用向量空间，克服上述“假溢出”现象的方法是：将为队列分配的向量空间看成为一个首尾相接的圆环，并称这种队列为循环队列(Circular Queue)。 在循环队列中进行出队、入队操作时，队首、队尾指针仍要加1，朝前移动。当队首、队尾指针指向数组上界(MAX_QUEUE_SIZE-1)时，其加1操作的结果是指向向量的下界0。例如：

if(rear+1==MAX_QUEUE_SIZE) rear=0;

else rear++;

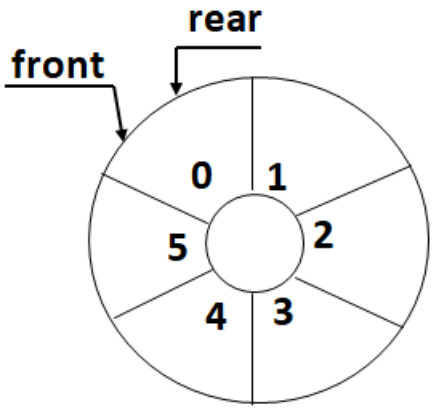

Figure 2: 循环队列示意图

问题： 循环队列如何来判断队列为空还是为满？

如果rear指向队尾元素，那么$front == rear$无法来区分是队列为空，还是为满，因为随着入队操作，“循环”回来也会使得$front == rear$，因此无法区分，因此我们还是约定rear指针是指向队尾元素的下一个位置。这样我们可以通过队尾指针加上1是否等于front来判断队列是否为满，而空队列的判断条件仍然是$front == rear$。

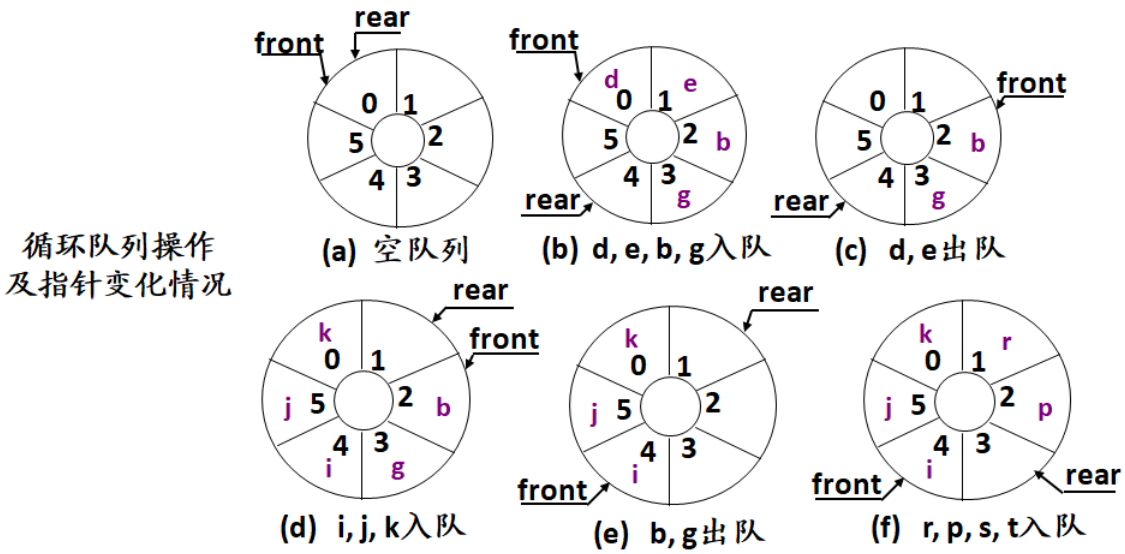

Figure 3: 循环队列出队和入队示意图

但是注意是，由于MAX_QUEUE_SIZE-1的下一个是第0个，可以通过求模运算来模拟循环队列。

因此判断队列为满的条件是：$(rear+1)\%MAX\_QUEUE\_SIZE == front$

下面分别介绍循环队列的操作:

循环队列的结构：

```
#define   MAX_QUEUE_SIZE    100
typedef   struct   SqQueue
{
    ElemType* data;
    int    front;   //队首指针
    int    rear;    //队尾指针
}SqQueue;
```

(1)循环队列的初始化

循环队列初始化就是声明一个队列，同时将该队列的队首、队尾指针都赋值为0。

```
Status initQueue(SqQueue& Q)
{
    Q.data=(ElemType*)malloc(MAX_QUEUE_SIZE*sizeof(
        ElemType));
    if(!Q.data)exit(OVERFLOW);//分配失败
    Q.front=Q.rear=0;
    return OK;
}
```

(2)入队操作

入队操作首先判断队列是否满了，如果满了返回，没有满则将插入的元素赋值给队尾指针指向的存储空间，然后队尾指针往后移动。算法描述如下：

```
Status push(SqQueue& Q, ElemType e)
{
    //将数据元素e插入到循环队列Q的队尾
    if((Q.rear+1)%MAX_QUEUE_SIZE== Q.front)
        return ERROR;//队满，返回错误标志
    Q.Queue_array[Q.rear]=e ;    // 元素e入队
    Q.rear=(Q.rear+1)% MAX_QUEUE_SIZE ;//移动队尾指针
    return OK;
}
```

(3)出队操作

出队操作，在队首进行删除，移动队首指针就可以实现。但是出队前先判断队列是否为空。算法描述如下：

```
 //将循环队列Q的队首元素出队
Status pop(SqQueue& Q, ElemType &x )
{
    if(Q.front== Q.rear)
```

```
5           return ERROR ; //队空，返回错误标志
6       x=Q.Queue_array[Q.front]; // 取队首元素
7       Q.front=(Q.front+1)%MAX_QUEUE_SIZE;// 队首指针向前移动
8       return OK ;
9   }
```

其它操作自行实现。

3.3 队列的链式存储结构

队列的链式存储结构简称为链队列，它是限制仅在表头进行删除操作和表尾进行插入操作的单链表。需要建立两类不同的结点：数据元素结点，队列的队首指针和队尾指针的结点。具体的定义如下：

```
1   typedef struct Qnode
2   {
3       ElemType data ;
4       struct Qnode *next ;
5   }QNode;
6   typedef struct link_queue
7   {
8           QNode *front;
9       QNode *rear;
10  }LinkQueue;
```

出队和入队的操作示意图如下：

(1)初始化链队列

```
1
2   Status initQueue(LinkQueue& Q)
3   {
4       QNode  *p ;
5       //开辟头结点
```

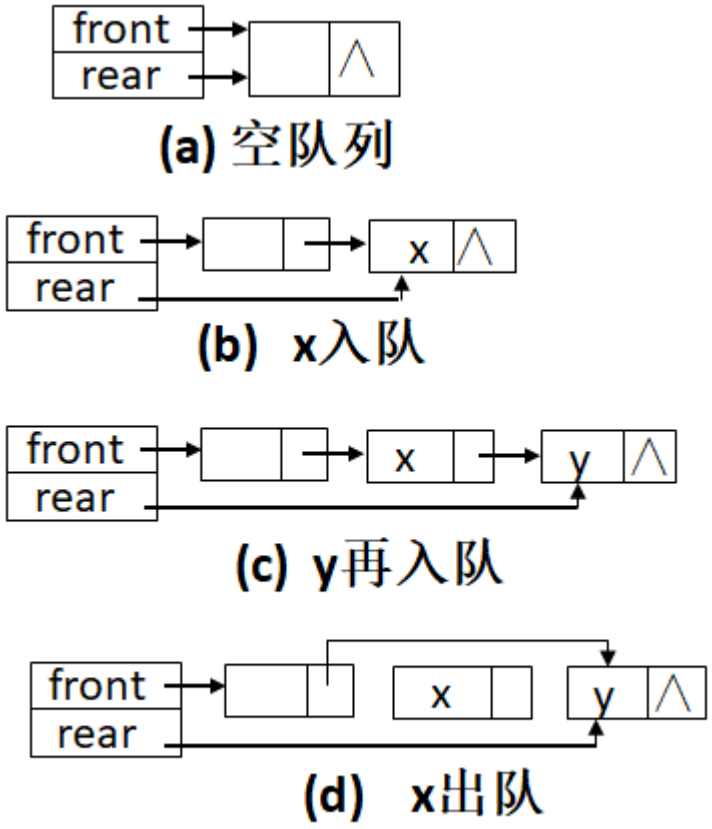

Figure 4: 循环队列出队和入队示意图

```
6       p=(QNode *)malloc(sizeof(QNode));
7       if(!p)
8       {
9           return ERROR;
10      }
11      p->next=NULL;
12      //分配
13      Q.front=Q.rear=p;
14      return OK;
15  }
```

(2)入队操作

链队列既有队首指针，又有队尾指针，插入的时候在队尾进行插入，在rear指针处插入，新插入的结点成为队尾元素。时间复杂度为：O(1)

```
 //将数据元素e插入到链队列Q的队尾
Status  Insert_CirQueue(LinkQueue  *Q , ElemType  e)
{
    p=(QNode *)malloc(sizeof(QNode));
    if(!p)
    {
        //申请新结点失败，返回错误标志
            return  ERROR;
    }
    p->data=e ;
    p->next=NULL;
    Q.rear->next=p;
    Q.rear=p; //新结点插入到队尾
    return OK;
}
```

(3)出队操作

在队首处进行删除，只需要改变队首指针的指向就可以，在执行出队前，需要判断队列是否为空。时间复杂度为O(1)。

```
Status pop(LinkQueue& Q, ElemType x){
    QNode *p ;
    if(Q.front==Q.rear)
    {
        return ERROR;//队空
    }
    p=Q.front->next ;    // 取队首结点
    *x=p->data ;
    Q.front->next=p->next ;        // 修改队首指针
```

```
10      if(p==Q.rear)
11      {
12          //当队列只有一个结点时应防止丢失队尾指针
13          Q.rear=Q.front ;
14      }
15      free(p) ;
16     return OK ;
17  }
```

其它操作类似于单链表，请自行实现。

4 队列的应用举例

实际生活中，队列的应用随处可见，比如排队买东西、医院的挂号系统、银行的业务办理系统等，采用的都是队列的结构。

例 1. 某银行有一人客户办理业务站，在单位时间内随机地有客户到达，设每位客户的业务办理时间是某个范围的随机值。设只有一个窗口，一位业务人员，要求程序模拟统计在设定时间内，业务人没的总空闲时间和客户的平均等待时间。假定模拟数据已按客户到达的先后顺序依次存于某个正文数据文件中，对应每位客户有两个数据：到达时间和需要办理业务的时间。

具体实现代码如下，自行分析并理解实现过程，在开发工具中去实现：

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  #define OVERFLOW -2
4  typedef struct{
5      int arrive;
6      int treat;     //客户的信息结构
7  }QNODE;
8  typedef struct node{
9      QNODE data;
10     struct node *next;     //队列中的元素信息
```

```
}LNODE;
LNODE *front,*rear;   //队头指针和队尾指针
QNODE curr,temp;
char Fname[120];
FILE *fp;
void EnQueue(LNODE **hpt,LNODE **tpt,QNODE e)     //队列进队
{
    LNODE *p=(LNODE*)malloc(sizeof(LNODE));
    if(!p)    exit(OVERFLOW);
    p->data=e;
    p->next=NULL;
    if(*hpt==NULL) *tpt=*hpt=p;
    else *tpt=(*tpt)->next=p;
}
int DeQueue(LNODE **hpt,LNODE **tpt,QNODE *cp)      //链接队列出队
{
    LNODE *p=*hpt;
    if(*hpt==NULL)    return 1;    //队列空
    *cp=(*hpt)->data;
    *hpt=(*hpt)->next;
    if(*hpt==NULL) *tpt=NULL;
    free(p);
    return 0;
}
void main()
{
    int dwait=0,clock=0,wait=0,count=0,have=0,finish;
    printf("/n enter file name:");
    scanf("%s",Fname);   //输入装客户模拟数据的文件的文件名
    if((fp=fopen(Fname,"r"))==NULL)    //打开数据文件
    {
        printf("cannot open file %s",Fname);
```

```
43          return 0;
44      }
45      front=NULL;   rear=NULL;
46      have=fscanf(fp,"%d%s",&temp.arrive,&temp.treat);
47      do{           //约定每轮循环，处理一个客户
48          if(front==NULL&&have==2)       //等待队列为空，但还
                有客户
49          {
50              dwait+=temp.arrive-clock;   //累计业务员总等待
                    时间
51              clock=temp.arrive;
52              EnQueue(&front,&rear,temp); //暂存变量中客户
                    的信息进队
53              have=fscanf(fp,"%d%d",&temp.arrive,&temp.
                    treat);
54          }
55          count++;
56          DeQueue(&front,&rear,&curr); //出队一位客户信息
57          wait+=clock-curr.arrive;   //累计到客户的等待时间
58          finish=clock+curr.treat;   //设定业务办理结束时间
59          while(have==2&&temp.arrive<=finish)     //下一位客
                户到达时间在当前客户处理结束之前
60          {
61              EnQueue(&front,&rear,temp);   //暂存变量中的客
                    户信息进队
62              have=fscanf(fp,"%d%d",&temp.arrive,&temp.
                    treat);
63          }
64          clock=finish;   //时钟推进到当前客户办理结束时间
65      }while(have==2||front!=NULL);
66      printf("结果：业务员等待时间%d\客户平均等待时
            间n%f\n",dwait,(double)wait/count);
67      printf("模拟总时间：%d,\客户人数：n%d,\总等待时间：
            n%d\n",clock,count,wait);
68      getchar();
69  }
```

5 小结

本章介绍了两种操作受限的线性表：栈与队列，主要内容包括：

- 栈的定义与特点：后进先出
- 栈的顺序存储结构与链式存储结构
- 退栈和进栈的算法与实现(重点)
- 队列的定语与特点：先进先出
- 队列的顺序存储结构与链式存储结构
- 出队与入队的算法与实现(重点)

6 Next

下一章介绍另外一种线性结构————串，又称字符串。主要内容如下：

(1)传的表示与实现

(2)简单模式匹配算法的实现

(3)KMP算法的实现。

7 习题

一、填空题

（1）栈的特点是_____,队列的特点是_____。

（2）进栈操作在underline 处插入，退栈操作在_____处进行。

（3）队列能插入的一端称为_____，能删除的一端是_____。

（4）顺序队列会出现_____现象，用_____解决。

（5）具有队首指针front和队尾指针rear的循环队列（最大空间为Max）为空的条件是_____,队列为满的条件为_____

二、算法题：

1. 如果一个程序需要使用多个栈，使用顺序栈就会造成栈空间大小难以估计，从而造成有的栈溢出有的栈空闲，此时可以建立一个共享栈，通

俗地讲就是将两个栈的栈底设置在同一个数组的两端，栈顶位置用top1、top2表示。如图：

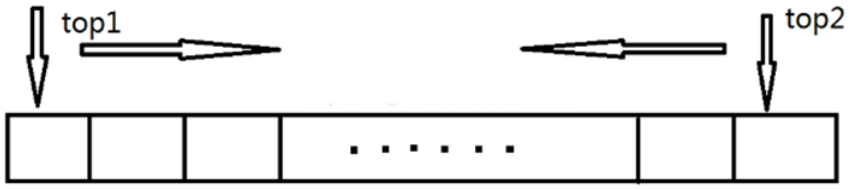

Figure 5: 共享栈

实现共享栈的顺序存储描述以及实现初始化、进栈、出栈等算法（算法思想和代码描述）。

2.设计双端队列：一端运行插入删除，另一端只运行插入。实现双端队列的顺序存储描述，并实现初始化、出队、入队等算法。

8 附录

8.1 顺序栈的实现代码

利用模板类来实现栈的顺序存储结构。

头文件“Stack.h”的代码：

```
#pragma once
#include "predefine.h"

template<class T>
class Stack
{
public:
        Stack();
        ~Stack();
public:
        void initStack();
        bool stackFull();
        bool stackEmpty();
        Status push(T x);
        Status pop(T &x);
        Status stackTop(T &x);
        void traverse();
private:
        T *data;
        T *top;
        int stackSize;
};
template<class T>
Stack<T>::Stack()
{
}

```

```
28 template<class T>
29 Stack<T>::~Stack()
30 {
31         free(data);
32 }
33 template<class T>
34 void Stack<T>::initStack()
35 {
36         data = (T*)malloc(MAX * sizeof(T));
37         if (!data)
38         {
39                 cout << "构造失败" << endl;
40                 exit(OVERFLOW);
41         }
42         top = data;
43
44         stackSize = MAX;
45 }
46 template<class T>
47 bool Stack<T>::stackFull()
48 {
49         if (top - data >= stackSize - 1)
50         {
51                 return true;
52         }
53         return false;
54 }
55 template<class T>
56 bool Stack<T>::stackEmpty()
57 {
58         if (top == data)
59         {
60                 return true;
```

```
61          }
62          return false;
63  }
64  template<class T>
65  Status Stack<T>::push(T x)
66  {
67          if (stackFull())
68          {
69                  T* newbase = (T*)realloc(data, (
                        stackSize + INCREMENT) * sizeof(T))
                        ;
70
71                  if (!newbase)
72                  {
73                          cout << "重新分配失败" << endl;
74                          exit(OVERFLOW);
75                  }
76                  data = newbase;
77                  top = data + stackSize;
78          }
79
80          *top = x;
81          top++;
82          return OK;
83  }
84  template<class T>
85  Status Stack<T>::pop(T &x)
86  {
87          if (stackEmpty())
88          {
89                  cout << "空栈，删除失败! " << endl;
90                  return ERROR;
91          }
```

```
92          top--;
93          x = *top;
94
95          return OK;
96  }
97  template<class T>
98  Status Stack<T>::stackTop(T &x)
99  {
100         if (!stackEmpty())
101         {
102                 cout << "空栈。。。" << endl;
103                 return ERROR;
104         }
105         x = *(top - 1);
106         return OK;
107 }
108 template<typename T>
109 void Stack<T>::traverse()
110 {
111         T* p = data;
112         if (stackEmpty()) {
113                 cout << "空栈当前没有元素," << endl;
114                 return;
115         }
116         while (p < top) {
117                 cout << *p++ << " ";
118         }
119         cout << endl;
120 }
```

测试文件“TestStack.cpp”中的代码：

```
1  #pragma once
2  #include <iostream>
```

```cpp
#include "Stack.h"

using namespace std;

int main()
{
	Stack<int> s;
	s.initStack();

	while (true)
	{
		cout << "输入进栈的元素：";
		int e;
		cin >> e;
		s.push(e);
		cout << "继续输入(y or n)？";
		char ch;
		cin >> ch;
		if (ch == 'n')
		{
			break;
		}
	}
	s.traverse();


	int x;
	while(!s.stackEmpty())
	{
		s.pop(x);
		cout << "退栈的元素为：" << x << endl;
	}

```

```
36          s.traverse();
37
38          system("pause");
39          return 0;
40  }
```

8.2 链栈的实现代码

头文件“StackList.h”中的代码：

```
1   #pragma once
2   #include "predefine.h"
3   template<class T>
4   struct stackNode
5   {
6           T data;
7           stackNode* next;
8   };
9   template<class T>
10  class StackList
11  {
12  public:
13          StackList();
14          ~StackList();
15  public:
16          void initStack();
17          bool stackEmpty();
18          Status push(T x);
19          Status pop(T &x);
20          Status stackTop(T &x);
21          void traverse();
22  private:
23          int length;
```

```
24          stackNode<T> *top;
25  };
26  template<class T>
27  StackList<T>::StackList()
28  {
29  }
30
31  template<class T>
32  StackList<T>::~StackList()
33  {
34          stackNode<T>* p=new stackNode<T>;
35          while (top)
36          {
37                  p = top;
38                  top = top->next;
39                  delete (void*)p;
40          }
41  }
42  template<class T>
43  void StackList<T>::initStack()
44  {
45          top=new stackNode<T>;
46          if (!top)
47          {
48                  cout << "构造失败" << endl;
49                  exit(OVERFLOW);
50          }
51          top->next = 0;
52
53          length=0;
54  }
55  template<class T>
56  bool StackList<T>::stackEmpty()
```

```
57 {
58         if (top->next==NULL)
59         {
60                 return true;
61         }
62         return false;
63 }
64 template<class T>
65 Status StackList<T>::push(T x)
66 {
67         stackNode<T>* node = new stackNode<T>;
68         node->next = top->next;
69         node->data = x;
70         top->next = node;
71         return OK;
72 }
73 template<class T>
74 Status StackList<T>::pop(T &x)
75 {
76         if (stackEmpty())
77         {
78                 cout << "空栈，删除失败! " << endl;
79                 return ERROR;
80         }
81         stackNode<T>* p = top->next;
82         x = p->data;
83         top->next = p->next;
84         delete p;
85
86         return OK;
87 }
88 template<class T>
89 Status StackList<T>::stackTop(T &x)
```

```
90  {
91          if (!stackEmpty())
92          {
93                  cout << "空栈。。。" << endl;
94                  return ERROR;
95          }
96          x = top->data;
97          return OK;
98  }
99  template<typename T>
100 void StackList<T>::traverse()
101 {
102         stackNode<T>* p = top->next;
103         if (stackEmpty()) {
104                 cout << "空栈当前没有元素," << endl;
105                 return;
106         }
107         while (p) {
108                 cout << p->data << " ";
109                 p = p->next;
110         }
111         cout << endl;
112 }
```

测试文件“TestStackList.cpp”中的代码：

```
1  #include <iostream>
2  #include <string>
3  #include "StackList.h"
4  using namespace std;
5
6  int main()
7  {
8          StackList<string> s;
```

```
9           s.initStack();
10
11          while (true)
12          {
13                  cout << "输入进栈的元素：";
14                  string e;
15                  cin >> e;
16                  s.push(e);
17                  cout << "继续输入(y or n)？";
18                  char ch;
19                  cin >> ch;
20                  if (ch == 'n')
21                  {
22                          break;
23                  }
24          }
25          s.traverse();
26
27
28          string x;
29          while (!s.stackEmpty())
30          {
31                  s.pop(x);
32                  cout << "退栈的元素为：" << x << endl;
33          }
34
35          s.traverse();
36          system("pause");
37          return 0;
38  }
```

8.3 栈的应用实现代码

（1）进制转换

```cpp
#include <iostream>
#include "Stack.h"
using namespace std;

int main()
{
	Stack<int> s;
	s.initStack();

	int number;
	int decimal;
	while (true)
	{
		cout << "请输入要转换的数：";
		cin >> number;
		cout << "输入转换的进制：";
		cin >> decimal;

		while (number > 0)
		{
			int k = number % decimal;
			s.push(k);
			number = number / decimal;
		}
		while (!s.stackEmpty())
		{
			int x;
			s.pop(x);
			cout << x;
		}
		cout << endl;
	}
	system("pause");
```

```
34          return 0;
35  }
```

（2）括号匹配

```
1   #include <iostream>
2   #include <string>
3   #include "Stack.h"
4
5   using namespace std;
6
7   int main()
8   {
9           Stack<char> s;
10          s.initStack();
11          string expr;
12
13          cout << "请输入表达式：";
14          cin >> expr;
15          for (int i = 0; i < expr.size(); i++)
16          {
17              if (expr[i] == '(' || expr[i] == '[' ||
                    expr[i] == '{')
18              {
19                      s.push(expr[i]);
20              }
21              else if (expr[i] == ')' || expr[i] == ']'
                    || expr[i] == '}')
22              {
23                      if (s.stackEmpty())
24                      {
25                              cout << "左右括号不一致：
                                    " << expr[i] << endl;
26                              break;
```

```
27                  }
28                  char ch = expr[i];
29                  char e;
30                  switch (ch)
31                  {
32                  case ']':
33                    s.pop(e);
34                        if(e!='[')
35                        {
36                              cout << "当前" << ch << "
                                  和" << e << "不匹
                                  配" << endl;
37                                 break;
38                        }
39                        break;
40                  case ')':
41                        s.pop(e);
42                        if (e != '(')
43                        {
44                         cout << "当前" << ch << "和"
                              << e << "不匹
                              配" << endl;
45                                          break;
46                        }
47                        break;
48                  case '}':
49                        s.pop(e);
50                        if(e != '{')
51                        {
52                         cout << "当前" << ch << "和"
                              << e << "不匹
                              配" << endl;
53                         break;
54                        }
55                        break;
```

```
56                      default:
57                              break;
58              }
59                      }
60          }
61          system("pause");
62          return 0;
63  }
```

8.4 循环队列的实现代码

头文件“Queue.h”中的代码：

```
1  #pragma once
2  #include "predefine.h"
3
4  template<typename T>
5  class Queue
6  {
7  public:
8          Queue();
9          ~Queue();
10 public:
11         Status initQueue();
12         int getLength();
13         bool isEmpty();
14         bool isFull();
15         Status push(T e);
16         Status pop(T& e);
17         void traverse();
18 private:
19         T* data;
20         int rear;
```

```
21          int front;
22  };
23  template<class T>
24  Queue<T>::Queue()
25  {
26  }
27
28  template<class T>
29  Queue<T>::~Queue()
30  {
31          free(data);
32  }
33  template<typename T>
34  Status Queue<T>::initQueue()
35  {
36          data = (T*)malloc(MAX * sizeof(T));
37          if (!data)
38          {
39                  cout << "初始化失败..." << endl;
40                  return ERROR;
41          }
42          rear = 0;
43          front = 0;
44          return OK;
45  }
46  template<typename T>
47  int Queue<T>::getLength()
48  {
49          return (rear - front + MAX) % MAX;
50  }
51  template<typename T>
52  bool Queue<T>::isEmpty()
53  {
```

```
54          if (rear == front)
55          {
56                  return true;
57          }
58          return false;
59  }
60  template<typename T>
61  bool Queue<T>::isFull()
62  {
63          if ((rear + 1) % MAX == front)
64          {
65                  return true;
66          }
67          return false;
68  }
69  template<typename T>
70  Status Queue<T>::push(T e)
71  {
72          if (isFull())
73          {
74                  return ERROR;
75          }
76          data[rear] = e;
77          rear = (rear + 1) % MAX;
78          return OK;
79  }
80  template<typename T>
81  Status Queue<T>::pop(T& e)
82  {
83          if (isEmpty())
84          {
85                  return ERROR;
86          }
```

```
87          e = data[front];
88          front = (front + 1) % MAX;
89          return OK;
90  }
91  template<typename T>
92  void Queue<T>::traverse()
93  {
94          if (isEmpty())
95          {
96                  cout << "空队列..." << endl;
97                  return;
98          }
99
100         int p = front;
101
102         while (p != rear)
103         {
104                 cout << data[p] << " ";
105                 p = (p + 1) % MAX;
106         }
107         return;
108 }
```

测试文件“TestQueue.cpp”中的代码：

```
1   #include <iostream>
2   #include "Queue.h"
3
4   using namespace std;
5   int main_test()
6   {
7           Queue<int> s;
8           s.initQueue();
9           while (true)
```

```
10          {
11                  cout << "输入入队的元素：";
12                  int e;
13                  cin >> e;
14                  s.push(e);
15                  cout << "继续输入(y or n)？";
16                  char ch;
17                  cin >> ch;
18                  if (ch == 'n')
19                  {
20                          break;
21                  }
22          }
23          s.traverse();
24
25          int x;
26          while (!s.isEmpty())
27          {
28                  s.pop(x);
29                  cout << "出队的元素为：" << x << endl;
30          }
31
32          s.traverse();
33          system("pause");
34          return 0;
35  }
```