数据结构

北京邮电大学 网络空间安全学院

武 斌

上章内容

上一章（线性表）内容：

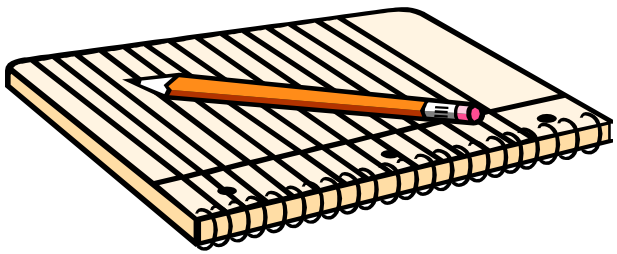
- 了解线性表的概念及其逻辑结构特性
- 理解顺序存储结构和链式存储结构的描述方法
- 掌握线性表的基本操作及算法实现
- 重点掌握线性链表的存储结构及算法实现
- 分析两种存储结构的时间和空间复杂度

本次课程学习目标

学习完本次课程，您应该能够：

- 掌握队列抽象数据类型的特点
- 熟练掌握循环队列和链队列的基本操作实现算法
- 离散事件驱动

本章课程内容（第三章 栈和队列）



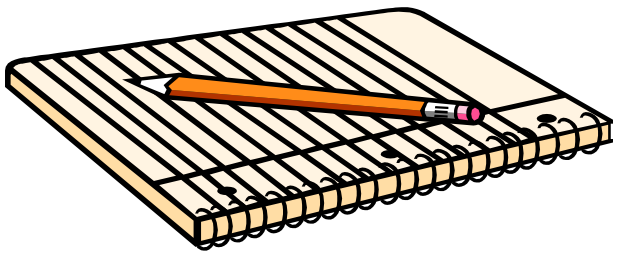
- 3.1 栈
- 3.2 栈的应用举例
- 3.3 队列
- 3.4 队列应用举例
- 3.5 离散事件模拟

第三章 栈和队列

- 栈和队列是在程序设计中被广泛使用的两种线性数据结构。
- 它们的特点在于基本操作的特殊性，栈必须按“后进先出”的规则进行操作，而队列必须按“先进先出”的规则进行操作。和线性表相比，它们的插入和删除操作受更多的约束和限定，故又称为限定性的线性表结构。可将线性表和栈及队列的插入和删除操作对比如下：

→	插 入	删 除	
线性表：	Insert(L,i,x) (1≤i≤n+1)	Delete(L,i) (1≤i≤n)	线性表允许在表内任一位置进行插入和删除
栈：	Insert(L,n+1,x)	Delete(L,n)	栈只允许在表尾一端进行插入和删除
队列：	Insert(L,n+1,x)	Delete(L,1)	队列只允许在表尾一端进行插入，在表头一端进行删除

队列的类型定义



- 3.1 栈的类型定义
- 3.2 栈的应用举例
- 3.3 队列的类型定义
- 3.4 队列的应用举例
- 3.5 离散事件模拟

队列的类型定义

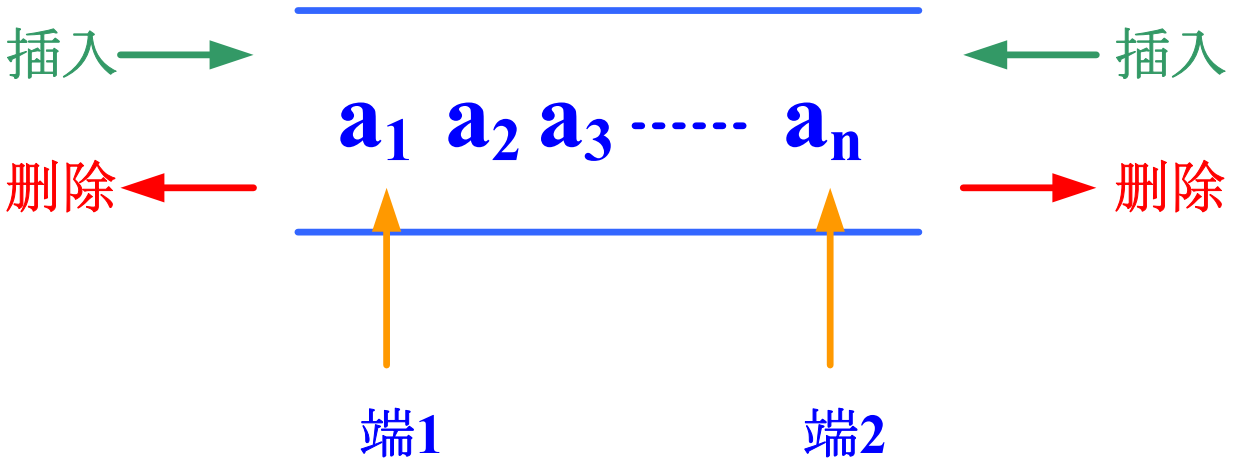
- 队列(Queue)也是一种运算受限的线性表。它只允许在表的一端进行插入，而在另一端进行删除。允许删除的一端称为队头(front)，允许插入的一端称为队尾(rear)。
- 例如：排队购物，操作系统中的作业排队。
- 先进入队列的成员总是先离开队列，因此，队列亦称作先进先出(First In First Out)的线性表，简称FIFO表。

队列的类型定义

- 当队列中没有元素时称为空队列。在空队列中依次加入元素$a_1,a_2,\ldots a_n$之后，a_1是队头元素，a_n是队尾元素。显然退出队列的次序也只能是$a_1,a_2,\ldots a_n$ ，也就是说**队列的修改**是**依先进先出的原则**进行的。
- 下图是队列的示意图：

队列的类型定义

- 双端队列（deque）：



队列的类型定义

- 队列的类型定义如下：

ADT Queue **{**

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

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

约定 a_n 端为队列尾，a_1 端为队列头。

基本操作：

InitQueue(&Q)

操作结果：构造一个空队列 Q。

DestroyQueue(&Q)

初始条件：队列 Q 已存在。

操作结果：队列 Q 被销毁，不再存在。

队列的类型定义

ClearQueue(&Q)

初始条件：队列 Q 已存在。

操作结果：将 Q 清为空队列。

QueueEmpty(Q)

初始条件：队列 Q 已存在。

操作结果：若 Q 为空队列，则返回TRUE，否则返回FALSE。

QueueLength(Q)

初始条件：队列 Q 已存在。

操作结果：返回 Q 的元素个数，即队列的长度。

GetHead(Q,**&**e)

初始条件：Q 为非空队列。

操作结果：用 e 返回Q的队头元素。

队列的类型定义

EnQueue(**&**Q,e)

初始条件：队列 Q 已存在。

操作结果：插入元素 e 为 Q 的新的队尾元素。

DeQueue(**&**Q,**&**e)

初始条件：Q 为非空队列。

操作结果：删除 Q 的队头元素，并用 e 返回其值。

QueueTraverse(Q,visit( ))

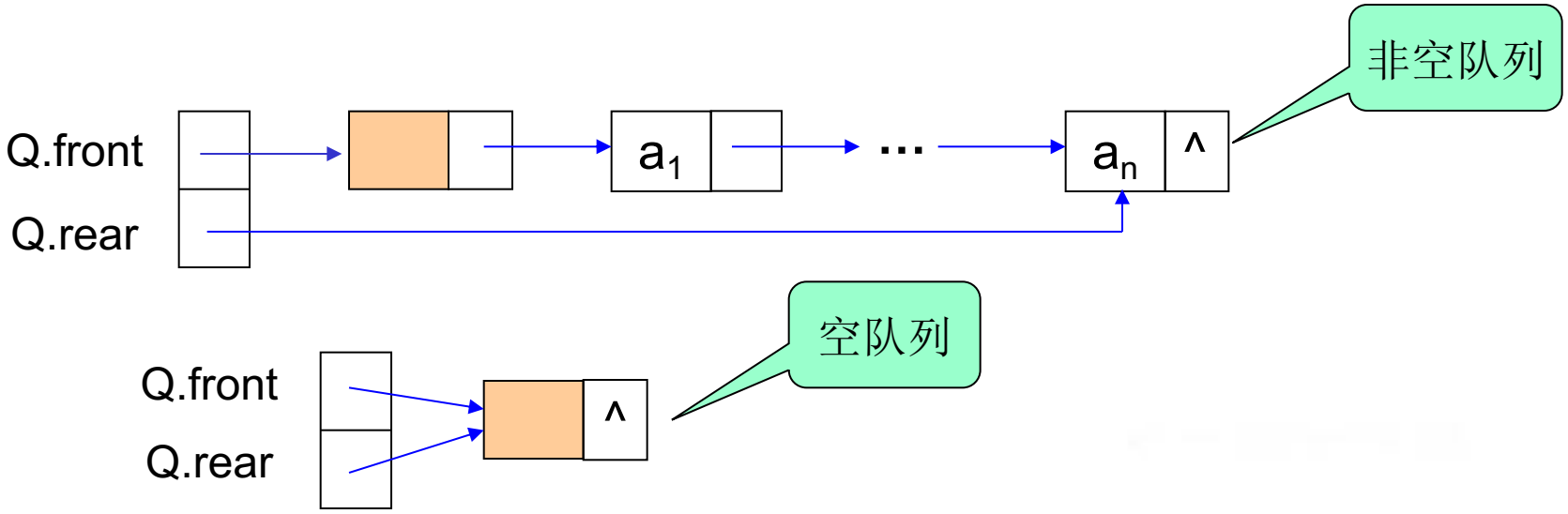
初始条件：队列 Q 已存在且非空，visit( )为元素的访问函数。

操作结果：依次对 Q 的每个元素调用函数 visit( )，一旦 visit( ) 失败则操作失败。

} ADT Queue

队列的存储表示和操作的实现

- 队列的链式存储结构简称为**链队列**，它是限制仅在表头删除和表尾插入的单链表。显然仅有单链表的头指针不便于在表尾做插入操作，为此再增加一个尾指针，指向链表的最后一个结点。于是，一个链队列由一个头指针唯一确定。



队列的存储表示和操作的实现

- 链队列的类型定义：
  - // 结构定义

```
typedef struct QNode
{
   QElemType data;
   struct QNode *next;
}QNode,*QueuePtr;
typedef struct
{
   QueuePtr front; //队头指针
   QueuePtr rear;  //队尾指针
}LinkQueue;
```

队列的存储表示和操作的实现

➔ //基本操作接口（函数声明）

```
Status InitQueue (LinkQueue &Q);
//   构造一个空队列 Q
Status DestroyQueue (LinkQueue &Q);
//   销毁队列Q，Q 不再存在
Status ClearQueue (LinkQueue &Q);
//   将 Q 置为空队列
Status QueueEmpty (LinkQueue Q);
//   若队列 Q 为空队列，则返回TRUE，否则返回FALSE
int QueueLength (LinkQueue Q);
//   返回队列 Q 中元素个数，即队列的长度
```

队列的存储表示和操作的实现

➔ 基本操作接口（函数声明续）

```
Status GetHead (LinkQueue Q, QElemType &e);
//  若队列不空，则用 e 返回Q的队列头元素，并返回TRUE；
//  否则返回FALSE
Status EnQueue (LinkQueue &Q, QElemType e);
//  在当前队列的尾元素之后插入元素 e 为新的队列尾元素
Status DeQueue (LinkQueue &Q, QElemType &e);
//  若队列不空，则删除当前队列Q中的头元素，用 e 返回其值，
//  并返回TRUE； 否则返回 FALSE
Status QueueTraverse(LinkQueue Q, void (*visit(QElemType ));
//  依次对Q的每个元素调用函数visit( )，一旦visit( )失败，则操作失败。
```

队列的存储表示和操作的实现

- 以下给出其中4个函数的定义：
  - **Status** InitQueue (LinkQueue **&**Q)

```
{
    // 构造一个空队列 Q
    Q.Front = Q.rear = (QueuePtr)malloc(sizeof(QNode));
    if ( !Q.front ) exit(OVERFLOW);      // 存储分配失败
    Q.front->next = NULL;
    return OK;
}
```

队列的存储表示和操作的实现

- **Status** DestroyQueue (LinkQueue **&**Q)

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

队列的存储表示和操作的实现

- 演示 3-3-11

➔ **Status** EnQueue(LinkQueue **&**Q,QElemType e)

```
{// 在当前队列的尾元素之后，插入元素 e 为新的队列尾元素
    p = (QueuePtr) malloc (sizeof(QNode));
    if (!p) exit(OVERFLOW);        // 存储分配失败
    p->data=e;    p->next = NULL;
    Q.rear->next = p;    // 修改尾结点的指针
    Q.rear = p;              // 移动队尾指针
    return OK;
}
```

队列的存储表示和操作的实现

- 演示 3-3-12-1

```
Status DeQueue( LinkQueue &Q, QElemType &e)
{
    // 若队列不空，则删除当前队列 Q 中的头元素，用 e 返回其
    //值，并返回 TRUE；否则返回 FALSE
    if ( Q.front == Q.rear )       // 链队列中只有一个头结点
        return FALSE;
    p = Q.front->next;
    e = p->data;                   // 返回被删元素
    Q.front->next = p->next;       // 修改头结点指针
    if(Q.rear == p)  Q.rear = Q.front;  //如果没有此句？？？ 演示
                                                    3-3-12-2
    free(p);                       // 释放被删结点
    return OK;
} // DeQueue
```

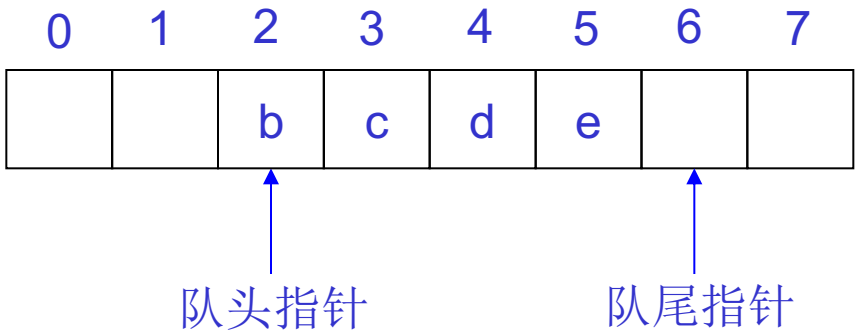
- bo3-2.cpp main3-2.cpp

循环队列

- 循环列队（队列的顺序表示和实现）
  - 和顺序栈相类似，在利用顺序分配存储结构实现队列时，除了用一维数组描述队列中数据元素的存储区域之外，尚需设立两个指针 front 和 rear 分别指示“队头”和“队尾”的位置。
  - 为了叙述方便，在此约定：
    - 初始化建空队列时，令 front=rear=0；
    - 每当插入一个新的队尾元素后，尾指针 rear增1；
    - 每当删除一个队头元素之后，头指针front增1。
  - 因此，在非空队列中，头指针始终指向队头元素，而尾指针指向队尾元素的“下一个”位置。

循环队列

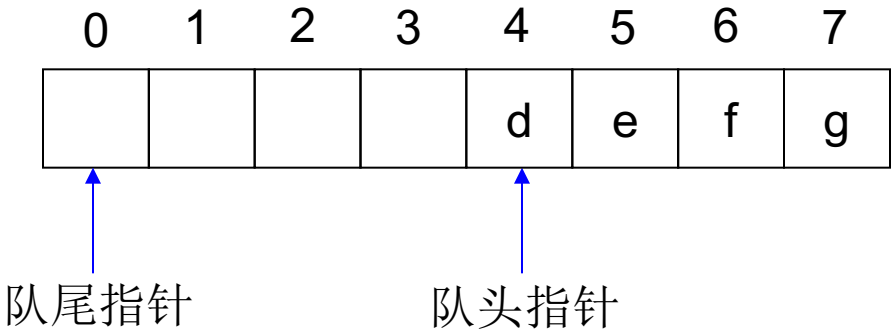
- 循环队列



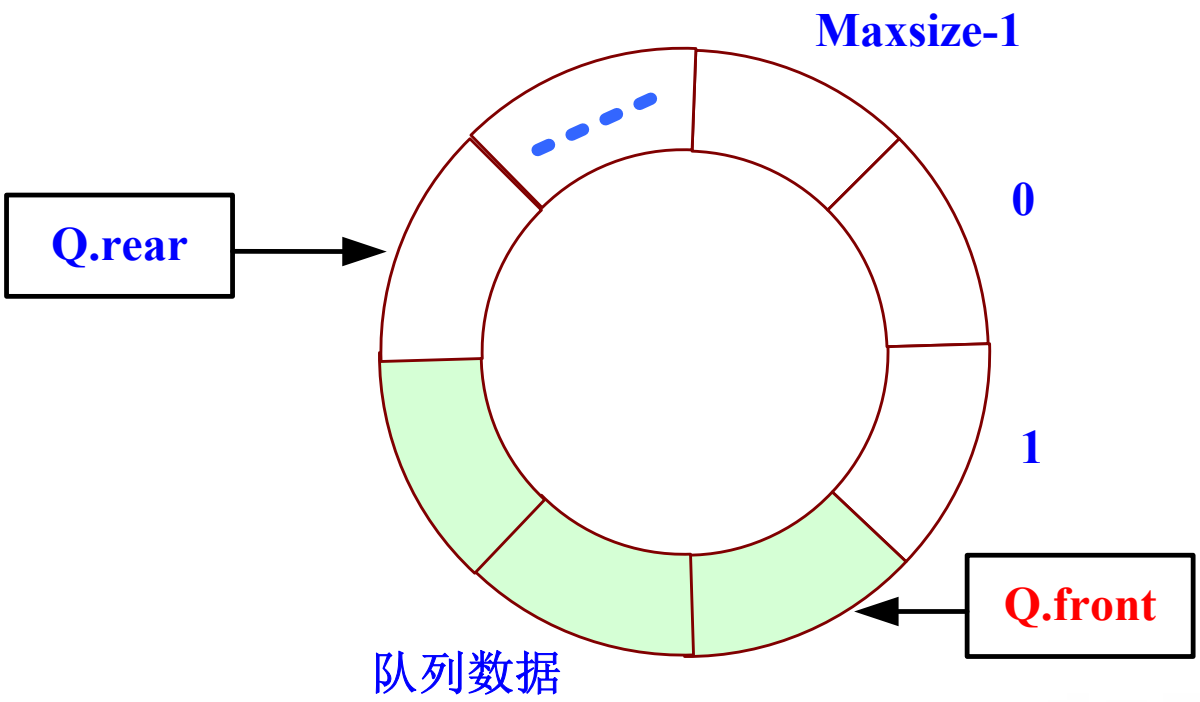
- 假设在这之后又有两个元素 f 和 g 相继入队列，而队列中的元素 b 和 c 又相继出队列，队列的变化？

循环队列

- 队头指针指向元素 d，队尾指针则指到数组"之外"的位置上去了，致使下一个入队操作无法进行(请注意此时队列空间并未满)。为此，设想这个数组的存储空间是个"环"，认定"7"的下一个位置是"0"。



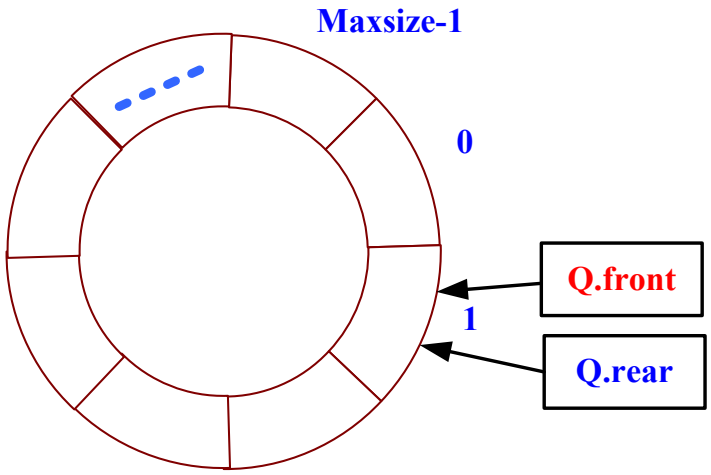
循环队列



演示 3-3-13

循环队列

- 请判断如图队列的状态？（满或空？）



- 可用看出，只凭Q.front==Q.rear无法判断队列是“空”或“满”，可以使用下面方法解决：
  - 另设标志位以区别队列是“空”或“满”；
  - 少用一个元素空间，以“队列头指针在队列尾指针的下一位置上”作为队列“满”；

队列的存储表示和操作的实现

- 循环列队的类型定义：
  - //结构定义

```
#define MAX_QSIZE 5 /* 最大队列长度+1 */
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 */
}SqQueue;
```

队列的存储表示和操作的实现

- 以下是循环队列的主要操作的函数定义。

```
Status InitQueue (SqQueue &Q)
{
    // 构造一个最大存储空间为 maxsize 的空队列 Q
    Q.Base = (QElemType*)malloc(MAXQSIZE *
       sizeof(QElemType));
     // 为循环队列分配存储空间
    if (!Q.base) exit(OVERFLOW);          // 存储分配失败
    Q.front = Q.rear = 0;
    return OK;
} // InitQueue
```

队列的存储表示和操作的实现

➔ **Status** EnQueue (SqQueue **&**Q, QElemType e)

```
{
    // 若当前队列不满，就在当前队列的尾元素之后，
    // 插入元素 e 为新的队列尾元素,并返回TRUE,否则返回FALSE
    if ((Q.rear + 1) % MAXQSIZE == Q.front )  //注意判满的方法
            return ERROR;
    Q.base[Q.rear] = e;
    Q.rear = (Q.rear+1) % MAXQSIZE;
    return OK;
}
```

队列的存储表示和操作的实现

➔ **Status** DeQueue (SqQueue **&**Q, QElemType **&**e)

```
{
    // 若队列不空，则删除当前队列Q中的头元素，用 e 返回其值
    // 并返回TRUE；否则返回 FALSE
    if (Q.front == Q.rear) //注意判空的方法
        return ERROR;
    e = Q.base[Q.front];
    Q.front = (Q.front+1) % MAXQSIZE;
    return OK;
}
```

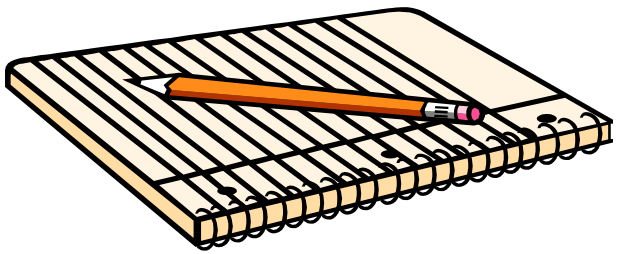
队列的存储表示和操作的实现

```
int QueueLength (SqQueue Q)
{
    // 返回队列Q中元素个数，即队列的长度
    return ((Q.rear-Q.front+MAXQSIZE) % MAXQSIZE);
}
```

❊ 注意：因为在循环队列中，队尾指针的“数值”有可能比队头指针的数值小，因此为避免在求队列长度两者相减时出现负值的情况，在作取模运算之前先加上一个最大容量的值。

- bo3-3.cpp main3-3.cpp

队列的应用举例



- 3.1 栈的类型定义
- 3.2 栈的应用举例
- 3.3 队列的类型定义
- 3.4 队列的应用举例
- 3.5 离散事件模拟

队列的应用举例

- **例3-6** 循环队列应用举例
  - 编写一个打印二项式系数表（即**杨辉三角**）的算法。

1

1 1

1 2 1

1 3 3 1

1 4 6 4 1

… … … … … … … …

  - 系数表中的第 k 行有 k 个数，除了第一个和最后一个数为1之外，其余的数则为上一行中位其左、右的两数之和。

队列的应用举例

- 一种最直接的想法
  - 利用两个数组，其中一个存放已经计算得到的第 k 行的值
  - 然后输出第 k 行的值，同时计算第 k+1行的值。
  - 如此写得的程序显然结构清晰，但需要两个辅助数组的空间，并且这两个数组在计算过程中需相互交换。
- **“循环队列”**引入
  - 可以省略一个数组的辅助空间，而且可以利用队列的操作将“琐碎操作”屏蔽起来，使程序结构变得清晰，容易被人理解。

队列的应用举例

- 如果要求计算并输出杨辉三角前 n 行的值，则队列的最大空间应为 n+2。假设队列中已存有第 k 行的计算结果，并为了计算方便，在两行之间添加一个“0”作为行界值，则在计算第 k+1 行之前，头指针正指向第 k 行的“0”，而尾元素为第 k+1 行的“0”。
- 由此从左到右依次输出第 k 行的值，并将计算所得的第 k+1 行的值插入队列的基本操作为：

```
do {
        DeQueue(Q, s);      // s 为二项式系数表第 k 行中“左上方”的值
        GetHead(Q, e);      // e 为二项式系数表第 k 行中“右上方”的值
        cout<<e;            // 输出 e 的值
        EnQueue(Q, s+e);    // 计算所得第 k+1 行的值入队列
} while (e!=0);
```

- 演示 3-4-1

队列的应用举例

●算法3.4

```
void yanghui ( int n )
{
    // 打印输出杨辉三角的前 n( n>0 )行
    Queue Q;
    for( i=1; i<=n; i++)
        cout<< ' ';
    cout<< '1'<<endl;          // 在中心位置输出杨辉三角最顶端的"1"
    InitQueue(Q,n+2);          // 设置最大容量为 n+2 的空队列
    EnQueue(Q,0 );             // 添加行界值
    EnQueue( Q,1);
    EnQueue( Q,1 );            // 第一行的值入队列
    k = 1;
```

队列的应用举例

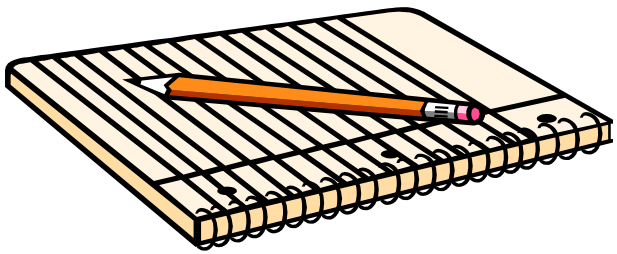
```
while ( k < n )
{                                          // 通过循环队列输出前 n-1 行的值
      for( i=1; i<=n-k; i++)
           cout<< ‘ ’;                      // 输出n-k个空格以保持三角型
      EnQueue ( Q,0 );                     // 行界值“0”入队列
      do {                                 // 输出第 k 行，计算第 k+1 行
           Dequeue( Q,s );
           GetHead( Q,e );
           if (e) cout<< e << ' ';
           // 若e为非行界值0，则打印输出 e 的值并加一空格
           else cout << endl;// 否则回车换行，为下一行输出做准备
           EnQueue(Q,s+e);
           } while (e!=0);
       k++;
 } // while
```

队列的应用举例

```
    DeQueue ( Q,e );                    // 行界值 “0”出队列
    while (!QueueEmpty (Q) )
    {                                   // 单独处理第 n 行的值的输出
        DeQueue ( Q,e );
        cout<<e<< ‘ ’;
     } // while
} // yanghui
```

- 外循环的次数为 **n-1**，内循环的次数分别为**3,4,…,n+1**，可计算出此算法的时间复杂度为***O*** **(n^2)** 。

离散事件模拟



- 3.1 栈的类型定义
- 3.2 栈的应用举例
- 3.3 队列的类型定义
- 3.4 队列的应用举例
- 3.5 离散事件模拟

离散事件模拟

- 编制一个事件驱动仿真程序以模拟理发馆内一天的活动，要求输出在一天的营业时间内，到达的顾客人数、顾客在馆内的平均逗留时间和排队等候理发的平均人数以及在营业时间内空椅子的平均数。

离散事件模拟

- 为计算出每个顾客自进门到出门之间在理发馆内逗留的时间，只需要在顾客“进门”和“出门”这两个时刻进行模拟处理即可。
- 习惯上称这两个时刻发生的事情为“事件“；
- 整个仿真程序可以按事件发生的先后次序逐个处理事件，这种模拟的工作方式称为“事件驱动模拟”；
- 程序将依事件发生时刻的顺序依次进行处理，整个仿真程序则以事件表为空而告终。
- 演示 3-4-2

离散事件模拟

- 数据类型定义：
  - // "事件"的结构定义

```
typedef struct {
    int occurTime; // 事件发生时刻
    char nType;       // 事件类型
} ElemType,Event;
```

  - // "事件表"的定义

```
tydedef  OrderedLinkList  eventList;
```

离散事件模拟

- // "顾客"的结构定义

```
typedef struct {
    int arrivalTime; // 顾客到达时间
    int duration;       // 顾客理发所需时间
} qElemType, customer;
```

- // 候理顾客队列的定义

```
typedef  Queue  waitingQueue;
```

离散事件模拟

```
void BarberShop_Simulation( int chairNum, int closeTime )
 {// 理发馆业务模拟，统计一天内顾客在馆内逗留的平均时间
  // 和顾客在等待理发时排队的平均长度以及空椅子的平均数
  // chatrNum 为假设的理发馆的规模，closeTime 为营业时间
    OpenForDay（）;                                    // 初始化
    while（ MoreEvent）
    {  EventDrived(OccurTime, EventType);          // 事件驱动
       switch (EventType) {
          case ‘A’ : CustomerArrived（）; break;      // 处理顾客到达事件
          case ‘D’ : CustomerDeparture（）; break; // 处理顾客离开事件
          default : Invalid（）;
       } // switch
    } // while
    CloseForDay（）;          // 计算平均逗留时间和排队的平均长度等
 } // BarberShop_Simulation
```

离散事件模拟

- “顾客进门”事件的处理：
  - 1．生成“本顾客理发所需时间 durtime”和 “下一顾客到达的时间间隔 intertime”两个随机数；
  - 2．若下一顾客到达的时刻没有超过营业时间，则产生一个新的“进门事件”插入事件表；
  - 3．若此时理发馆内尚有空理发椅，则空椅子数减1且产生一个新的“出门事件”插入事件表，并累计顾客逗留时间；
  - 4．否则将该“顾客”插入“候理队列”；
  - 5．累计顾客人数和排队长度。

离散事件模拟

- “顾客出门”事件的处理：
  - 1．若候理队列为空，则空椅子数增1；
  - 2．否则删除队头“顾客”，并产生一个新的“出门事件”插入事件表，且累计顾客逗留时间；
  - 3．累计空椅子数。

离散事件模拟

- void CustomerArrived(eventList evL, Queue Q, Event en)

```
{ // 处理顾客进门事件
    Random(durtime, intertime);
    nextAT = en.occurTime + intertime;     // 下一顾客到达时刻
    if (nextAT<closeTime) {
        newAEvent = ( nextAT, 'A');                  // 新的进门事件
        MakeNode(newp, newAEvent);
        LocateElem(evL, newAEvent, compare);
        Insafter(evL, newp);                         // 插入事件表
    }
```

离散事件模拟

```
if (freeChair) {                                    // 顾客即刻开始理发
        dT = en.occurTime + durtime;
        newDEvent = (dT, 'D');                      // 新的出门事件
        MakeNode(newp, newDEvent);
        LocateElem(evL, newDEvent, compare);
        Insafter(evL, newp);                        // 插入事件表
        totalTime += durtime;                       // 累计逗留时间
        --freeChair;
    }
else {                                              // 顾客排队等候
        newCustomer = (en.occuTime, durtime);
        EnQueue(Q, newCustomer);
    }
```

离散事件模拟

```
    ++customerNum;                          // 统计顾客总人数
     totalQLength += QueueLength(Q);        // 累计排队的长度
} // CustomerArrived
```

离散事件模拟

```
void CustomerDeparture(eventList evL, Queue Q, Event en)
{ // 处理顾客出门事件
   if (!DeQueue(Q, cm)) ++FreeChair;        // 无人等候理发
   else {                                   // 排头顾客出列开始理发
      dT = en.occurTime + cm.duration;
      newDEvent = (dT, 'D');                // 新的出门事件
      MakeNode(newp, newDEvent);
      LocateElem(evL, newDEvent, compare);
      Insafter(evL, newp);                  // 插入事件表
      totalTime += (dT - cm.arrivalTime); // 累计逗留时间
   } totalFreeChair += freeChair;           // 累计空椅数
} // CustomerDeparture
```

本章小结

- 在这一章我们学习了**栈**和**队列**这两种抽象数据类型。在学习过程中大家已经了解到，栈和队列都属**线性结构**，因此他们的存储结构和线性表非常类似，同时由于他们的基本操作要比线性表简单得多，因此它们在相应的存储结构中实现的算法都比较简单，相信对大家来说都不是难点。
- 这一章的**重点**则在于**栈和队列的应用**。通过本章所举的例子学习分析应用问题的特点，在算法中适时应用栈和队列。
- 离散事件模拟

本章知识点与重点

- **知识点**

  顺序栈、链栈、循环队列、链队列

- **重点和难点**

  栈和队列是在程序设计中被广泛使用的两种线性数据结构，因此本章的学习重点在于掌握这两种结构的特点，以便能在应用问题中正确使用。