数据结构

主讲：项若曦　助教：申智铭、黄毅
rxxiang@blcu.edu.cn
主楼南329

回顾

- **栈与递归**
  - 递归与形式化表达式
  - 递归与递归调用图
- **算法设计实践**
  - 单链表
    - 正、反向打印单链表
    - 创建单链表
    - 求单链表最大值（选做）求该链表数据元素个数；
    - （选做）求链表中元素的平均值；
  - 用递归求解小型迷宫

本节内容

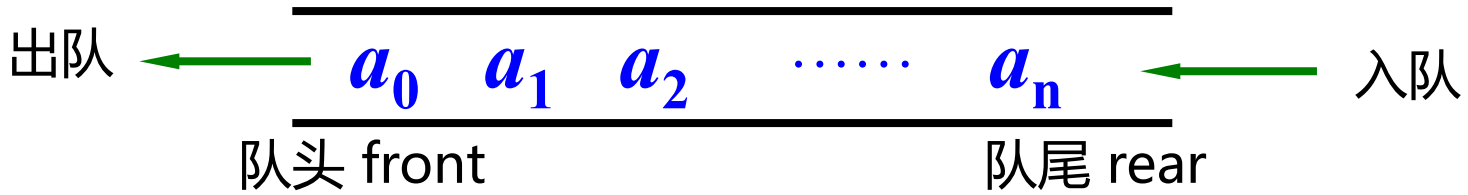
- **队列**
  - 概念、ADT
  - 各种存储结构、基本操作实现及对比
  ——链栈和顺序栈
  - 应用
  ——杨辉三角和舞伴问题

本节内容

- 概念、ADT

队列（Queue）的基本概念

- **定义：限定只允许在一端进行插入，在另一端删除元素的线性表**
  - 特殊的线性表：操作受限
    - 只允许在一端进行插入，而在另一端删除元素
    - 允许插入的一端为队尾**(rear)**,允许删除的一端为队头**(head)**
  - 逻辑特征：先进先出（FIFO，First In First Out）
- **主要操作**
  - 初始化空队、入队、出队、判断队空、判断队满、取队头



队列-ADT Queue

ADT Queue{

数据对象：D={a_i | a_i∈ElemSet, i=1,2,...,n, n≥0}

数据关系：R={R1},R1={<a_{i-1},a_i>|a_{i-1},a_i∈D, i=2,3,...,n }

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：销毁队列Q

ClearQueue(&Q)

初始条件：队列Q已存在

操作结果：将队列Q重置为空队列

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：返回队列Q中数据元素的个数

GetHead(Q,&e) **GetElem(L, i, &e)**

初始条件：队列Q已存在且非空

操作结果：用e返回Q中队头元素

EnQueue(&Q, e) **ListInsert(&L, i, 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) **ListDelete(&L, i, &e)**

初始条件：队列Q已存在且非空

操作结果：删除Q的队头元素，并用e返回其值

QueueTraverse(Q)

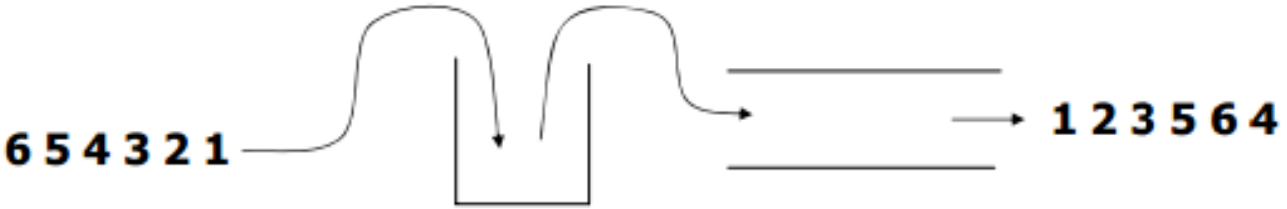
初始条件：队列Q已存在且非空

操作结果：从队头到队尾依次对Q的每个数据元素访问

}**ADT** Queue

栈与队列的概念应用

- 问题： 设栈**S**和队列 **Q**的初态均为空，将元素**1, 2, 3, 4, 5, 6**依次入栈**S**，一元素出栈后即进入队列 **Q**，若这**6**个元素出队次序是**4,6,5,3,2,1**，则栈**S**至少应能容纳多少个元素?



本节内容

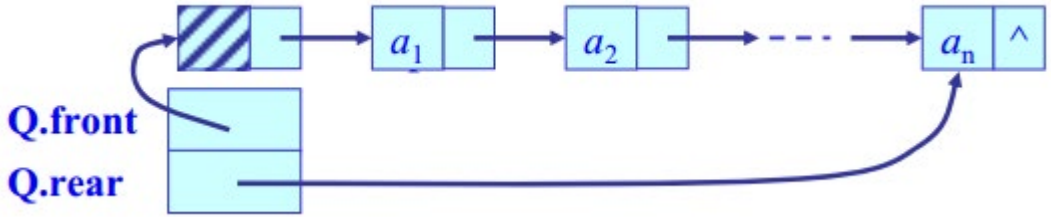
- **队列**
  - 各种存储结构、基本操作实现及对比
——顺序队列、链队列

本节内容

- 链队列

队列——链队的表示与实现（带头结点情况）

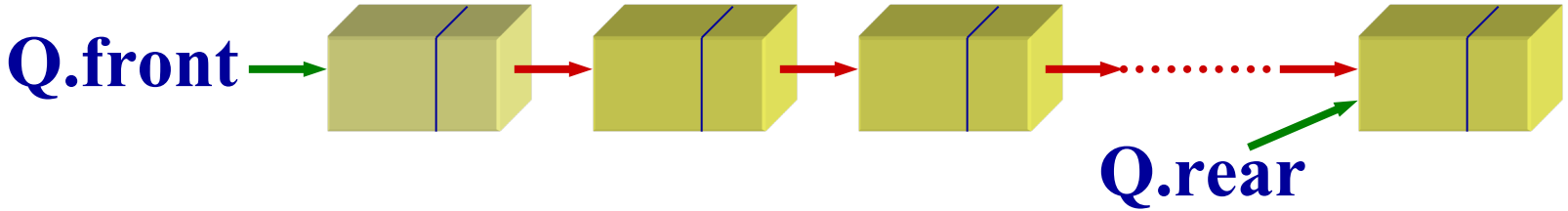
- 链队列
  - 基本操作的实现
    - 无队满问题(除非分配不出内存), 空间可扩充
    - 引入头结点(一定需要吗？)



1）引入队头指针、队尾指针：在O(1)时间内找到操作结点
2）引入头结点：队尾插入时，使队空和队不空的结点表示一致
3）队空的判断：头、尾指针均指向头结点
4）队满的判断转变成对申请空间是否成功的判断。

队列——链队的表示与实现（无头结点情况）

- 链队列
  - 基本操作的实现
    - 无队满问题(除非分配不出内存), 空间可扩充
    - 若无头结点



1）引入队头指针、队尾指针：在O(1)时间内找到操作结点
2）没有引入头结点：队尾插入时，队空和队不空**要单独判断!**
3）队空的判断：**头==NULL**
4）队满的判断转变成对申请空间是否成功的判断。

队列的存储表示——链队列

链队列

- 约定与类型定义： **Q.front和Q.rear的含义**

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

typedef struct {
                QueuePtr front; /* 队头指针，指向头元素*/
                QueuePtr rear;  /* 队尾指针，指向队尾元素*/
}LinkQueue;
```

队列——链队的表示与实现

- 类型说明
  - 操作实现：初始化、队空、入队、出队、看队头、……

注意：队空的判断、入队、出队依赖于队列的表示及其约定。
进一步考虑：若无头结点，此时队列的初始化、入队、队空的条件、出队等如何表示与实现？

队列——链队的表示与实现(带头结点)


```
1. Status InitQueue(LinkQueue& Q) {//P73算法3.16
2.         Q.rear = Q.front = new QNode;
3.         Q.front->next = NULL;
4.         return OK;
5. }
```

```
1. Status QueueEmpty(const LinkQueue& Q){
2.         return Q.front == Q.rear;
3. }
```

队列——链队的表示与实现(带头结点)


```
1.  Status EnQueue(LinkQueue& Q, QElemType e) {//入队P74算法3.17
2.          QueuePtr p = new QNode;
3.          p->data = e;   p->next = NULL;
4.          Q.rear->next = p;                    //接在链尾后
5.          Q.rear = p;
6.          return OK;
7.  }
```

队列——链队的表示与实现(带头结点)


```
1.  Status DeQueue(LinkQueue& Q, QElemType& e) {//出队P75算法3.18
2.          if (QueueEmpty(Q)) return ERROR;
3.          QueuePtr p = Q.front->next;
4.          e = p->data;
5.          Q.front->next = p->next;
6.          if (Q.rear == p)
7.                      Q.rear = Q.front;
8.          delete p;
9.          return OK;
10.}
```

队列——链队的表示与实现(带头结点)


```
1.  Status GetHead(const LinkQueue& Q, QElemType& e) {//P75算法3.19
2.          if (QueueEmpty(Q))
3.                  return ERROR;
4.          e = Q.front->next->data;
5.          return OK;
6.  }
```

队列——链队的表示与实现(无头结点)

- 类型说明
  - 操作实现：初始化、队空、入队、出队、看队头、……

注意：队空的判断、入队、出队依赖于队列的表示及其约定。
进一步考虑：若无头结点，此时队列的初始化、入队、队空的条件、出队等如何表示与实现?

队列——链队的表示与实现(无头结点)


```
1.  Status InitQueue(LinkQueue& Q) {
2.          Q.rear = Q.front = NULL;
3.          return OK;
4.  }
```

```
1.  bool QueueEmpty(const LinkQueue& Q) {
2.          return Q.front == NULL;
3.  }
```

```
1.  Status GetHead(const LinkQueue& Q, QElemType& e) {
2.          if (QueueEmpty(Q))
3.                  return ERROR;
4.          e = Q.front->data;
5.          return OK;
6.  }
```

队列——链队的表示与实现(无头结点)


```
1.  Status EnQueue(LinkQueue& Q, QElemType e) {
2.          QueuePtr p = new QNode;
3.          p->data = e;   p->next = NULL;
4.          if (Q.front == NULL)                          //空
5.                  Q.front = Q.rear = p;                 //创建第一个结点
6.          else {                                        //不空
7.                  Q.rear->next = p;                     //接在链尾后
8.                  Q.rear = p;
9.          }
10.         return OK;
11.}
```

队列——链队的表示与实现(无头结点)


```
1.  Status DeQueue(LinkQueue& Q, QElemType& e) {
2.          //删去队头结点，并返回队头元素的值
3.          if (QueueEmpty(Q))
4.                  return ERROR;                   //判队空
5.          QueueNode* p = Q.front;
6.          e = p->data;                            //保存队头的值
7.          Q.front = Q.front->next;                //新队头
8.          if (Q.front == NULL)
9.                  Q.rear = NULL;
10.         delete p;
11.         return OK;
12.}
```

本节内容

- 顺序队列

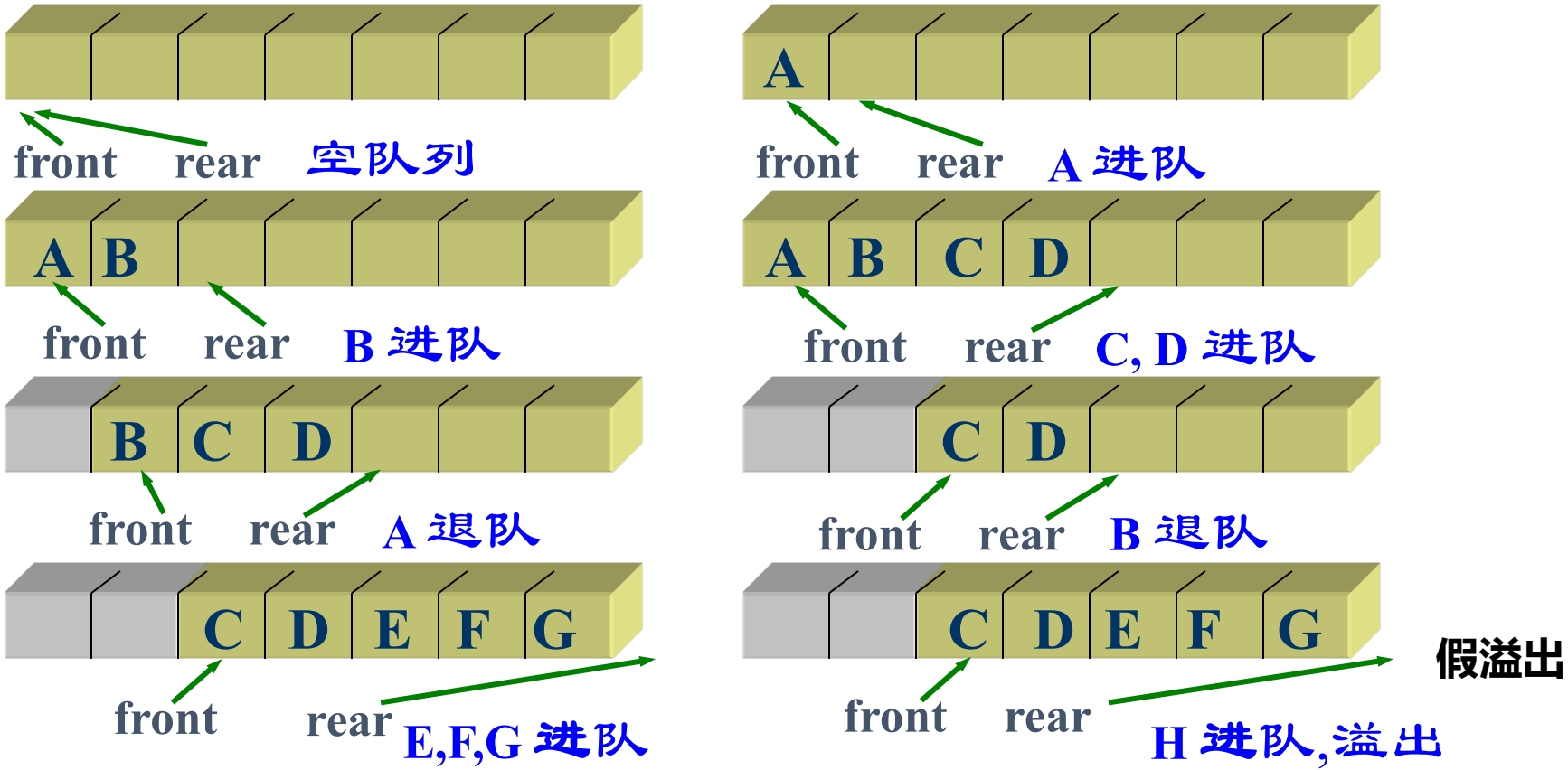
队列——顺序队列

- 队列的顺序存储
  - 队列的顺序存储
    - 约定与类型定义： **Q.front**和**Q.rear**的含义：**约定**front指向队列头元素，rear 指向队尾元素的下一位置。

```
#define MAXQSIZE 100          /* 最大队列长度 */
typedef struct{
        QElemType *base;      /* 存储空间 */
        int front;            /* 头指针，指向队列的头元素 */
        int rear;             /* 尾指针，指向队尾元素的下一个位置 */
}SqQueue;                     /* 非增量式的空间分配 */
```

    - 删除：避免大量的移动->头指针增**1**

队列——顺序队列


front
rear
空队列
A
front
rear
A 进队
A B
front
rear
B 进队
A B C D
front
rear
C, D 进队
B C D
front
rear
A 退队
C D
front
rear
B 退队
C D E F G
front
rear
E,F,G 进队
C D E F G
front
rear
H 进队,溢出
假溢出

队列——循环队列

- 操作实现

1. 空队：Q.front=Q.rear=0;
2. 入队：判断是否队满，非队满时，Q.rear位置放新插入的元素，Q.rear++
3. 出队：判断是否队空，非队空时，Q.front位置为待删除的元素，Q.front++
4. 队空条件：Q.front == Q.rear
5. 队满条件：Q.rear == MAXQSIZE-1

   问题：存在假上溢(由于出队操作，队列空间的下部可能存在空闲空间)

队列——循环队列

- 循环队列
  - 队列的顺序存储
    - 约定与类型定义： **Q.front**和**Q.rear**的含义：**约定**front指向队列头元素，rear 指向队尾元素的下一位置。

```
#define MAXQSIZE 100            /* 最大队列长度 */
typedef int QElemType;
typedef struct{
        QElemType *base;        /* 存储空间 */
        int front;              /* 头指针，指向队列的头元素 */
        int rear;               /* 尾指针，指向队尾元素的下一个位置 */
}SqQueue;                       /* 非增量式的空间分配 */
```

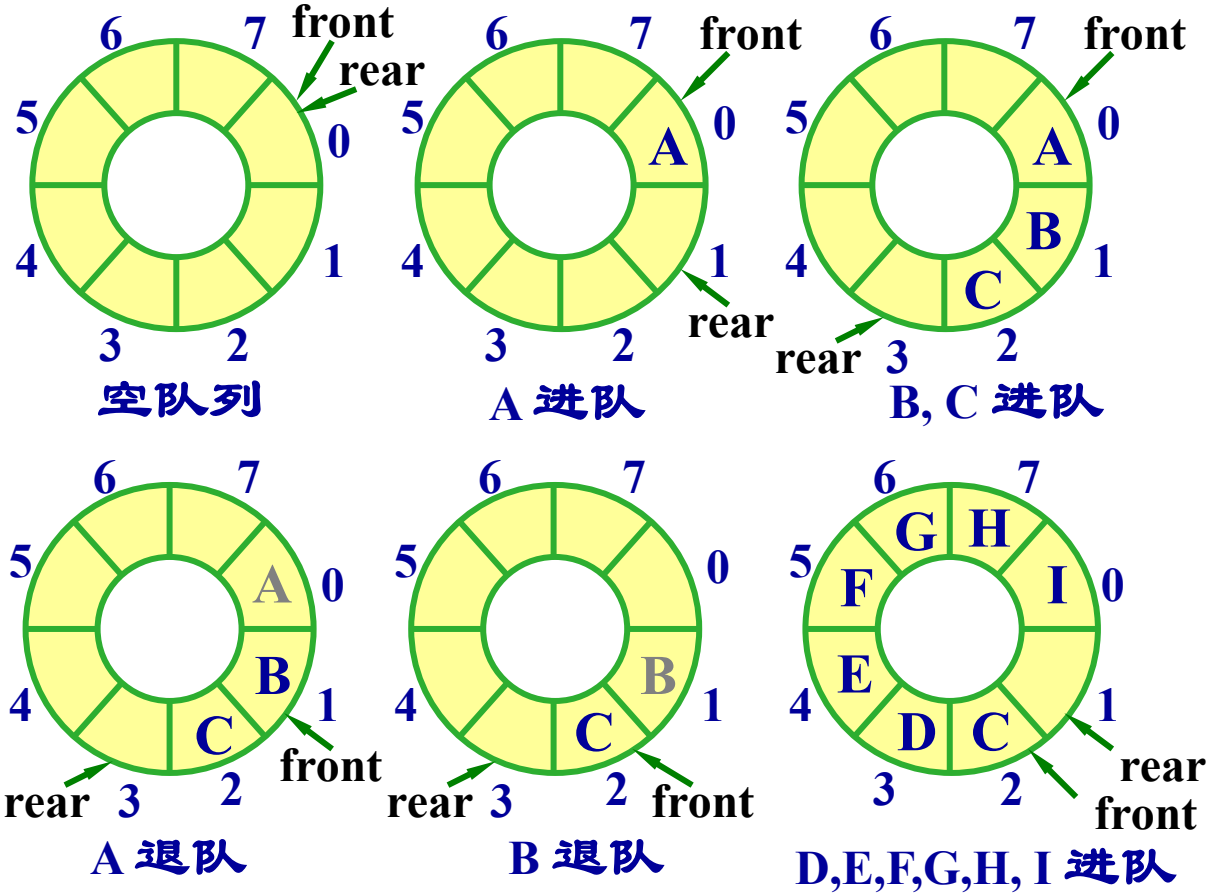
    - 删除：避免大量的移动->头指针增**1**

队列——循环队列

- 假上溢的解决: 将队列假想为首尾相接的环，即循环队列。
- 队头、队尾指针加 1 时从 QueueSize-1 直接进到0，可用语言的取模（%）运算实现。
- 注意，进队和出队时指针都是顺时针前进。

1. 空队：Q.front=Q.rear=0;（初始化）
2. 入队：……，Q.rear = (Q.rear+1)%MAXQSIZE
3. 出队：……，Q.front = (Q.front+1)%MAXQSIZE
4. 队空条件：Q.front == Q.rear，由于出队Q.front追上了Q.rear
5. 队满条件：Q.front == Q.rear，由于入队Q.rear追上了Q.front

约定Q.front（实际队头）和Q.rear(队尾的下一个)


6
7
front
rear
5
0
4
1
3
2
空队列
A
A 进队
B
C
B, C 进队
A 退队
B 退队
G
H
F
I
E
D
D,E,F,G,H, I 进队

队列——循环队列

- 如何区分队空和队满

有维护代价

1. 设标志位：不足在于需要额外对标志位的判断及维护：0-创建/删除，1-插入
2. 在队列的结构中引入长度成员，在初始化队列、入队、出队操作中维护这个成员。
3. 少用一个元素空间，即队满的条件如下

   (Q.rear+1)% MAXQSIZE == Q.front

   插入前判断**Q.front==(Q.rear+1)%MAXQSIZE**

队列——循环队列

- **循环队列类型的模块声明：P.71，**

Status InitQueue(SqQueue &Q);

bool QueueEmpty(const SqQueue&Q);

bool QueueFull(const SqQueue&Q);

int QueueLength(SqQueue Q);

Status EnQueue(SqQueue &Q, QElemType e);

Status DeQueue(SqQueue &Q, QElemType &e);

Status GetHead(const SqQueue& Q, ElemType&e);

循环队列的实现——InitQueue、QueueLength


```
1.  Status InitQueue(SqQueue& Q) {//初始化队，P71算法3.11
2.          Q.base = new QElemType[MAXQSIZE];
3.          if (!Q.base)exit(OVERFLOW);
4.          Q.front = Q.rear = 0;
5.          return OK;
6.  }
```

```
1.  int QueueLength(SqQueue Q) {//
2.          return(Q.rear - Q.front + MAXQSIZE) % MAXQSIZE;
3.  }
```

循环队列的实现——QueueEmpty、QueueFull、GetFront


```
1. bool QueueEmpty(const SqQueue& Q) {
2.         return Q.rear == Q.front;
3. }
```

```
1. bool QueueFull(const SqQueue& Q) {
2.         return (Q.rear + 1) % MAXQSIZE == Q.front;
3. }
```

```
1. Status GetHead(const SqQueue& Q, QElemType& e) {//P73 算法3.15
2.         if (QueueEmpty(Q)) return ERROR;
3.         e = Q.base[Q.front];    return OK;
4. }
```

循环队列的实现——EnQueue、DeQueue


```
1. Status EnQueue(SqQueue& Q, QElemType e) {//P72算法3.13
2.         if (QueueFull(Q))return ERROR;
3.         Q.base[Q.rear] = e;
4.         Q.rear = (Q.rear + 1) % MAXQSIZE;
5.         return OK;
6. }
```

```
1. Status DeQueue(SqQueue& Q, QElemType& e) {//P72算法3.14
2.         if (QueueEmpty(Q))return ERROR;
3.         e = Q.base[Q.front];
4.         Q.front = (Q.front + 1) % MAXQSIZE;
5.         return OK;
6. }
```

队列——循环队列（自己回去思考）

- 难点：连续的存储单元的上下界：**[d1,d2]**
  - 初始化空队：Q.front = Q.rear = d1;
  - 队空：Q.front==Q.rear
  - 队满：Q.front==(Q.rear-d1+1)%(d2-d1+1)+d1
  - 入队： 前提：队列不满
    Q.base[Q.rear] = e;
    Q.rear = (Q.rear-d1+1)%(d2-d1+1)+d1;
  - 出队：前提：队列不空
    e = Q.base[Q.front];
    Q.front = (Q. front-d1+1)%(d2-d1+1)+d1

本节内容

- **应用**
  - 舞伴问题
  - 杨辉三角

- 舞伴问题

队列的应用——舞伴问题

- **案例分析**
  - 设置两个队列分别存放男士和女士入队者。
  - 假设男士和女士的记录存放在一个数组中作为输入，然后依次扫描该数组的各元素，并根据性别来决定是进入男队还是女队。
  - 当这两个队列构造完成之后，依次将两队当前的队头元素出队来配成舞伴，直至某队列变空为止。
  - 此时，若某队仍有等待配对者，则输出此队列中排在队头的等待者的姓名，此人将是下一轮舞曲开始时第一个可获得舞伴的人。

舞伴问题

- **数据结构**

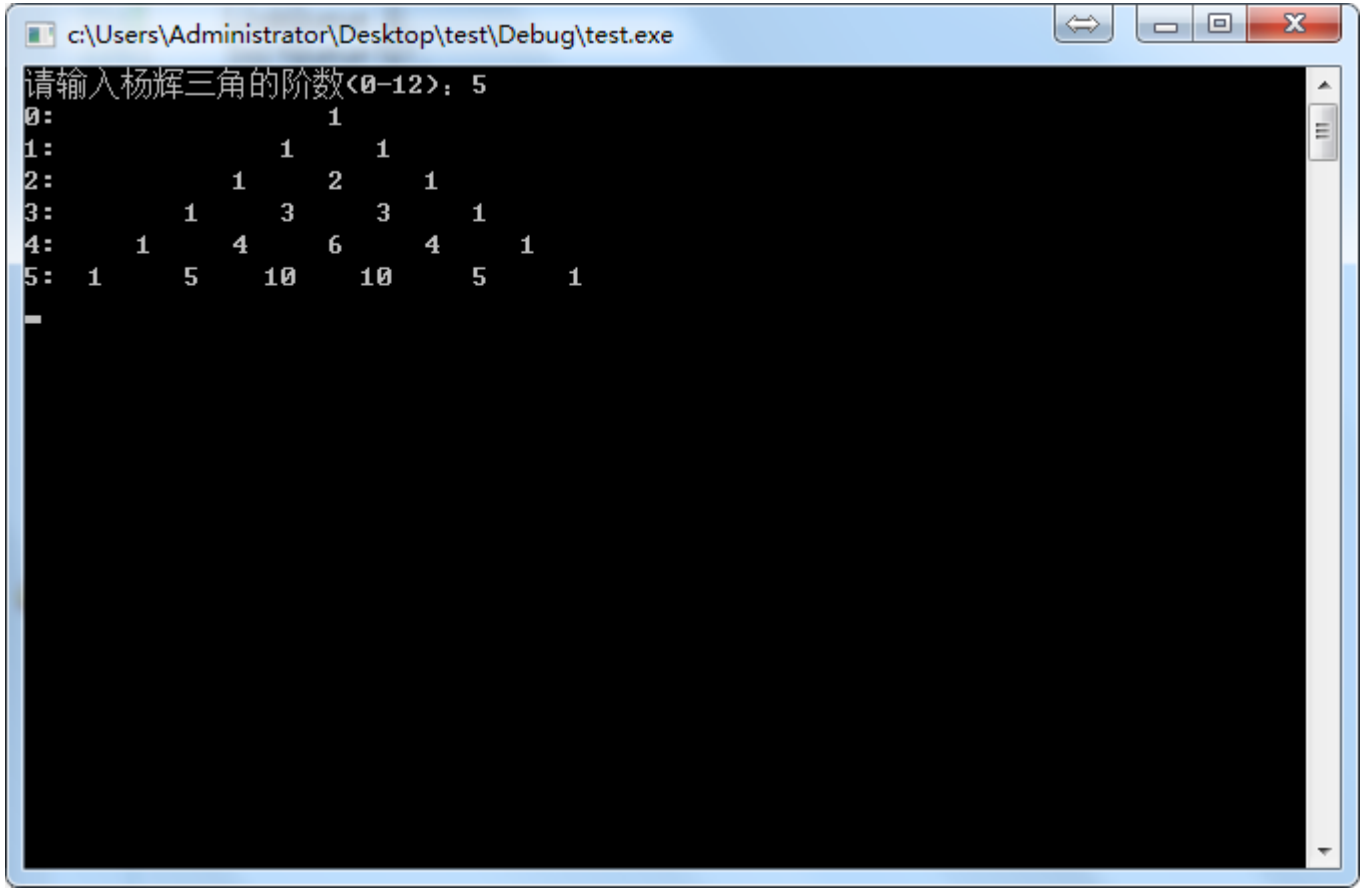
```
//- - - - - 跳舞者个人信息- - - - -
typedef struct{
        char name[20];                  //姓名
        char sex;                       //性别，'F'表示女性，'M'表示男性
}Person;
//- - - - - 队列的顺序存储结构- - - - -
#define MAXQSIZE 100                    //队列可能达到的最大长度
typedef struct{
        Person* base;                   //队列中数据元素类型为Person
        int front;                      //头指针
        int rear;                       //尾指针
}SqQueue;
SqQueue Mdancers, Fdancers;             //分别存放男士和女士入队者队列
```

舞伴问题

- **算法步骤**
  - ① 初始化Mdancers队列和Fdancers队列。
  - ② 反复循环，依次将跳舞者根据其性别插入Mdancers队列或Fdancers队列。
  - ③ 当Mdancers队列和Fdancers队列均为非空时，反复循环，依次输出男女舞伴的姓名。
  - ④ 如果Mdancers队列为空而Fdancers队列非空，则输出Fdancers队列的队头女士的姓名。
  - ⑤ 如果Fdancers队列为空而Mdancers队列非空，则输出Mdancers队列的队头男士的姓名。

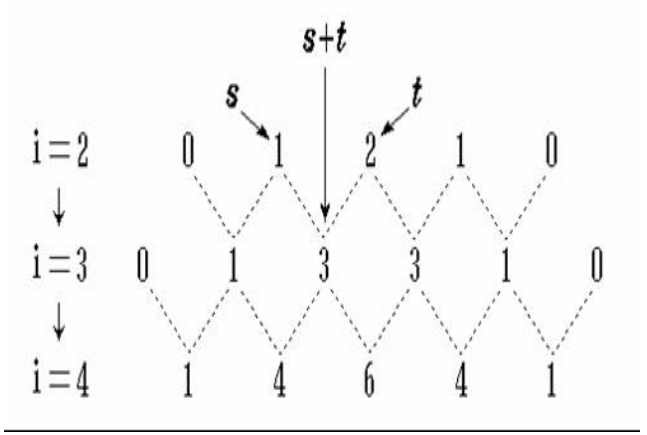
- 杨辉三角

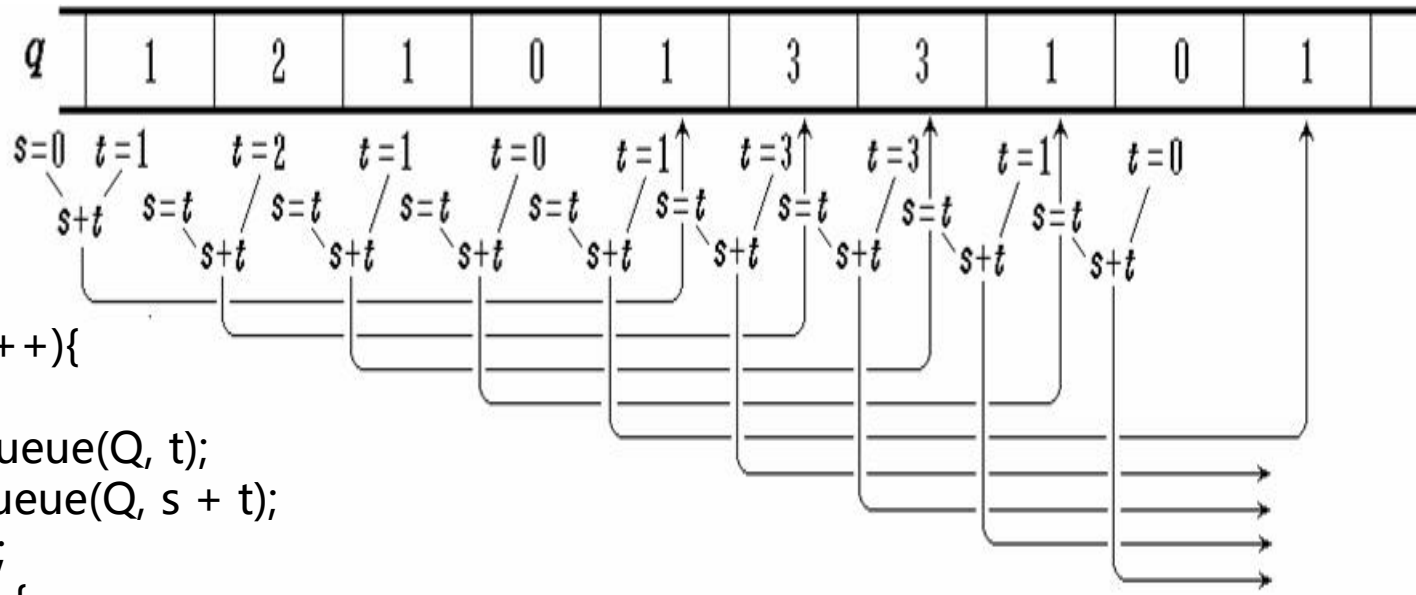
队列的应用——杨辉三角


c:\Users\Administrator\Desktop\test\Debug\test.exe
请输入杨辉三角的阶数(0-12): 5
0: 1
1: 1 1
2: 1 2 1
3: 1 3 3 1
4: 1 4 6 4 1
5: 1 5 10 10 5 1

队列的应用——杨辉三角

		1	0			i = 0	
		1	1	0		1	
		1	2	1	0	2	
		1	3	3	1	0	3
	1	4	6	4	1	0	4
1	5	10	10	5	1	0	5






```
1.  int s = 0;
2.  EnQueue(Q, 1);
3.  EnQueue(Q, 0);
4.  for (int i = 0; i < n; i++){
5.          while (1) {
6.                  DeQueue(Q, t);
7.                  EnQueue(Q, s + t);
8.                  s = t;
9.                  if (!t) {
10.                         EnQueue(Q, 0);
11.                         break;
12.                 }
13.         }
14.         cout << endl;
15. }
```

本节内容

- 优先队列(*)

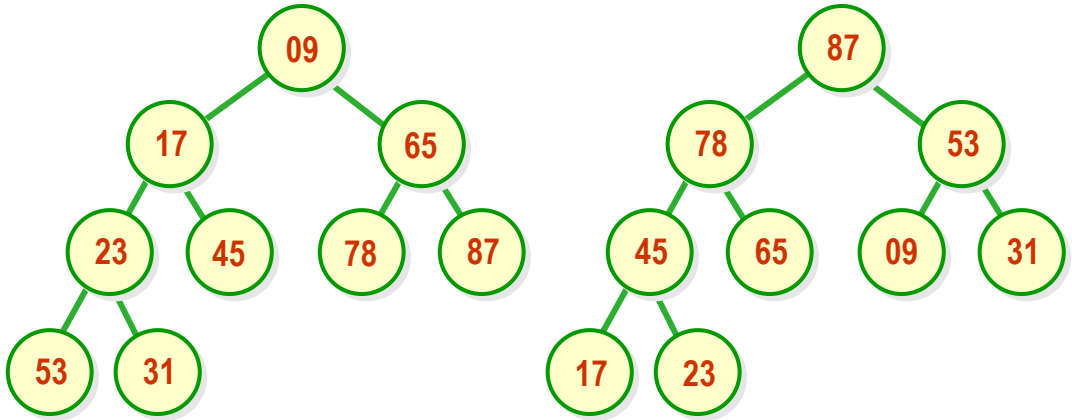
优先队列(priority_queue)

- **优先队列**
  - 每次从队列中取出的是具有最高优先权的元素
  - 任务优先权及执行顺序的关系

任务编号	1	2	3	4	5
优先权	20	0	40	30	10
执行顺序	3	1	5	4	2

数字越小，优先权越高

- **优先队列的实现**
  - 堆



哪种堆最好？

Dijkstra 算法的运行时间严重依赖于优先队列的所用的实现方法。以下是几种典型的选择：

实现方式	deletemin	insert/ decreasekey	$\|V\|$ × deletemin + ($\|V\|+\|E\|$) × insert
数组	$O(\|V\|)$	$O(1)$	$O(\|V\|^2)$
二分堆	$O(\log \|V\|)$	$O(\log \|V\|)$	$O((\|V\|+\|E\|)\log \|V\|)$
d 堆	$O\left(\frac{d \log \|V\|}{\log d}\right)$	$O\left(\frac{\log \|V\|}{\log d}\right)$	$O\left((\|V\| \cdot d + \|E\|)\frac{\log \|V\|}{\log d}\right)$
Fibonacci 堆	$O(\log \|V\|)$	$O(1)$(平摊后)	$O(\|V\| \log \|V\| + \|E\|)$

作业预告

- **队列的实现**
  - 链队列（带、无头结点）
  - 循环队列
- **杨辉三角**
- **舞伴问题**

预告

第四章 串和数组（1）串

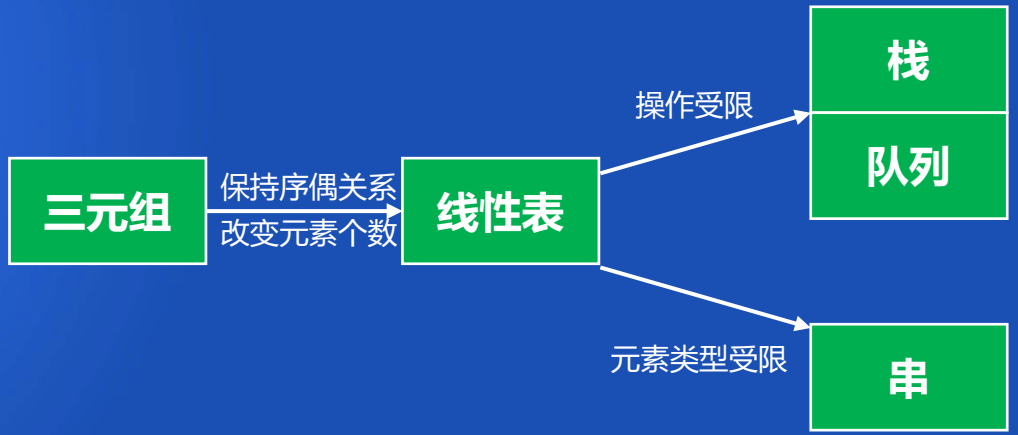
- **4.1 串的定义**
- **4.2 案例引入**
- **4.3 串的类型定义、存储及运算**
  - 串抽象数据类型定义（ADT）
  - 串的存储结构
  - 串的模式匹配算法

重点：串的定义、表示与实现，模式匹配
难点：匹配算法

回顾

ADT=(D, S, P)

- **D** — 数据对象,数据元素的有限集(元素个数、类型等)
- **S** — 是D上关系的有限集
- **P** — 是对D的基本操作



回顾

C语言中的字符串——以“\0”结尾

- **字符串初始化**

1. char name[12] ="Jisuanji";
2. char name[ ] ="Jisuanji";
3. char name[12] = {'J','i','s','u','a','n','j','i'};
4. char name[ ] = {'J','i','s','u','a','n','j','i', '\0'};
5. char *name = "Jisuanji";
6. char name[12];
   name = "Jisuanji"; × 因数组名是地址常量

```
1>1.cpp
1>c:\users\administrator\desktop\1\1.cpp(10) : error C2440: '=' : cannot convert from 'const char [9]' to 'char [12]'
1>        There is no context in which this conversion is possible
1>Build log was saved at "file://c:\Users\Administrator\Desktop\1\Debug\BuildLog.htm"
1>1 - 1 error(s), 0 warning(s)
========== Build: 0 succeeded, 1 failed, 0 up-to-date, 0 skipped ==========
```

回顾

- **单个字符串的输入函数 gets (str)**

  例 char name[12];

  gets (name);

- **多个字符串的输入函数 scanf**

  例 char name1[12], name2[20];

  scanf (“%s%s\n” , name1, name2);

- **字符串输出函数 puts (str)**

  例 char name[12];

  gets(name);

  puts(name);

- **字符串求长度函数 strlen(str)**

  字符串长度不包括“\0”和分界符

回顾

- **字符串连接函数 strcat (str1, str2)**

  例 str1　“Jisuanji\0”　　//连接前
  　str2　“Good\0”　　//连接前
  　str1　“JisuanjiGood\0”　　//连接后
  　str2　“Good\0”　　//不变

- **字符串比较函数 strcmp (str1, str2)**

  //从两个字符串第 1 个字符开始，逐个对
  //应字符进行比较，全部字符相等则函数返
  //回0，否则在不等字符处停止比较，函数
  //返回其差值 — ASCII代码

  例 str1　“University”　　i 的代码值105
  　str2　“Universal”　　a 的代码值97，差8

回顾

- C++:String 类

本节内容

- 串类型的定义、ADT、操作

4.1 串类型的定义

- **串 P.87.**
  - 一般地，串是有零个或多个字符组成的有限序列
    - C中的字符串：一对双引号括起来的字符序列，如“abcd ef”
  - 一些概念：
    - 串的长度、空串、子串、主串、字符/子串在串/主串中的位置、空格串 v.s. 空串Ø
      - 如：S=“Data Structure”，S为串名，“Data Structure”为串值长度为14；“ata Str”为主串S的子串，它在S中的位置为2。
      - 称两个串是相等的，当且仅当这两个串的值相等。
  - 串是特殊的线性表
    - 1)元素类型为字符；
    - 2)操作对象：个体(字符)与整体(子串)

4.1 串类型的定义 – ADT String

ADT String{

数据对象：D={a_i|a_i∈**CharacterSet**, i=1,2,...,n, n≥0}

数据关系：R={R1},R1={<a_{i-1},a_i>|a_{i-1},a_i∈D, i=2,3,...,n }

基本操作：

StrAssign(&T, chars)

初始条件：chars是字符串常量

操作结果：生成一个其值等于chars的串T

StrCopy(&T, S)

初始条件：串S存在

操作结果：由串S复制得串T

StrEmpty(S)

初始条件：串S存在

操作结果：若S为空串，则返回TRUE，否则返回FALSE

StrCompare(S, T)

初始条件：串S和T存在

操作结果：若串S>T，则返回值>0；若串S=T，则返回值=0；若串S<T，则返回值<0

StrLength(S)

初始条件：串S已存在

操作结果：返回串S中数据元素的个数

ClearString(&S)

初始条件：串S已存在

操作结果：将串S清为空串

Concat(&T, S1, S2)

初始条件：串S1和S2存在

操作结果：用T返回由S1和S2联接而成的新串

SubString(&Sub, S, pos, len)

初始条件：串S存在，1 ≤ pos ≤ StrLength(S)且0 ≤ len ≤ StrLength(S)-pos+1

操作结果：用Sub返回串S的第pos个字符起长度为len的子串

Index(S, T, pos)

初始条件：串S和T存在，T非空，1 ≤ pos ≤ StrLength(S)

操作结果：若主串S中存在和串T值相同的子串，则返回它在主串S中第pos个字符之后第一次出现的位置；否则函数值为0

Replace(&S, T, V)

初始条件：串S，T和V存在，T是非空串

操作结果：用V替换主串S中出现的所有与T相等的不重叠的子串

StrInsert(&S, pos, T)

初始条件：串S和T存在，1 ≤ pos ≤ StrLength(S)+1

操作结果：在串S的第pos个字符之前插入串T

StrDelete(&S, pos, len)

初始条件：串S存在，1 ≤ pos ≤ StrLength(S)-len+1

操作结果：从串S中删除第pos个字符起长度为len的子串

DestroyString(&S))

初始条件：串S存在

操作结果：串S被销毁

}**ADT** String

4.1 串类型的定义 – ADT String

ADT String 串的整体操作

- 赋值StrAssign(S, “Data Structure”)
- 复制StrCopy(T, S) // T<=S, T为 “Data Structure”
- 比较StrCompare(S, T)
- 连接Concat(T, “Data”, “Structure”) //T为 “DataStructure”
- 取子串SubString(sub, S, 2, 5) // sub为 “ata S”
- 子串在主串中的定位Index(S, “a”, 3) // 4
- 子串置换Replace(S, “a”, “b”) // S为 “Dbtb Structure”
- 子串插入StrInsert(S, 3, “aha”) // “Daahata Structure”
- 子串删除StrDelete(S, 3, 5) // “Daructure”

串的最小操作子集

- 1) 赋值操作 StrAssign(&T, chars)
- 2) 比较函数 StrCompare(T, S)
- 3) 求串长函数 StrLength(S)
- 4) 联接函数 Concat(&T, S1, S2)
- 5) 求子串函数 SubString(&Sub, S, pos, len)

其他操作可在此最小操作集上实现。

- **串的最小操作子集**
  - 赋值、求串长、比较、联接、取子串
- **模式匹配(定位函数) Index（S， T， pos)**

```
1.  int Index(String S, String T, int pos) {
2.      if (pos>0){ // pos的合法性
3.          n = StrLength(S); m = StrLength(T); i = pos;
4.          while( i<=n-m+1) {
5.              SubString(sub, S, i, m);
6.              if (StrCompare(sub, T) != 0) ++i;
7.              else return i;                          // 返回子串在主串中的位置
8.          }                                           // while
9.      }                                               // if
10.     return 0;                                       // S中不存在与T相等的子串
11. }                                                   // Index
```

预告

- **串的表示及实现**

——三种表示方法，对应操作实现的例子