

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（实现、应用、栈与递归）**
- **队列的实现**
 - 链队列（带头结点版本、无头结点版本）
 - 循环队列（顺序队）
- **队列的应用**
 - 杨辉三角
 - 舞伴问题

本章内容

第四章 串和数组 (1) 串

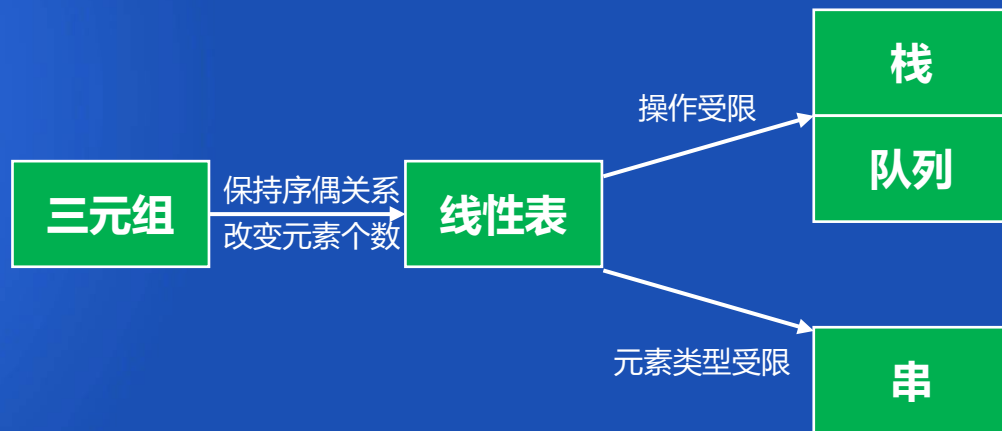
- 4.1 串的定义
- 4.2 案例引入
- 4.3 串的类型定义、存储及运算
 - 串抽象数据类型定义 (ADT)
 - 串的存储结构
 - 串的模式匹配算法

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. `const 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

展望

► C++:String 类

本节内容

- 串类型的定义、ADT、操作

➤ 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 \emptyset
 - 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: $D = \{a_i | a_i \in \text{CharacterSet}, i = 1, 2, \dots, n, n \geq 0\}$

数据关系: $R = \{R_1\}, R_1 = \{ \langle a_{i-1}, a_i \rangle | a_{i-1}, a_i \in D, i = 2, 3, \dots, n \}$

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, $1 \leq \text{pos} \leq \text{StrLength}(S)$ 且 $0 \leq \text{len} \leq \text{StrLength}(S) - \text{pos} + 1$

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, $1 \leq \text{pos} \leq \text{StrLength}(S)$

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, $1 \leq \text{pos} \leq \text{StrLength}(S) + 1$

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, $1 \leq \text{pos} \leq \text{StrLength}(S) - \text{len} + 1$

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

➤ ADT String 串的整体操作

- ▶ 赋值StrAssign(S, "Data Structure")
- ▶ 复制StrCopy(T, S)
- ▶ 比较StrCompare(S, T)
- ▶ 连接Concat(T, "Data", "Structure")
- ▶ 取子串SubString(sub, S, 2, 5)
- ▶ 子串在主串中的定位Index(S, "a", 3)
- ▶ 子串替换Replace(S, "a", "b")
- ▶ 子串插入StrInsert(S, 3, "aha")
- ▶ 子串删除StrDelete(S, 3, 5)

// T<=S, T为 "Data Structure"

//T为 "DataStructure"

// sub为 "ata S"

// 4

// S为 "Dbtb Structure"

// "Daahata Structure"

// "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。

- ▶ 赋值、求串长、比较、联接、取子串

```

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

```

本节内容

- **串表示及实现**
——三种表示方法，对应操作实现的例子

➤ 串的实现

- ▶ 顺序存储
 - 串的定长顺序存储表示
 - 串的堆式存储表示动态分配
- ▶ 链式存储
 - 串的块链存储表示——链式映像



➤ 定长顺序存储表示

▶ 定长顺序存储表示----顺序映像

▶ 类型定义

```
#define MAXSTRLEN 255
```

```
typedef unsigned char SString[MAXSTRLEN+1];
```

- 约定：1) 下标为0的分量存放串的长度

局限性：对于MAXSTRLEN超过255的串，此存储结构不适合，需要修改。

或2)串值后加入一个不计入串长的结束标记字符，如C语言中的'\0'

不足：串长是隐含的，需要遍历整个串才能得出。

▶ 串联接**Concat(&T, S1, S2)**

- ∴是定长存储，联接后T的串长为S1和S2串长之和，该长度可能会超出MAXSTRLEN
- ∴分情况处理，超出部分要“截断”

4.2 串的实现和实现-定长顺序存储



▶ 求子串**SubString(&Sub, S, pos, len) P.74.**

- 合法的pos和len

$\text{pos} > 0 \ \&\& \ \text{len} \geq 0 \ \&\& \ \text{pos} \leq \text{S}[0] - \text{len} + 1$

- 串的复制

$\text{Sub}[1..\text{len}] = \text{S}[\text{pos}..\text{pos} + \text{len} - 1];$

$\text{Sub}[0] = \text{len};$

```
1. Status SubString(SString Sub, SString S,int pos,int len){
2.     if(pos<1 || pos>S[0] || len<0 || len>S[0]-pos+1)
3.         return ERROR;
4.     for(i = 1;i<=len;i++)
5.         Sub[i] = S[pos+i-1];
6.     Sub[0] = len;
7.     return OK;
8. }
```

4.2 串的实现和实现-定长顺序存储



- ▶ 评价
 - 串长超出最大长度，约定采用截尾法
 - 串长过小，则串空间浪费较大



➤ 堆式存储表示

➤ 堆式存储表示----顺序映像

▶ 类型定义

```
typedef struct {  
    char *ch;           // 串值所在的存储区的起始地址  
    int length;         // 串长度  
} HString;
```

➤ 堆存储表示-----顺序映像

▶ 一些操作的实现

◦ 基于复制

利用malloc() or new 分配一块足够的空间, 再按要求完成复制; 如
StrAssign(&T,chars), Concat(&T, S1, S2), SubString(&Sub, S, pos, len)

利用realloc()重新分配空间, 如StrInsert(&S, pos, T)

4.2 串的实现-堆式存储



举例：串的插入算法，其他操作请参见红皮课本P76-77

```
1. Status StrInsert(HString &S, int pos, HString T){
2.     //1<=pos<=StringLength(S)+1。在串S的第pos个字符之前插入串T
3.     if(pos<1 || pos>StringLength(S)+1)
4.         return ERROR;//pos不合法
5.     if(T.length) { //T非空，则重新分配空间，插入T
6.         if(!(S.ch = (char *) realloc(S.ch, (S.length+T.length)*sizeof(char))))
7.             exit(OVERFLOW);
8.         for(i=S.length-1; i>=pos-1; --i) //为插入T而腾出位置
9.             S.ch[i+T.length] = S.ch[i];
10.        for(i=pos-1; i<=pos+T.length-2; i++) //插入T
11.            S.ch[i] = T.ch[j++];
12.        S.length +=T.length;
13.    }
14.    return OK;
15.}
```

➤ 堆式存储表示----顺序映像

▶ 评价

- 基于动态存储管理
- 处理方便、对串长没有限制



➤ 块链存储表示

➤ 块链表示----链式映像

▶ 类型定义

```
#define CHUNKSIZE 80
typedef struct Chunk {
    char ch[CHUNKSIZE];
    struct Chunk *next;
} Chunk;
typedef struct {
    Chunk *head, *tail; // 串的头和尾指针
    int curlen; // 串的当前长度
} LString;
```

4.2 串的实现和实现-块链存储（续）



- ▶ $\text{sizeof}(\text{char}) < \text{sizeof}(\text{链域}) \rightarrow$ 存储密度不高
存储密度 = 串值所占的存储位 / 实际分配的存储位
- ▶ 存储密度的影响
存储密度小，运算处理方便，但存储占用量大
- ▶ 评价
 - 存储密度较低
 - 块链使串的操作复杂化

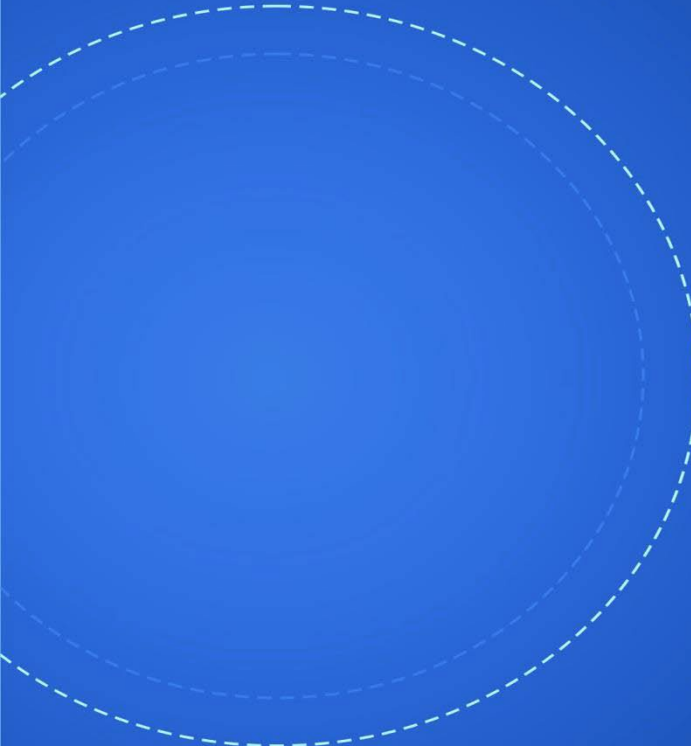
➤ 顺序映像和块链映像的缺点

▶ 顺序存储:

- 1) 对于定长顺序存储空间利用率低;
- 2) 对于定长顺序存储空间不可扩充, 使串的某些操作 (联接、置换) 受到限制;
- 3) 插入、删除的移动。

▶ 块链存储:

- 1) 存储密度较低;
- 2) 块链使串的操作复杂化。

- 
- **重难点：串的模式匹配**
 - BF算法
 - KMP算法（自学）

4.3 串的模式匹配算法 (BF算法)

S="ababcbabcac" T="abcac"
pos=1

index(S,T,pos)返回值为6

第一趟匹配

↓ i=3
a b a b c a b c a c b a b
a b c
↑ j=3

第二趟匹配

↓ i=2
a b a b c a b c a c b a b
a
↑ j=1

第三趟匹配

↓ i=7
a b a b c a b c a c b a b
a b c a c
↑ j=5

4.3 串的模式匹配算法 (BF算法)



北京语言大学
BEIJING LANGUAGE AND CULTURE UNIVERSITY

S="ababcabcac" T="abcac"
pos=1

index(S,T,pos)返回值为6

第四趟匹配

↓ i=4
a b a b c a b c a c b a b
↑ a j=1

第五趟匹配

↓ i=5
a b a b c a b c a c b a b
↑ a j=1

第六趟匹配

↓ i=11
a b a b c a b c a c b a b
a b c a c ↑ j=6

匹配成功

4.3 串的模式匹配算法 (BF算法)



求子串位置的定位函数Index(S, T, pos)——模式匹配(T:模式串)

基于堆分配存储的算法

```
1. int Index( SString S, SString T, int pos ){
2.     i = pos; j = 1;
3.     while ( i <= S[0] && j <= T[0] ){
4.         if ( S[i] == T[j] )
5.             { i++; j++; } //继续比较后继字符
6.         else
7.             { i = i - j + 2; j = 1; } // 指针后退重新开始匹配
8.     }
9.     if ( j > T[0] ) return ( i - T[0] );
10.    else return 0;
11. }
```

4.3 串的模式匹配算法 (BF算法)



➤ 效率分析P93

- ▶ 最好情况 $O(n+m)$
- ▶ 最差情况 $O(n*m)$

➤ 提升——KMP算法

4.4 串操作应用举例——自学~



- ▶ 病毒感染检测
- ▶ 文本编辑
 - 处理规则：行插入/删除，页插入/删除，
 - 数据结构：页表、行表(行号，起始地址，长度)
- ▶ 建立词索引表
 - 数据结构
 - 词表——书名中的关键词集合
 - 关键词索引表

本节内容

第四章 串和数组

- 4.4.1 数组的类型定义
- 4.4.2 数组的顺序表示和实现
- 4.4.3 特殊矩阵的压缩存储

重点：数组的定义、表示与实现、特殊矩阵压缩存储

难点：稀疏矩阵的存储、转置

本节内容

- 数组——逻辑关系、类型定义

➤ 一维数组和多维数组

- ▶ 多维数组是一维数组的推广。
- ▶ 一维数组是线性结构，然而多维数组则是一种非线性结构。其特点是每一个数据元素可以有多个直接前驱和多个直接后继。
- ▶ 例如，二维数组可以视为其每一个数组元素为一维数组的一维数组，但从整体来看，每一个数组元素同时处于两个向量（行、列），它可能有两个直接前驱，有两个直接后继。
- ▶ 数组元素的下标一般具有固定的下界和上界，因此它比其他复杂的非线性结构简单。

➤ 数组的抽象数据类型定义

▸ ADT List{

数据对象:

$$j_i = 0, \dots, b_i - 1, i = 1, 2, \dots, n.$$

$$D = \{a_{j_1 j_2 \dots j_n} \mid a_{j_1 j_2 \dots j_n} \in \text{ElemSet}\} // n: \text{维度}, b_i: \text{数组第 } i \text{ 维的长度}$$

数据关系:

$$R = \{R_1, R_2, \dots, R_n\},$$

$$R_i = \{ \langle a_{j_1 \dots j_i \dots j_n}, a_{j_1 \dots j_{i+1} \dots j_n} \rangle \mid \\ 0 \leq j_k \leq b_k - 1, 1 \leq k \leq n, \text{ 且 } k \neq i, \\ 0 \leq j_i \leq b_i - 2, \\ a_{j_1 \dots j_i \dots j_n}, a_{j_1 \dots j_{i+1} \dots j_n} \in D, i = 2, 3, \dots, n \\ \}$$

基本操作:

InitArray (&A,n,bound1, ..., boundn) //构造数组A

操作结果: 若维度数n和各维度长度合法, 则构造相应的数组A,

DestroyArray (&A) // 销毁数组A

操作结果: 销毁数组A

Value(A,&e,index1,...,indexn) //取数组元素值

初始条件: A是n维数组, e为元素变量, 随后是n个下标值

操作结果: 若各下标不越界, 则e赋值为所指定的A的元素值, 并返回OK

Assign (A,&e,index1,...,indexn) //给数组元素赋值

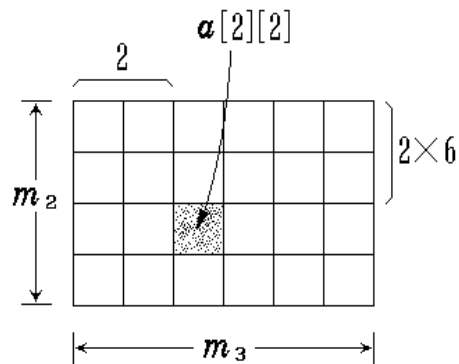
初始条件: A是n维数组, e为元素变量, 随后是n个下标值

操作结果: 若下标不越界, 则将e的值赋给所指定的A的元素, 并返回OK

}

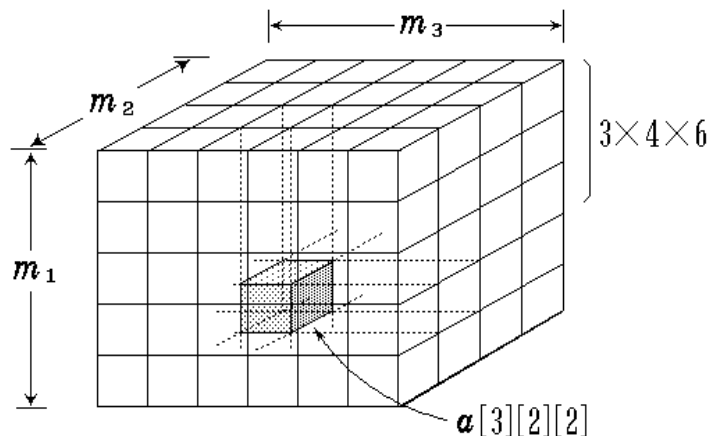
二维数组

$$m_1 = 5 \quad m_2 = 4 \quad m_3 = 6$$



行向量 下标 i
列向量 下标 j

三维数组



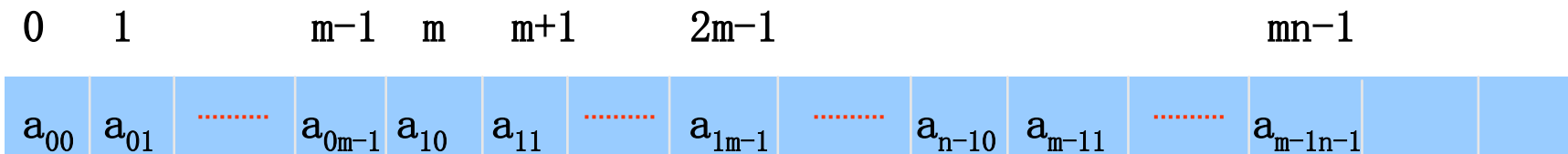
页向量 下标 i
行向量 下标 j
列向量 下标 k

$$\mathbf{a} = \begin{pmatrix} \mathbf{a}[0][0] & \mathbf{a}[0][1] & \cdots & \mathbf{a}[0][m-1] \\ \mathbf{a}[1][0] & \mathbf{a}[1][1] & \cdots & \mathbf{a}[1][m-1] \\ \mathbf{a}[2][0] & \mathbf{a}[2][1] & \cdots & \mathbf{a}[2][m-1] \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{a}[n-1][0] & \mathbf{a}[n-1][1] & \cdots & \mathbf{a}[n-1][m-1] \end{pmatrix}$$

行优先存放（以行为主序）：

设数组开始存放位置 $LOC(a[0][0]) = a$ ，每个元素占用 l 个存储单元，则 $a[i][j]$ 的存储位置为： $LOC(a[i][j]) = a + (i*m + j)*l$

其中， m 是每行元素个数，即列数。



$$\mathbf{a} = \begin{pmatrix} \mathbf{a}[0][0] & \mathbf{a}[0][1] & \cdots & \mathbf{a}[0][m-1] \\ \mathbf{a}[1][0] & \mathbf{a}[1][1] & \cdots & \mathbf{a}[1][m-1] \\ \mathbf{a}[2][0] & \mathbf{a}[2][1] & \cdots & \mathbf{a}[2][m-1] \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{a}[n-1][0] & \mathbf{a}[n-1][1] & \cdots & \mathbf{a}[n-1][m-1] \end{pmatrix}$$

▶ 列优先存放（以列为主序）：

设数组开始存放位置 $LOC(a[0][0]) = a$ ，每个元素占用 l 个存储单元，则 $a[i][j]$ 的存储位置为： $LOC(a[i][j]) = a + (j * n + i) * l$ 其中， n 是每列元素个数，即行数。

0 1 n-1 n n+1 2n-1 nm-1

a_{00}	a_{10}	a_{n-10}	a_{01}	a_{11}	a_{n-11}	a_{0n-1}	a_{1n-1}	a_{n-1n-1}	
----------	----------	-------	------------	----------	----------	-------	------------	-------	------------	------------	-------	--------------	-------	--

本节内容

➤ 物理存储

► 静态结构定义

1. typedef int ElemType; //数组元素的数据类型
2. int m = ..., n = ...; //行数和列数
3.
4. ElemType A[m][n]; //静态结构定义
5.

- 在程序中自动建立静态定义的二维数组，在程序结束时将自动销毁，因此要注意保存。
- 静态定义的二维数组不能扩充，在定义时就要充分考虑各维的上、下界。

► 动态结构定义

```
typedef int ElemType;      //数组元素的数据类型
int m = ..., n = ...;      //行数和列数
.....
ElemType **A;              //动态指针定义
.....
```

- 在程序中用动态存储分配建立的二维数组。

```
A = new ElemType *[m];
for ( i = 0; i < m; i++) A[i] = new ElemType [n];
```

- 动态回收也需要分两步：

```
for ( i = 0; i < m; i++) delete []A[i];
delete []A;
```

➤ 三维数组

- ▶ 可以看作是数组元素为二维数组的一维数组。它的处理类似二维数组。
- ▶ 假设各维元素个数为 m_1, m_2, m_3 , 每个元素占据 l 个存储单元, 按页/行/列方式存放, 那么, 下标为 i_1, i_2, i_3 的数组元素的存储地址为:

$$\text{LOC} (a[i_1][i_2][i_3]) =$$

$$a + (\underbrace{i_1 * m_2 * m_3}_{\substack{\text{前 } i_1 \text{ 页} \\ \text{总元} \\ \text{素数}}} + \underbrace{i_2 * m_3}_{\substack{\text{第 } i_1 \text{ 页的} \\ \text{前 } i_2 \text{ 行总} \\ \text{元素数}}} + \underbrace{i_3}_{\substack{\text{第 } i_2 \text{ 行} \\ \text{前 } i_3 \text{ 列} \\ \text{元素数}}}) * l$$

本节内容

- 特殊矩阵的压缩存储

➤ 特殊矩阵

- ▶ 特殊矩阵是指非零元素或零元素的分布有一定规律的矩阵。
- ▶ 特殊矩阵的压缩存储主要是针对阶数很高的特殊矩阵。为节省存储空间，对可以不存储的元素，如零元素或对称元素，不再存储。

➤ 有两种特殊矩阵：

- ▶ 对称矩阵、上三角/下三角矩阵
- ▶ 三对角矩阵

- ▶ 设有一个 $n \times n$ 的矩阵 A 。如果在矩阵中, $a_{ij} = a_{ji}$, 则此矩阵是对称矩阵。
 - ▶ 若只保存对称矩阵的对角线和对角线以上(下)的元素, 则称此为对称矩阵的压缩存储。

$$A = \begin{bmatrix} \mathbf{a}_{00} & \mathbf{a}_{01} & \mathbf{a}_{02} & \cdots & \mathbf{a}_{0n-1} \\ \mathbf{a}_{10} & \mathbf{a}_{11} & \mathbf{a}_{12} & \cdots & \mathbf{a}_{1n-1} \\ \mathbf{a}_{20} & \mathbf{a}_{21} & \mathbf{a}_{22} & \cdots & \mathbf{a}_{2n-1} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \mathbf{a}_{n-10} & \mathbf{a}_{n-11} & \mathbf{a}_{n-12} & \cdots & \mathbf{a}_{n-1n-1} \end{bmatrix}$$

- 若只存对角线及对角线以上的元素，称为上三角矩阵；若只存对角线或对角线以下的元素，称之为下三角矩阵。

$$\begin{bmatrix} \mathbf{a}_{00} & \mathbf{a}_{01} & \mathbf{a}_{02} & \cdots & \mathbf{a}_{0n-1} \\ \mathbf{a}_{10} & \mathbf{a}_{11} & \mathbf{a}_{12} & \cdots & \mathbf{a}_{1n-1} \\ \mathbf{a}_{20} & \mathbf{a}_{21} & \mathbf{a}_{22} & \cdots & \mathbf{a}_{2n-1} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \mathbf{a}_{n-10} & \mathbf{a}_{n-11} & \mathbf{a}_{n-12} & \cdots & \mathbf{a}_{n-1n-1} \end{bmatrix}$$

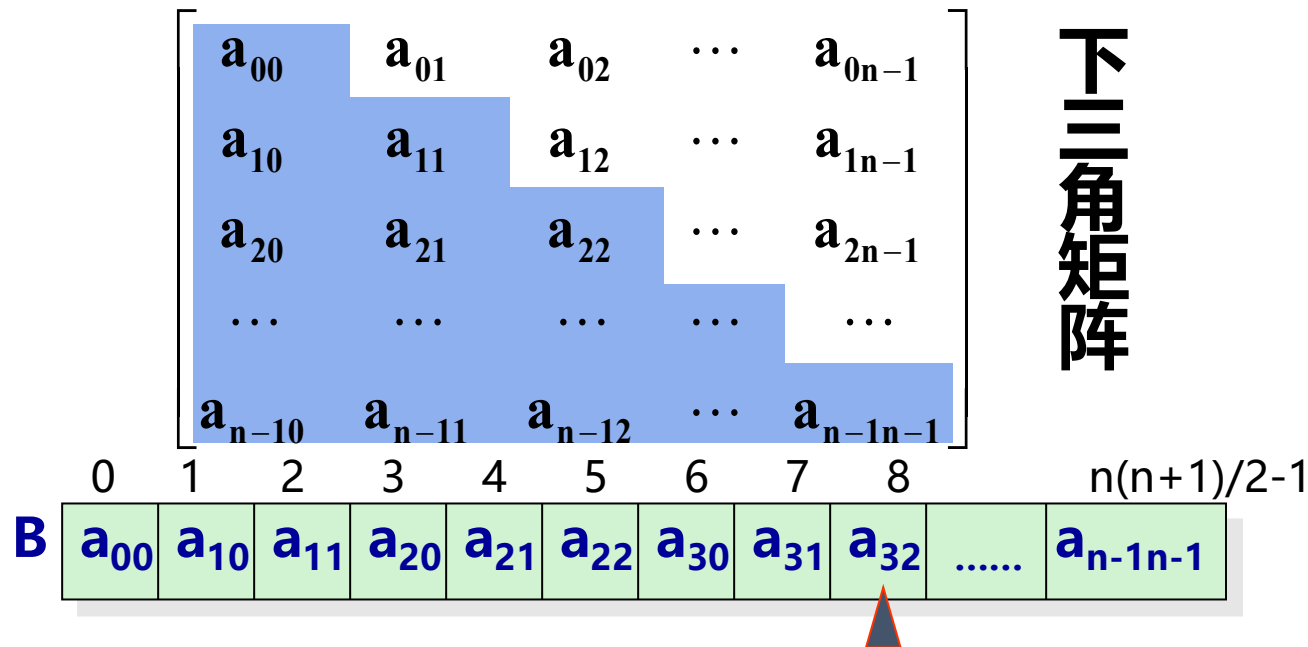
下三角矩阵

a_{00}	a_{01}	a_{02}	\cdots	a_{0n-1}
a_{10}	a_{11}	a_{12}	\cdots	a_{1n-1}
a_{20}	a_{21}	a_{22}	\cdots	a_{2n-1}
\cdots	\cdots	\cdots	\cdots	\cdots
a_{n-10}	a_{n-11}	a_{n-12}	\cdots	a_{n-1n-1}

上三角矩阵

- ▶ 把它们按行存放于一个一维数组 B 中，称之为对称矩阵 A 的压缩存储方式。
- ▶ 数组 B 共有 $n+(n-1)+\cdots+1 = n*(n+1)/2$ 个元素。

对称矩阵的压缩存储



- 若 $i \geq j$, 数组元素 $A[i][j]$ 在数组 B 中的存放位置为

$$1 + 2 + \dots + i + j = \underbrace{(i+1)*i/2}_{\text{前 } i \text{ 行元素总数}} + \underbrace{j}_{\text{第 } i \text{ 行第 } j \text{ 个元素前元素个数}}$$

前 i 行元素总数 第 i 行第 j 个元素前元素个数

- ▶ 若 $i < j$, 数组元素 $A[i][j]$ 在矩阵的上三角部分, 在数组 B 中没有存放, 可以找它的对称元素 $A[j][i] = j * (j + 1) / 2 + i$

- ▶ 反过来, 若已知某矩阵元素位于数组 B 的第 k 个位置, 可寻找满足

$$i(i+1)/2 \leq k < (i+1)(i+2)/2$$

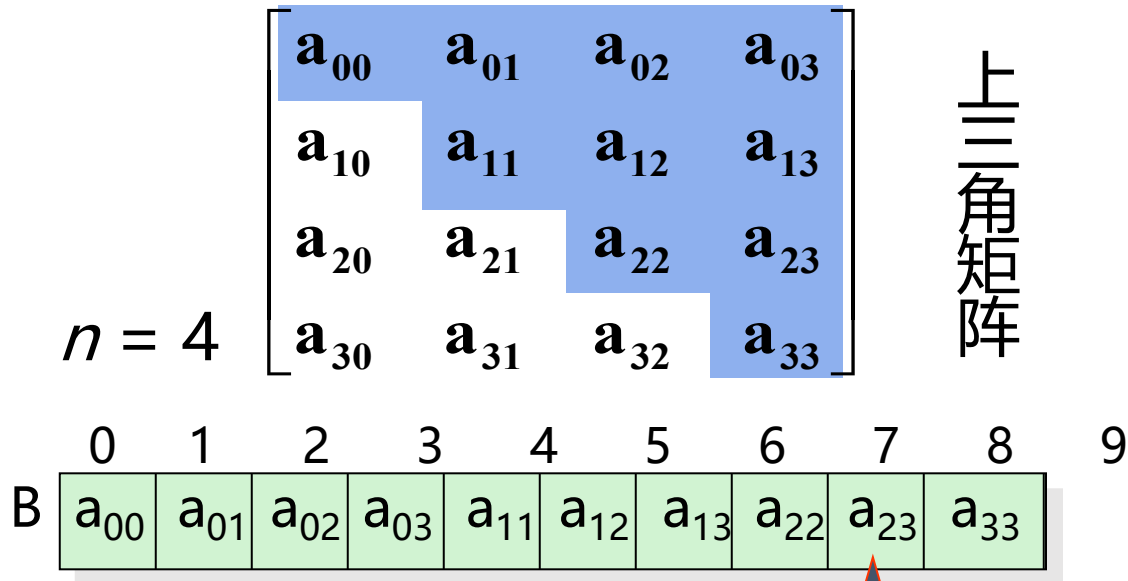
的 i , 此即为该元素的行号。

$$j = k - i * (i + 1) / 2$$

此即为该元素的列号。

- ▶ 例, 当 $k = 8$, $3*4/2 = 6 \leq k < 4*5/2 = 10$, 取 $i = 3$ 。则 $j = 8 - 3*4/2 = 2$ 。

对称矩阵的压缩存储



若 $i \leq j$, 数组元素 $A[i][j]$ 在数组 B 中的存放位置为

$$\underbrace{n + (n-1) + (n-2) + \cdots + (n-i+1)}_{\text{前 } i \text{ 行元素总数}} + \underbrace{j-i}_{\text{第 } i \text{ 行第 } j \text{ 个元素前元素个数}}$$

前 i 行元素总数

第 i 行第 j 个元素前元素个数

- ▶ 若 $i \leq j$, 数组元素 $A[i][j]$ 在数组 B 中的存放位置为

$$n + (n-1) + (n-2) + \cdots + (n-i+1) + j - i =$$

$$= (2*n-i+1) * i / 2 + j - i =$$

$$= (2*n-i-1) * i / 2 + j$$

- ▶ 若 $i > j$, 数组元素 $A[i][j]$ 在矩阵的下三角部分, 在数组 B 中没有存放。因此, 找它的对称元素 $A[j][i]$ 。 $A[j][i]$ 在数组 B 的第 $(2*n-j-1) * j / 2 + i$ 的位置中找到。

三对角矩阵的压缩存储

$$\mathbf{A} = \begin{bmatrix} a_{00} & a_{01} & 0 & 0 & 0 & 0 \\ a_{10} & a_{11} & a_{12} & 0 & 0 & 0 \\ 0 & a_{21} & a_{22} & a_{23} & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & a_{n-2n-3} & a_{n-2n-2} & a_{n-2n-1} \\ 0 & 0 & 0 & 0 & a_{n-1n-2} & a_{n-1n-1} \end{bmatrix}$$

\mathbf{B}

0	1	2	3	4	5	6	7	...	$3n-4$	$3n-3$
a_{00}	a_{01}	a_{10}	a_{11}	a_{12}	a_{21}	a_{22}	a_{23}	\dots	a_{n-1n-2}	a_{n-1n-1}

- ▶ 三对角矩阵中除主对角线及在主对角线上下最临近的两条对角线上的元素外，所有其它元素均为0。总共有 $3n-2$ 个非零元素。
- ▶ 将三对角矩阵A中三条对角线上的元素按行存放在一维数组 B 中，且 a_{00} 存放于B[0]。
- ▶ 在三条对角线上的元素 a_{ij} 满足
$$0 \leq i \leq n-1, i-1 \leq j \leq i+1$$
- ▶ 在一维数组 B 中 $A[i][j]$ 在第 i 行，它前面有 $3*i-1$ 个非零元素, 在本行中第 j 列前面有 $j-i+1$ 个，所以元素 $A[i][j]$ 在 B 中位置为 $k = 2*i + j$ 。