

数据结构和算法

刘聪

第5周 (2022/09/27-2022/09/29)

周二 3-4节 周四 3-4节

第4章

串、数组和广义表

本章内容

- 串的定义
- 案例引入
- 串的类型定义、存储结构及其运算
- 数组
- 广义表
- 案例分析与实现

串的定义

- 串(string)(或字符串)是由零个或多个字符组成的有限序列

本章内容

- 串的定义
- 案例引入
- 串的类型定义、存储结构及其运算
- 数组
- 广义表
- 案例分析与实现

案例引入

- 文字编辑、信息检索、语言编译
- 串匹配
 - 网络入侵检测、计算机病毒特征码匹配以及 DNA 序列匹配

案例引入

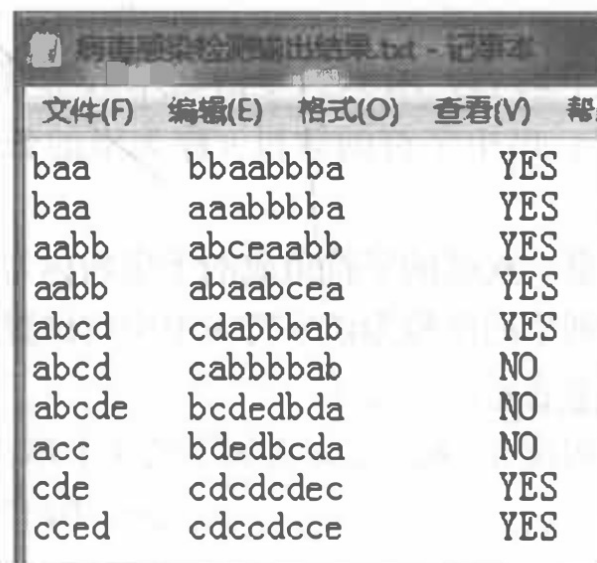
- 病毒感染检测

- 人的DNA序列是线性的，而病毒的DNA序列是环状的
- 假设病毒的DNA序列为 baa
- 患者1的DNA序列为 aaabbba,则感染
- 患者2的DNA序列为babbbba, 则未感染



文件(F)	编辑(E)	格式(O)	查看(V)
10			
baa	bbaabbba		
baa	aaabbbba		
aabb	abceaabb		
aabb	abaabcea		
abcd	cdabbbab		
abcd	cabbbbab		
abcde	bcdedbda		
acc	bdedbda		
cde	cdcdcdec		
cced	cdccdcce		

图 4.1 病毒感染检测输入数据



文件(F)	编辑(E)	格式(O)	查看(V)	帮助(H)
baa	bbaabbba		YES	
baa	aaabbbba		YES	
aabb	abceaabb		YES	
aabb	abaabcea		YES	
abcd	cdabbbab		YES	
abcd	cabbbbab		NO	
abcde	bcdedbda		NO	
acc	bdedbda		NO	
cde	cdcdcdec		YES	
cced	cdccdcce		YES	

图 4.2 病毒感染检测输出结果

本章内容

- 串的定义
- 案例引入
- 串的类型定义、存储结构及其运算
- 数组
- 广义表
- 案例分析于实现

串的类型定义、存储结构及其运算

- 串的抽象类型定义

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: $R1=\{ \langle a_{i-1}, a_i \rangle | a_{i-1}, a_i \in D, i=2, \dots, n \}$

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 $S > T$, 则返回值 > 0 ; 若 $S = T$, 则返回值 $= 0$; 若 $S < T$, 则返回值 < 0 。

串的类型定义、存储结构及其运算

• 串的抽象类型定义

`StrLength(S)`

初始条件：串 S 存在。

操作结果：返回 S 的元素个数，称为串的长度。

`ClearString(&S)`

初始条件：串 S 存在。

操作结果：将 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， $1 \leq pos \leq StrLength(S)$ 且 $0 \leq len \leq StrLength(S) - pos + 1$ 。

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 是非空串， $1 \leq pos \leq StrLength(S)$ 。

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

串的类型定义、存储结构及其运算

- 串抽象类型定义

`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 相等的非重叠的子串。

`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

串的类型定义、存储结构及其运算

- 串的存储结构
 - 串多采用顺序存储结构
 - 串的链式存储
 - 顺序串的插入和删除操作不方便，需要移动大量的字符。因此，可采用单链表方式存储串。如文本编辑器。



串的类型定义、存储结构及其运算

- 串的匹配算法
 - 搜索引擎、拼写检查、语言翻译、数据压缩
 - S为主串，也称正文串;设T为子串，也称为模式
 - Brute-Force 算法 和 KMP
 - Brute-Force
 - 在S的每个位置上看长度为T.size()的子串是否与T相同

```
4 int brute_force_search(char S[], char T[]) {  
5     int s_len = strlen(S);  
6     int t_len = strlen(T);  
7     for (int i = 0; i < s_len; ++ i) {  
8         if (strncmp(S+i, T, t_len) == 0)  
9             return i;  
10    }  
11    return -1;
```

串的类型定义、存储结构及其运算

- 串的存储结构
 - Brute-Force 的算法分析
 - 总比较趟数为 $n-1+m$
 - 每趟字符比较次数为 m
 - 最坏情况下的平均时间复杂度是 $O(n \times m)$

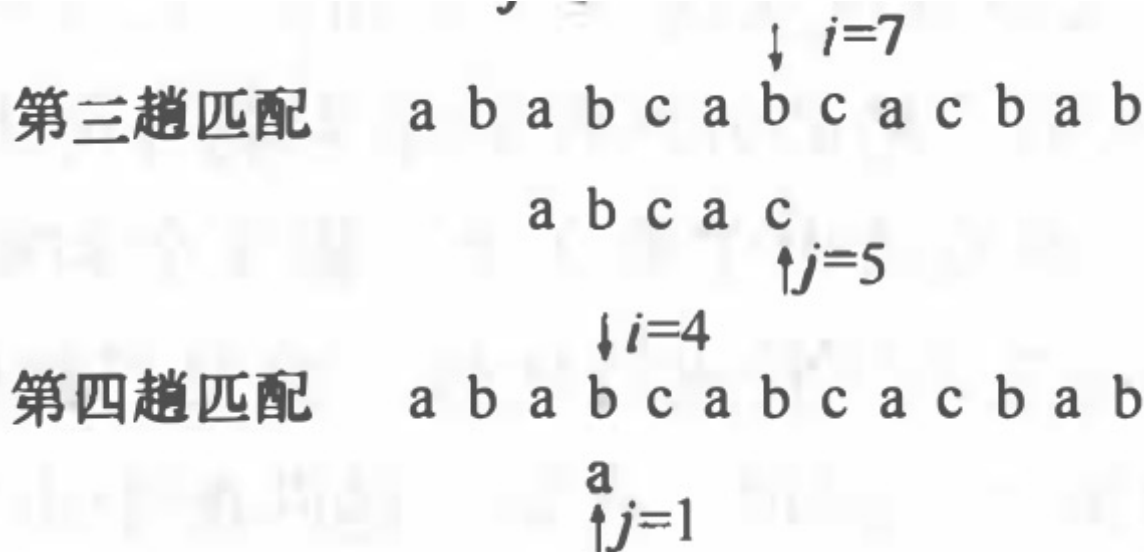
串的类型定义、存储结构及其运算

- 串的存储结构

- KMP算法

- Knuth 、 Morris 和 Pratt

- 假设 $T.size() \ll S.size()$, 因此预先分析T不影响复杂度



串的类型定义、存储结构及其运算

- 串的存储结构
 - KMP算法
 - i指针无须回溯

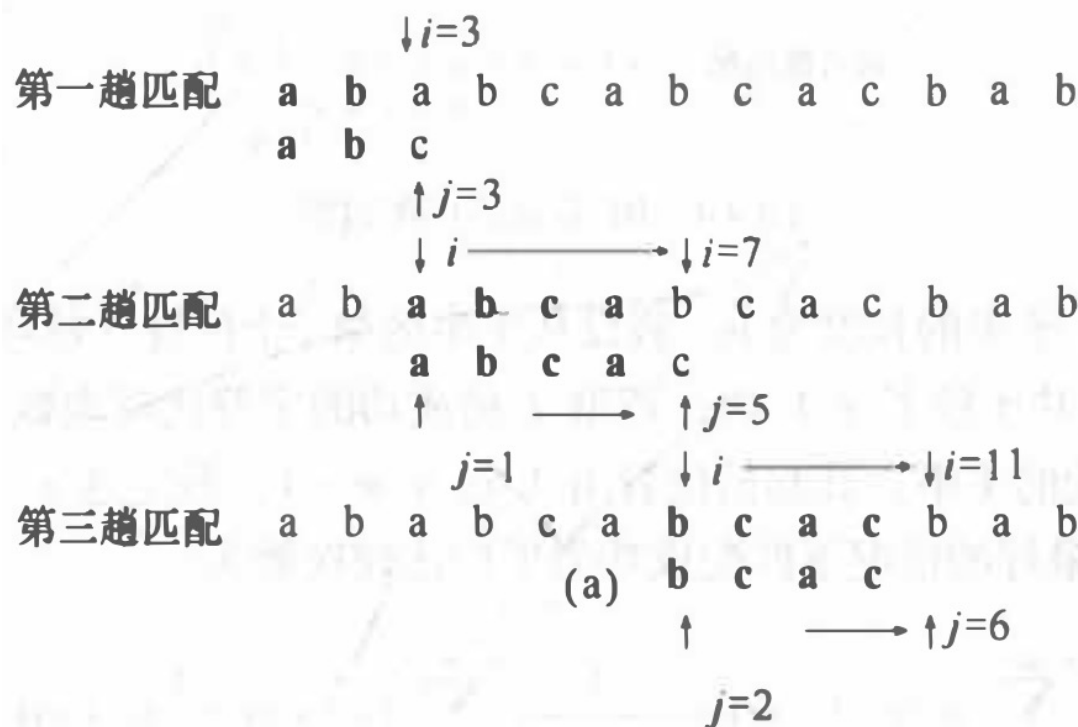


图 4.5 KMP 算法的匹配过程

串的类型定义、存储结构及其运算

- 串的存储结构

- KMP算法

- 假设主串为 "S1,S2 ... Sn" 模式串为 "T1T2占... Tm", 从上例的分析可知, 为了 实现改进算法, 需要解决下述问题:
 - 当匹配过程中产生 “失配” (即 $S_i \neq T_j$) 时, 模式串 “向右滑动” 可行的距离多远, 换句话说, 当主串中第 i 个字符与模式中第 j 个字符 “失配” (即比较不等) 时, 主串中第 i 个字符 (i 指针不回溯) 应与模式中哪个字符再比较?
 - 假设此时应与模式中第 k ($k < j$) 个字符继续比较, 则模式中前 $k-1$ 个字符的子串必须匹配 $S_{i-k+1} \dots S_{i-1}$, 且不可能存在 同样匹配的 $k' > k$.
 - 由于 $S_{i-k+1} \dots S_{i-1} = T_{j-k+1} \dots T_{j-1}$
 - 所以, 我们可以预先计算 T 中每个位置 j 上的 k

j	1	2	3	4	5	6	7	8
模式串	a	b	a	a	b	c	a	c
next[j]	0	1	1	2	2	3	1	2

图 4.6 模式串的 next 函数值

串的类型定义、存储结构及其运算

- 串的存储结构

- KMP算法

$$\text{next}[j] = \begin{cases} 0 & j=1 \text{ (} t_1 \text{与} s_i \text{比较不等时, 下一步进行} t_1 \text{与} s_{i+1} \text{的比较)} \\ \text{Max} \{k \mid 1 < k < j \text{ 且有 } "t_1 t_2 \cdots t_{k-1}" = "t_{j-k+1} t_{j-k+2} \cdots t_{j-1}"\} & \\ 1 & k=1 \text{ (不存在相同子串, 下一步进行} t_1 \text{与} s_i \text{的比较)} \end{cases}$$

串的类型定义、存储结构及其运算

- 串的存储结构
 - KMP算法 (简单版)

```
52 int main()  
53 {  
54     char S[] = "acabaabaabcacaabc";  
55     char T[] = "abaabcac";  
56     // print_next(T);  
57     printf("%d\n", KMP_search(S, T));  
58     return 0;  
59 }
```

串的类型定义、存储结构及其运算

- 串的存储结构
 - KMP算法 (简单版)

```
19 void print_next(char T[]) {  
20     update_next(T);  
21     int t_len = strlen(T);  
22     for (int i = 0; i < t_len; ++ i)  
23         printf("%c ", T[i]);  
24     printf("\n");  
25     for (int i = 0; i < t_len; ++ i)  
26         printf("%d ", next[i]);  
27     printf("\n");  
28 }
```

a	b	a	a	b	c	a	c
0	1	1	2	2	3	1	2

串的类型定义、存储结构及其运算

- 串的存储结构
 - KMP算法 (简单版)

```
4 char next[1000];
5
6 void update_next(char T[]) {
7     int t_len = strlen(T);
8     next[0] = 0;
9     for (int i = 1; i < t_len; ++ i) {
10         next[i] = 1;
11         for (int j = i; j >= 1; -- j)
12             if (strncmp(T, T+(i+1-j), j-1) == 0) {
13                 next[i] = j;
14                 break;
15             }
16     }
17 }
```

串的类型定义、存储结构及其运算

- 串的存储结构
 - KMP算法 (简单版)

```
30 int KMP_search(char S[], char T[]) {
31     update_next(T);
32     int s_len = strlen(S);
33     int t_len = strlen(T);
34     int j = 0;
35     for (int i = 0; i < s_len; ++ i) {
36         while (1) {
37             if (S[i] == T[j]) {
38                 ++ j;
39                 if (j == t_len) return i + 1 - t_len;
40                 break;
41             }
42             if (next[j] == 0) {
43                 j = 0;
44                 break;
45             }
46             j = next[j] - 1;
47         }
48     }
49     return -1;
50 }
```

串的类型定义、存储结构及其运算

- 串的存储结构

- KMP算法

```
28 int KMP_search(char S[], char T[]) {
29     update_next(T);
30     int s_len = strlen(S);
31     int t_len = strlen(T);
32     int i = -1;
33     int j = -1;
34     while (i < s_len && j < t_len) {
35         if (j == -1 || S[i] == T[j]) { ++i; ++j; }
36         else j = next[j] - 1;
37     }
38     if (j == t_len) return i - t_len;
39     return -1;
40 }
```

串的类型定义、存储结构及其运算

- 串的存储结构

- KMP算法

- $\text{next}[0] = 0$

- 递归地, 当 $\text{next}[j] == k \quad \dots (1)$

- 如果 $k == -1 \parallel T[j] == T[k]$ 则 $T[j] = k + 1 + 1$

- 否则 $k = \text{next}[k] - 1$, 回到 (1)

串的类型定义、存储结构及其运算

- 串的存储结构
 - KMP算法
 - next

```
6 void update_next(char T[]) { // O(m)
7     int t_len = strlen(T);
8     next[0] = 0;
9     int i = 0;
10    int j = -1;
11    while (i < t_len) {
12        if (j == -1 || T[i] == T[j]) { ++i; ++j; next[i] = j + 1; }
13        else j = next[j] - 1;
14    }
15 }
```

串的类型定义、存储结构及其运算

- 串的存储结构
 - BF 算法实际的执行时间近似 $O(n+m)$
 - KMP算法仅当模式与主串之间存在许多“部分匹配”的情况下，才显得比BF算法快得多
 - 但是KMP算法的最大特点是指示主串的指针不需回溯，整个匹配过程中，对主串仅需从头至尾扫描一遍，这对处理从外设输入的庞大文件很有效，可以边读入边匹配，而无需回头重读

本章内容

- 串的定义
- 案例引入
- 串的类型定义、存储结构及其运算
- 数组
- 广义表
- 案例分析与实现

数组

- 高维数组

$$A_{m \times n} = \begin{bmatrix} a_{00} & a_{01} & a_{02} & \cdots & a_{0, n-1} \\ a_{10} & a_{11} & a_{12} & \cdots & a_{1, n-1} \\ \vdots & \vdots & \vdots & & \vdots \\ a_{m-1, 0} & a_{m-1, 1} & a_{m-1, 2} & \cdots & a_{m-1, n-1} \end{bmatrix}$$

(a) 矩阵形式表示

$$, A_{m \times n} = \begin{bmatrix} a_{00} \\ a_{01} \\ \vdots \\ a_{m-1, 0} \end{bmatrix} \begin{bmatrix} a_{01} \\ a_{11} \\ \vdots \\ a_{m-1, 1} \end{bmatrix} \cdots \begin{bmatrix} a_{0, n-1} \\ a_{1, n-1} \\ \vdots \\ a_{m-1, n-1} \end{bmatrix}$$

(b) 列向量的一维数组

数组

- 高维数组

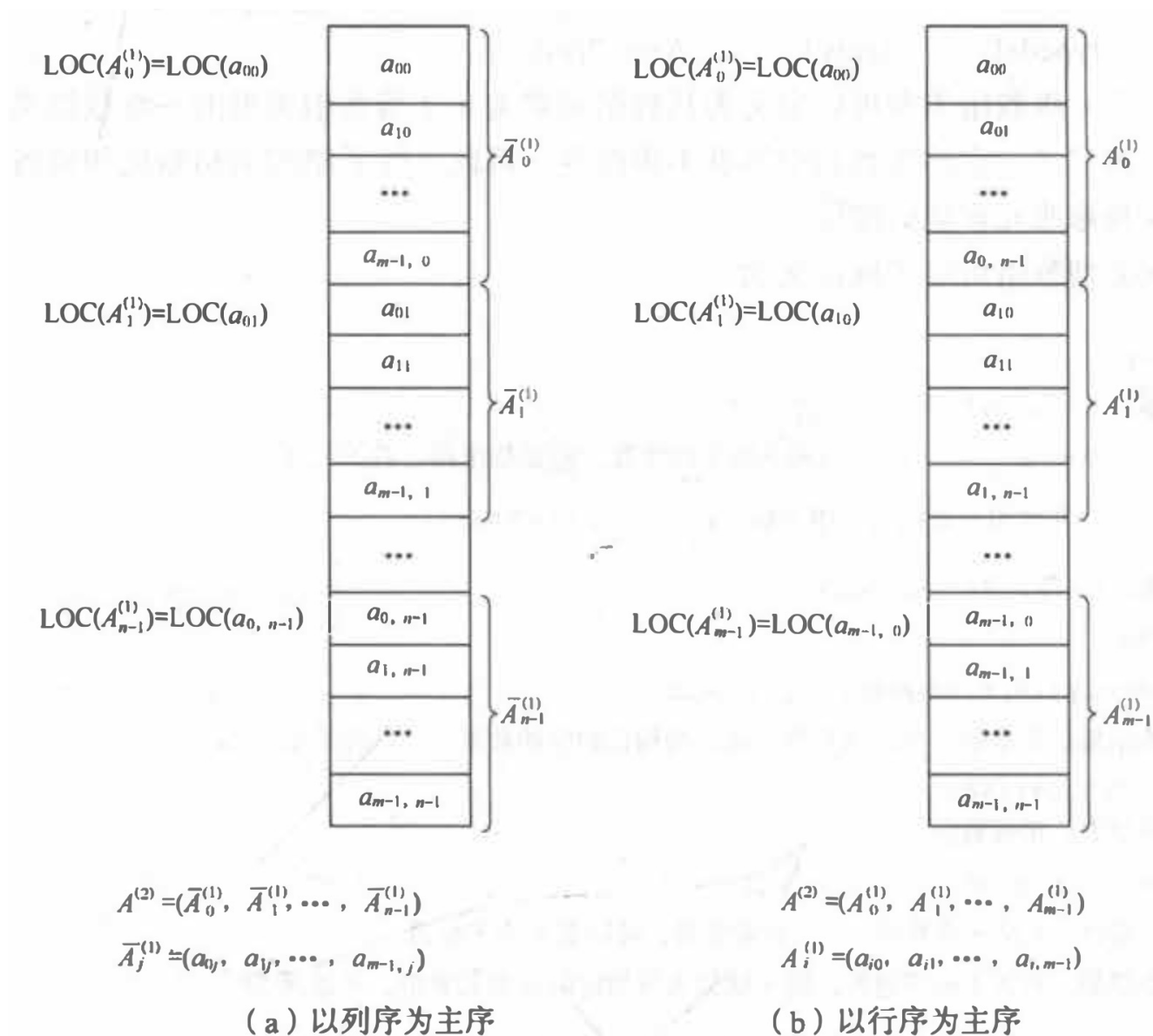


图 4.11 二维数组的两种存储方式

数组

- 高维数组
 - 高维索引转换为一维索引

$$\text{LOC}(j_1, j_2, \dots, j_n) = \text{LOC}(0, 0, \dots, 0) + \sum_{i=1}^n c_i j_i$$

数组

- 高维数组

```
1 #include <stdio.h>
2
3 int double_eq(double * d1, double * d2) {
4     return *d1 == *d2;
5 }
6 #define ELEM_EQ double_eq
7 #define ELEM_TYPE double
8
9 #include "seq_list.c"
10
11 typedef struct {
12     int _dims[3];
13     List _data;
14 } Tensor;
```

数组

- 高维数组

```
16 int Tensor_init(Tensor * this, int dim1, int dim2, int dim3) {
17     if (! (dim1>0 && dim2>0 && dim3>0))
18         return -1;
19     int num = dim1*dim2*dim3;
20     if (List_init(&(this->_data), num))
21         return -1;
22     double zero = 0;
23     for (int i = 0; i < num; ++ i)
24         List_insert(&(this->_data), i, &zero);
25     this->_dims[0] = dim1;
26     this->_dims[1] = dim2;
27     this->_dims[2] = dim3;
28     return 0;
29 }
30
31 void Tensor_finalize(Tensor * this) {
32     List_finalize(&(this->_data));
33 }
```


数组

```
35 int Tensor_dim(Tensor * this, int index) {
36     if (index < 0 || index >= 3)
37         return -1;
38     return this->_dims[index];
39 }
40
41 int _Tensor_index(Tensor * this, int index1, int index2, int index3) {
42     int dim1 = this->_dims[0];
43     int dim2 = this->_dims[1];
44     int dim3 = this->_dims[2];
45     if (index1 < 0 || index1 >= dim1) return -1;
46     if (index2 < 0 || index2 >= dim2) return -1;
47     if (index3 < 0 || index3 >= dim3) return -1;
48     return index1*dim2*dim3 + index2*dim3 + index3;
49 }
50
51 ELEM_TYPE * Tensor_get(Tensor * this, int index1, int index2, int index3) {
52     int index = _Tensor_index(this, index1, index2, index3);
53     if (index == -1)
54         return 0;
55     return List_get(&(this->_data), index);
56 }
```

数组

- 高维数组

```
58 void print(Tensor * this, int dim) {
59     for (int i = 0; i < Tensor_dim(this,1); ++ i) {
60         for (int j = 0; j < Tensor_dim(this,2); ++ j)
61             printf("%.1lf ", *Tensor_get(this, dim, i, j));
62         printf("\n");
63     }
64 }
65
66 // 测试: gcc -g -fno-omit-frame-pointer -fsanitize=address -fPIE 4-4-2.c
67 //      ASAN_OPTIONS=detect_leaks=1 ./a.out
68
69 int main() {
70     Tensor t;
71     Tensor_init(&t, 10, 10, 10);
72     for (int i = 0; i < 1000; ++ i)
73         *List_get(&(t._data), i) = i+1;
74     print(&t, 9);
75     Tensor_finalize(&t);
76 }
```

数组

- 特殊矩阵的压缩存储
 - 对称矩阵

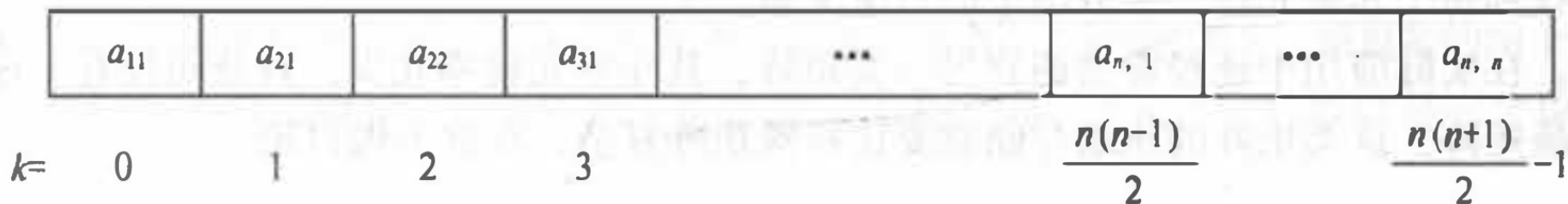


图 4.12 对称矩阵的压缩存储

$$k = \begin{cases} \frac{i(i-1)}{2} + j - 1 & \text{当 } i \geq j \\ \frac{j(j-1)}{2} + i - 1 & \text{当 } i < j \end{cases}$$

```
11 typedef struct {
12     int _dims[2];
13     List _data;
14 } SymmetricMatrix;
15
16 int SymmetricMatrix_init(SymmetricMatrix * this, int size) {
17     if (! (size>0))
18         return -1;
19     int num = size *(size+1)/2;
20     if (List_init(&(this->_data), num))
21         return -1;
22     double zero = 0;
23     for (int i = 0; i < num; ++ i)
24         List_insert(&(this->_data), i, &zero);
25     this->_dims[0] = size;
26     this->_dims[1] = size;
27     return 0;
28 }
29
30 void SymmetricMatrix_finalize(SymmetricMatrix * this) {
31     List_finalize(&(this->_data));
32 }
```

数组

- 特殊矩阵的压缩存储
 - 对称矩阵

```
40 int _SymmetricMatrix_index(SymmetricMatrix * this, int index1, int index2) {  
41     int dim1 = this->_dims[0];  
42     int dim2 = this->_dims[1];  
43     if (index1 < 0 || index1 >= dim1) return -1;  
44     if (index2 < 0 || index2 >= dim2) return -1;  
45     if (index1 < index2) {  
46         int tmp = index1;  
47         index1 = index2;  
48         index2 = tmp;  
49     }  
50     return index1 * (index1 + 1) / 2 + index2;  
51 }
```


数组

- 特殊矩阵的压缩存储

- 对称矩阵

```
71 int main() {  
72     SymmetricMatrix t;  
73     SymmetricMatrix_init(&t, 10);  
74     for (int i = 0; i < (10*11/2); ++ i)  
75         *List_get(&(t._data), i) = i+1;  
76     print(&t);  
77     SymmetricMatrix_finalize(&t);  
78 }
```

```
1.0 2.0 4.0 7.0 11.0 16.0 22.0 29.0 37.0 46.0  
2.0 3.0 5.0 8.0 12.0 17.0 23.0 30.0 38.0 47.0  
4.0 5.0 6.0 9.0 13.0 18.0 24.0 31.0 39.0 48.0  
7.0 8.0 9.0 10.0 14.0 19.0 25.0 32.0 40.0 49.0  
11.0 12.0 13.0 14.0 15.0 20.0 26.0 33.0 41.0 50.0  
16.0 17.0 18.0 19.0 20.0 21.0 27.0 34.0 42.0 51.0  
22.0 23.0 24.0 25.0 26.0 27.0 28.0 35.0 43.0 52.0  
29.0 30.0 31.0 32.0 33.0 34.0 35.0 36.0 44.0 53.0  
37.0 38.0 39.0 40.0 41.0 42.0 43.0 44.0 45.0 54.0  
46.0 47.0 48.0 49.0 50.0 51.0 52.0 53.0 54.0 55.0
```

数组

- 特殊矩阵的压缩存储

- 上三角矩阵

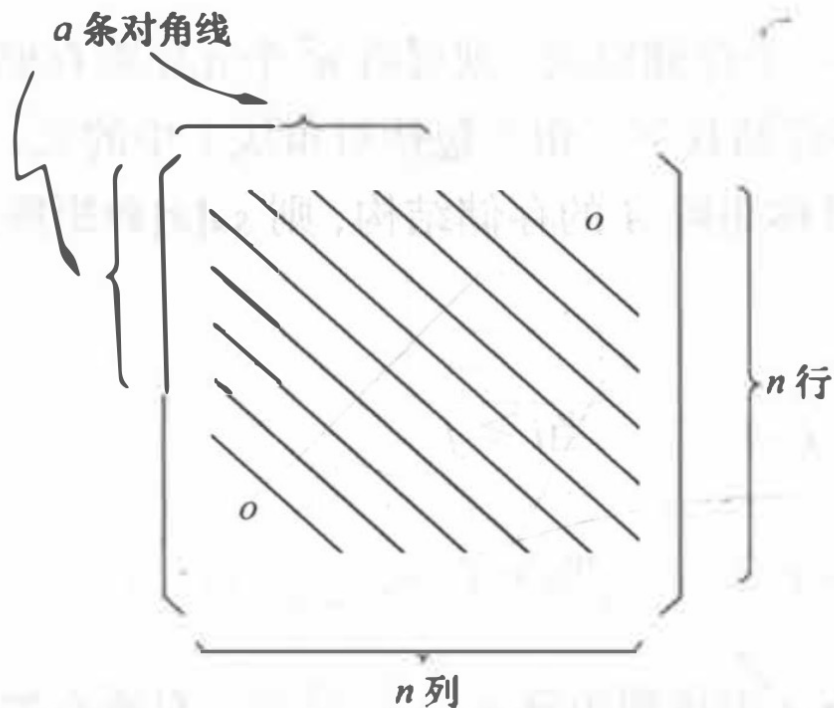
$$k = \begin{cases} \frac{(i-1)(2n-i+2)}{2} + (j-i) & \text{当 } i \leq j \\ \frac{n(n+1)}{2} & \text{当 } i > j \end{cases}$$

- 下三角矩阵

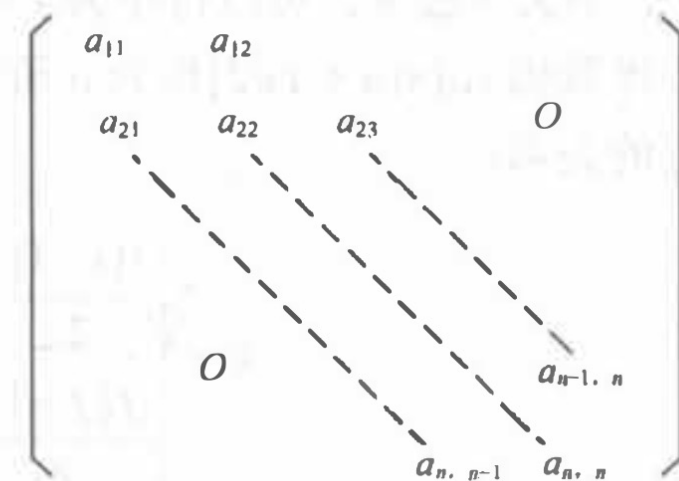
$$k = \begin{cases} \frac{i(i-1)}{2} + j - 1 & \text{当 } i \geq j \\ \frac{n(n+1)}{2} & \text{当 } i < j \end{cases}$$

数组

- 特殊矩阵的压缩存储
 - 对角矩阵



(a) 一般情形



(b) 三对角矩阵

图 4.13 对角矩阵

本章内容

- 串的定义
- 案例引入
- 串的类型定义、存储结构及其运算
- 数组
- 广义表
- 案例分析与实现

广义表

- 树，下一章再讲

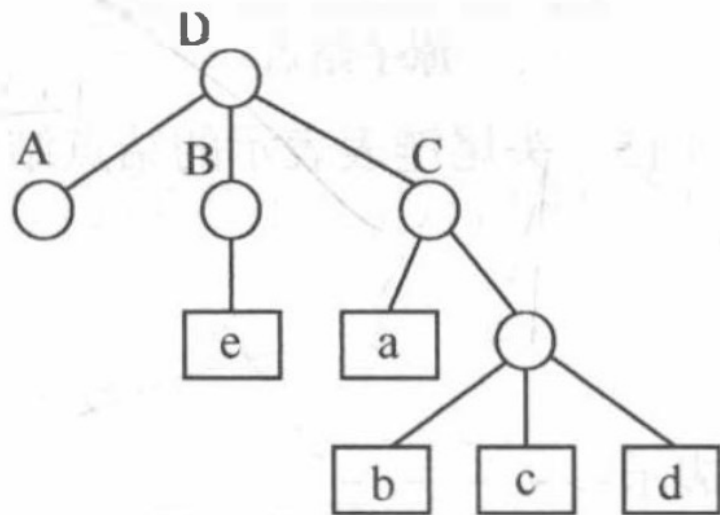


图 4.14 广义表的图形表示

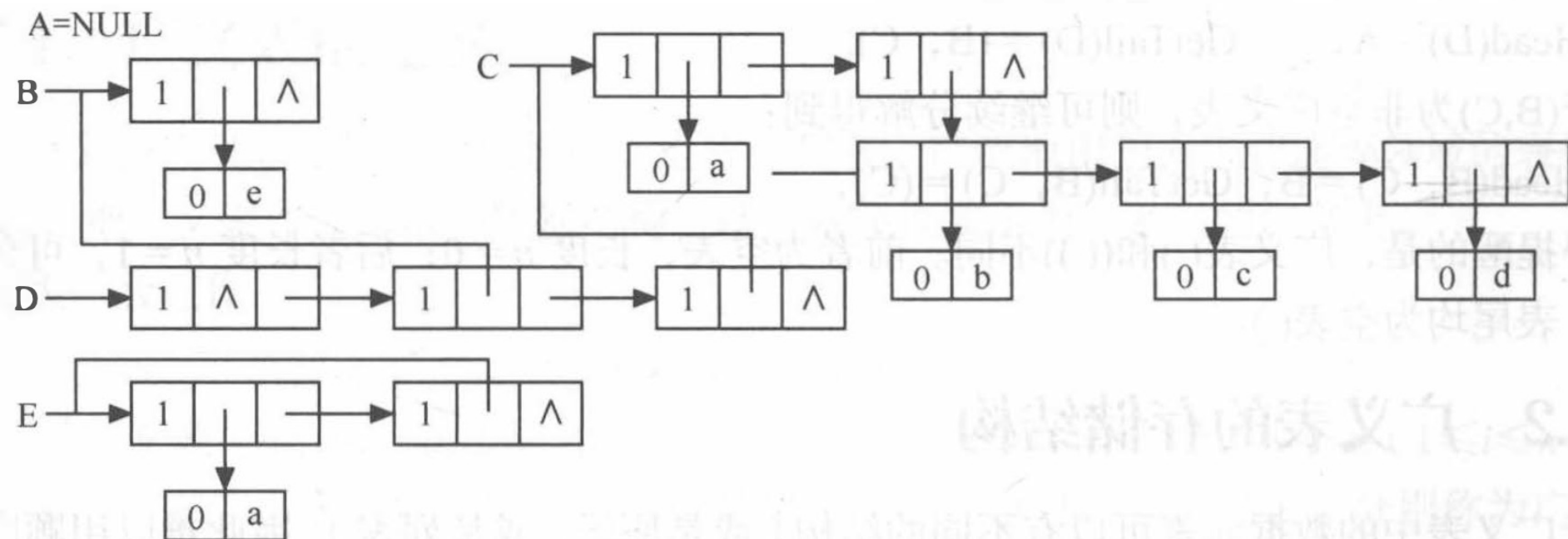


图 4.16 头尾链表表示的存储结构示例

本章内容

- 串的定义
- 案例引入
- 串的类型定义、存储结构及其运算
- 数组
- 广义表
- 案例分析与实现

案例分析与实现

```

void Virus_detection()
{
    //利用 BF 算法实现病毒检测

    ifstream inFile("病毒感染检测输入数据.txt");
    ofstream outFile("病毒感染检测输出结果.txt");

    inFile>>num; //读取待检测的任务数
    while(num-->0) //依次检测每对病毒 DNA 和人的 DNA 是否匹配
    {
        inFile>>Virus.ch+1; //读取病毒 DNA 序列，字符串从下标 1 开始存放
        inFile>>Person.ch+1; //读取人的 DNA 序列
        Vir=Virus.ch; //将病毒 DNA 临时暂存在 Vir 中，以备输出
        flag=0; //用来标识是否匹配，初始为 0，匹配后为非 0
        m=Virus.length; //病毒 DNA 序列的长度是 m
        for(i=m+1,j=1;j<=m;j++)
            Virus.ch[i++]=Virus.ch[j]; //将病毒字符串的长度扩大 2 倍
        Virus.ch[2*m+1]='\0'; //添加结束符号
        for(i=0;i<m;i++) //依次取得每个长度为 m 的病毒 DNA 环状字符串 temp
        {
            for(j=1;j<=m;j++) temp.ch[j]=Virus.ch[i+j];
            temp.ch[m+1]='\0'; //添加结束符号
            flag=Index_BF(Person,temp,1); //模式匹配
            if(flag) break; //匹配即可退出循环
        } //for
        if(flag) outFile<<Vir+1<<" "<<Person.ch+1<<" "<<"YES"<<endl;
        else outFile<<Vir+1<<" "<<Person.ch+1<<" "<<"NO"<<endl;
    } //while
}

```