串、数组、广义表

串string

> 串：零个或多个字符组成的有限序列。**必须调用标准库函数** `string.h`

串的定义

s='$a_1a_2a_3\cdots a_n$'

s——串名

$a_1a_2\cdots a_n$——串值

n——串长

n=0——空串

子串（模式串）——传中任意个连续字符组成的子序列。

主串（正文串）——包含子串的串。

字符位置——字符在序列中的序号。

子串位置——子串的第一个字符在朱传中的序号。

串的类型定义、存储结构及运算

类型定义

```
//数据对象：D={a_i|i=1,2···n,n≥0}
//数据关系：R_1={<a_i-1,a_i>|a_i-1,a_i∈D}
//基本操作
StrAssign(&T,chars)//串赋值
```

```
StrCompare(S,T)//串比较

StrLength(S)//求串长

Concat(&T,s1,s2)//串联

SubString(&Sub,S,pos,len)//求子串

StrCopy(&T,S)//串拷贝

StrEmpty(&S)//串判空

ClearString(&S)//清空串

Index(S,T,pos)//求子串的位置，模式匹配

Replace(&S,T,V)//串替换

StrInsert(&S,pos,T)//子串插入

StrDelete(&S,pos,len)//子串删除

DestoryString(&S)//串销毁
```

串的存储结构

- 顺序存储
  - 定长
  - 堆式
- 链式存储

串的定长顺序存储结构

```
#define MAXLEN 255//串的最大长度
typedef struct{
    char ch[MAXLEN+1];//存储串的一维数组
    int length;//串的当前长度
}SString;//0号单元闲置，串都是从下标为1的数组分量考试存储
```

串的堆式顺序存储结构

```
typedef struct{
    char *ch;//若串非空，则按串长分配存储空间，否则ch为NULL
    int length;//串的当前长度
}HString;//存储空间时在程序自行过程中哦对那个太分配得到的
```

串的链式存储结构

```
#define CHUNKSIZE 80//此处数值大小可自定义
typedef struct Chunk{
    char h[CHUNKSIZE];
    struct Chunk *next;
}Chunk;

typedef struct{
    Chunk *head,*tail;//串的头指针和尾指针
    int curlen;//串的当前长度
}LString//块链串
```

串的模式匹配算法

Purpose：确定珠串中所含子串第一次出现的位置（定位）。即如何实现Index（S，T，pos）函数，字串定位运算

算法：BF算法（暴力算法，普通的模式匹配算法）；KMP算法（改进，特点：速度快）

BF算法

设计思想：将主串S的第pos个字符和模式串T的第一个字字符比较，若相等则继续比较后续字符，若不等，则从主串S的下一个字符(i=i-j+2)比较起，重新与模式串的第一个字符比较。

知道主串S的一个连续子串字符序列与模式串T相等。返回值为S中与T匹配的子序列第一个字符的序号（`i-T.length`）,即匹配成功。否则，匹配失败，返回值0.

```
//算法描述
int Index(SString S,SString T,int pos){
```

```
    //1≤pos≤S.length
    i=pos;
    j=1;
    while(i<S.length&&j≤T.length){
        if(S,ch[i]==T.ch[j]){
            ++i;
            ++j;
        }
        else{
            i=i-j+2;
            //i的当前位置，减去j的当前位置，得到的是当前回合i初始位置前的位置，加2就会得到当前回合i的初始位置的下一个位置。
            j=1;
        }
    }
    if(j>T.length)
        return i-T.length;//匹配成功，返回当前匹配成功回合i的初始位置
    else
        return 0;
}
```

若n为主串长度，m为子串长度，最坏情况下匹配次数为：$(n-m)*m+m=(n-m+1)*m$

若m<<n，则算法复杂度O(n*m)

KMP算法

特点：S的指针i不用回溯，算法复杂度提速到O(n+m)

构建部分匹配表（next数组）

```
//计算next函数值
void get_next(SString T, int next[]) {
    int i = 1;            // i 表示当前正在处理 T 的字符位置（从第 1 位开始）
    int j = 0;            // j 表示前缀长度指针
    next[1] = 0;          // 初始化，模式串的第一个字符的 next 值为 0

    while (i < T[0]) {    // T[0] 表示串长（假设 SString 是顺序串，第 0 位是长度）
```

```
            if (j == 0 || T[i] == T[j]) {//i=j+1
                ++i;
                ++j;
                next[i] = j; // 记录当前 i 的 next 值
            } else {
                j = next[j]; // 回退 j，直到能匹配或退到 0
            }
        }
    }
```

这个函数的作用是：**生成每一个位置的最长相同前后元素长度**，构建出 **next 数组**，用于 KMP 算法中的跳跃匹配。

🚦 逻辑步骤

1. 初始化：
   - `i = 1`，表示从第一个字符开始构建；
   - `next[1] = 0`，第一个字符前面没有内容，所以设为 0；
   - `j = 0`，前缀长度初始为 0。
2. `while (i < T[0])`：
   - 继续处理直到扫描完整个字符串。
3. 匹配成功：
   - 如果当前字符 `T[i] == T[j]`，说明当前位置可以延续匹配，`j++`，然后设置 `next[i+1] = j`。
4. 匹配失败：
   - 如果 `T[i] ≠ T[j]`，我们就把 `j` 回退为 `next[j]`，尝试更短的前缀，看是否能继续匹配。

这个函数是构建 **KMP 算法用的 next 数组**，核心思想是：

> 找出每一位子串中前缀和后缀最长的公共部分长度，并用它来指导后续匹配时跳过重复对比的部分。

KMP（`knuth-Morris-pratt`）

```
```

```
int Index_KMP(SString S,SString T,int pos){
    i=pos;
    j=1;
    while(i⩽S.length&&j⩽T.length){
        if(j==0||S,ch[i]==T.ch[j]){
            i++;
            j++;
        }
        else{
            j=next[j]
        }
    }
    if(j>T.length)
        return i-T.length;
    else
        return 0;
}
```

KMP算法实现示例

数组

> 高级语言中的数组是顺序结构；此处所讨论的数组既可以是顺序的，也可以是链式结构。
>
> 从$a[0,0]$开始

```
//数组的抽象数据类型
ADT Array{
    //数据对象
    D={a_ji|}

    //基本操作
    InitArray(&A,n,bound_1,bound_2,...,bound_n);
    DestoryArray(&A);
    Value(A,&e,index1,index2,...,indexn);
    Assign(&A,e,index1,index2,...,indexn)
}ADT Array
```

一维数组

> 相当于一个线性表和一个向量，在计算机内存放在**连续的**存储单元，适合**随机查找**。

二维数组

> 看作**线性表**的推广。

- 行优先顺序* 以行序为主序，C/C++,JAVA,BASIC,PASCAL等高级语言
- 列优先顺序

行优先顺序（低下标优先），初始位置$LOC(0,0)=a$,元素`a[j][k]`的存储地址为$LOC(j,k)=a+(j*m+k)*L$,L为每个元素所占空间。

三维数组

三维数组`A[b1][b2][b3]`各维元素个数为$b_1,b2,b3$（维界）——页、行、列

下标为j_1,j_2,j_3的数组元素的储存位置$LOC(j_1,j_2,j_3)=a+(j_1*b_2*b_3+j_2*b_3*j_3)*L$

$j_1*b_2*b_3$是前j_1页总元素个数

n维数组

$b_1, b_2, \ldots, b_n$是n维的维界，数组元素$a_{j1, j2, \ldots, jn}$的**存储地址**为：

$$
\begin{aligned}
LOC(j_1, j_2, \ldots j_n) = LOC(0, 0, \ldots, 0) &+ (b_2*b_3*\ldots *b_n*j_1 \\
&+ b_3*\ldots *b_n*j_2 \\
&+ \ldots\ldots \\
&+ b_n*j_{n-1} \\
&+ j_n \ ) *L \\
= LOC(0, 0, \ldots, 0) &+ \left(\sum_{i=1}^{n-1} j_i \prod_{k=i+1}^{n} b_k + j_n\right) *L
\end{aligned}
$$

可由二、三维的计算公式类比推得。

▶▶ 课堂练习

设有一个二维数组 $A[m][n]$按行优先 顺序存储，假设元素 $A[0][0]$存放位置在$644_{(10)}$，$A[2][2]$存放位置在$676_{(10)}$，每个元素占一个空间，问$A[3][3]_{(10)}$存放在什么位置？脚注$_{(10)}$表示用10进制表示。

解答：设数组元素A[i][j]存放在起始地址为Loc (i, j) 的存储单元中

∵ Loc (2, 2) = Loc (0, 0) + 2 * n + 2 = 644 + 2 * n + 2 = 676.
∴ n = (676 - 2 - 644) / 2 = 15
∴ Loc (3, 3) = Loc (0, 0) + 3 * 15 + 3 = 644 + 45 + 3 = 692.

特殊矩阵得压缩存储

1. 什么是压缩存储?
   - 多个数据元素值相同，则只分配一个元素值得存储空间。
   - 零元素不分配存储空间。
2. 什么样的矩阵能够压缩?

特殊矩阵，如：对称矩阵、对角矩阵、三角矩阵、稀疏矩阵。

3. 什么叫做稀疏矩阵?

矩阵中非零元素得个数较少且分布无规律（一般小于5%）

对称矩阵

> 在n阶矩阵A中满足：$a_{ij}=a_{ji}(1=<i,j<=n)$

存储方法：只存储上（或下）三角（包括主对角线）的数据元素。共占用 `n*(n+1)/2` 个元素空间。

三角矩阵

> 对角线一下（或以上）的数据元素（不包括对角线）全部为常数c，假设矩阵下标从1开始。

存储方法：重复元素c共享一个元素存储空间，内存共占用 `n(n+1)/2+1` 个元素空间。

对角矩阵（带状矩阵）

> 在n阶方阵中，非零元素集中在对角线及其两侧
>
> 共L（奇数）条对角线的带状区域内——L对角矩阵

存储方法：只存储带状区内的元素。

稀疏矩阵

> 矩阵中大多数元素为零但分布无规律，通常系数银子<0.05时成为稀疏矩阵。

存储方法：只存储每一非零元素(i,j,a_{ij})。节省了空间，但丧失随机存取功能。

顺序储存：三元组（顺序）表——适用转置算法

链式储存：十字（正交）链表——适用加减算法

广义表 General Lists

n个表元素组成的有限序列，记作$LS=(a_0,a_1,a_2,\dots,a_{n-1})$

LS是表名，a_1是表元素，它可以是表（称为子表），可以是数据元素（称为原子）。n为表的长度。n=0的冠以表为空表。

广义表与线性表的区别

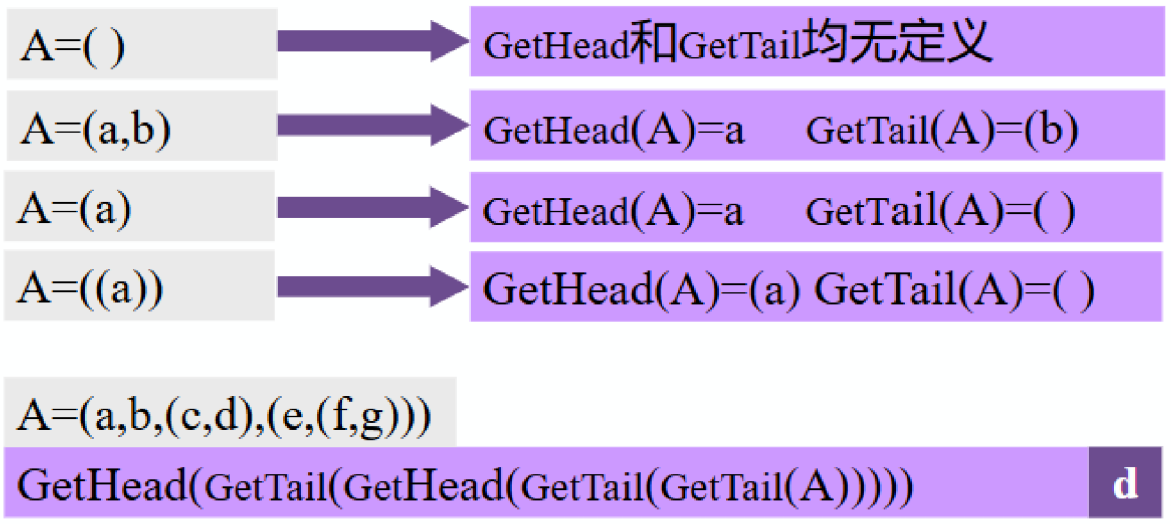
- 线性表的成分都是结构上不可分的单元素
- 广义表的成分可以是单元素，也可以是有结构的表。
- 线性表是一种特殊的广义表。
- 广义表不一定是线性表，也不一定是线性结构。

广义表的基本运算

- 求表头 `GetHead(L)` 非空广义表的第一个元素，可以是一个氮元素，也可以是一个子表。
- 求表尾 `GetTail(L)` 非空广义表除去表头元素以外其他元素所构成的表。表尾一定是一个表。

广义表的特点

- 有次序性——一个直接前驱和一个直接后继
- 有长度——表中元素的个数
- 有深度——表中括号的重数
- 可递归——自己可以作为自己的子表
- 可共享——可为其他广义表共享



A=(a,b,(c,d),(e,(f,g)))

GetHead(GetTail(GetHead(GetTail(GetTail(A))))) **d**