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

- **串**
  - 定义、ADT、操作
  - 实现（定长、堆分配、块链）
- **串的模式匹配**
  - BF算法
  - KMP算法

回顾

- **数组**
  - 逻辑关系、类型定义及实现
  - 特殊矩阵的压缩存储

——对称阵/上下三角阵、三对角阵（未完待续）

  - 稀疏矩阵的存储和转置(遗留问题)

本节内容

- 特殊矩阵的压缩存储（续）
- 稀疏矩阵
  - 存储
  - 转置
- 树

本节内容

- 特殊矩阵的压缩存储

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

对称矩阵的压缩存储

- **设有一个 n×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。

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

对称矩阵的压缩存储

- 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。

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

下三角矩阵

对称矩阵的压缩存储

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

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)+⋯+1 = n*(n+1)/2 个元素。

对称矩阵的压缩存储

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

下三角矩阵

	0	1	2	3	4	5	6	7	8		n(n+1)/2-1
B	a_{00}	a_{10}	a_{11}	a_{20}	a_{21}	a_{22}	a_{30}	a_{31}	a_{32}	……	a_{n-1n-1}

- 若 i≥j, 数组元素A[i][j]在数组B中的存放位置为

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

对称矩阵的压缩存储

- 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

$$i (i + 1) / 2 \leqslant k < (i + 1)*(i + 2) / 2$$

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

$$n = 4 \quad \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix}$$

上三角矩阵

	0	1	2	3	4	5	6	7	8	9
B	a_{00}	a_{01}	a_{02}	a_{03}	a_{11}	a_{12}	a_{13}	a_{22}	a_{23}	a_{33}

若 i≤j，数组元素A[i][j]在数组B中的存放位置为

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

对称矩阵的压缩存储

- 若 $i \le j$，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

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

	0	1	2	3	4	5	6	7	…	3n-4	3n-3
B	a_{00}	a_{01}	a_{10}	a_{11}	a_{12}	a_{21}	a_{22}	a_{23}	…	a_{n-1n-2}	a_{n-1n-1}

三对角矩阵的压缩存储

- 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aij 满足

$$0 \le i \le n-1,\ i-1 \le j \le i+1$$

- 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。

- **稀疏矩阵**
  - 存储

5.3.2 稀疏矩阵

- **稀疏矩阵(Sparse Matrix)**：对于稀疏矩阵，目前还没有一个确切的定义。设矩阵A是一个n×m的矩阵中有s个非零元素，设 δ=s/(n×m)，称δ为稀疏因子，如果某一矩阵的稀疏因子δ满足δ≦0.05时称为稀疏矩阵，如图所示。

$$A_{6\times7} = \begin{pmatrix} 0 & 12 & 9 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 0 & 14 & 0 \\ 0 & 0 & 24 & 0 & 0 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 & 0 \\ 15 & 0 & 0 & -7 & 0 & 0 & 0 \end{pmatrix}$$

稀疏矩阵示例

5.3.2 稀疏矩阵的压缩存储

$$A_{6\times7} = \begin{pmatrix} 0 & 12 & 9 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 0 & 14 & 0 \\ 0 & 0 & 24 & 0 & 0 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 & 0 \\ 15 & 0 & 0 & -7 & 0 & 0 & 0 \end{pmatrix}$$

- 对于稀疏矩阵，采用压缩存储方法时，只存储非0元素。必须存储非0元素的行下标值、列下标值、元素值。因此，一个三元组(i, j, a_{ij})唯一确定稀疏矩阵的一个非零元素。
- 如上图的稀疏矩阵A的三元组线性表为：

((1,2,12), (1,3,9), (3,1,-3), (3,6,14), (4,3,24), (5,2,18), (6,1,15), (6,4,-7))

5.3.2 三元组顺序表

- 若以行序为主序，稀疏矩阵中所有非0元素的三元组，就可以得构成该稀疏矩阵的一个三元组顺序表。相应的数据结构定义如下：
- 三元组结点定义

```
#define MAXSIZE 12500
typedef int ElemType ;
typedef struct{
      int   i;               /*  行下标  */
      int  j ;               /*  列下标  */
      ElemType e;            /*  元素值  */
}Triple ;
```

```
typedef struct {
     int  mu ;              /*   行数   */
     int  nu ;              /*   列数   */
     int  tu ;              /*  非0元素个数   */
     Triple   data[MAXSIZE+1] ;
}TSMatrix ;
```

$$A_{6\times7} = \begin{pmatrix} 0 & 12 & 9 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 0 & 14 & 0 \\ 0 & 0 & 24 & 0 & 0 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 & 0 \\ 15 & 0 & 0 & -7 & 0 & 0 & 0 \end{pmatrix}$$

6	rn行数	
7	cn列数	
8	tn元素个数	
1	2	12
1	3	9
3	1	-3
3	6	14
4	3	24
5	2	18
6	1	15
6	4	-7
row	col	value

(a) 原矩阵的三元组表

7	rn行数	
6	cn列数	
8	tn元素个数	
1	3	-3
1	6	15
2	1	12
2	5	18
3	1	9
3	4	24
4	6	-7
6	3	14
row	col	value

(b)转置矩阵的三元组表

稀疏矩阵及其转置矩阵的三元组顺序表

5.3.2 三元组顺序表

矩阵的运算包括矩阵的转置、矩阵求逆、矩阵的加减、矩阵的乘除等。在此，先讨论在这种压缩存储结构下的求矩阵的转置的运算。

一个m×n的矩阵A，它的转置B是一个n×m的矩阵，且b[i][j]=a[j][i]，0 ≤ i ≤ n，0 ≤ j ≤ m，即B的行是A的列，B的列是A的行。

设稀疏矩阵A是按行优先顺序压缩存储在三元组表a.data中，若仅仅是简单地交换a.data中i和j的内容，得到三元组表b.data，b.data将是一个按列优先顺序存储的稀疏矩阵B，要得到按行优先顺序存储的b.data，就必须重新排列三元组表b.data中元素的顺序。

- **稀疏矩阵**
  - 转置

三元组表稀疏矩阵的转置——方法一：

- **求转置矩阵的基本算法思想是：**
  - 将矩阵的行、列下标值交换。即将三元组表中的行、列位置值i 、j相互交换；
  - 重排三元组表中元素的顺序。即交换后仍然是按行优先顺序排序的。（例如：BubbleSort）

三元组表稀疏矩阵的转置——方法二 Brute Force野蛮转置

- Brute Force**算法思想：**
  - 按稀疏矩阵A的三元组表a.data中的**列次序依次**找到相应的三元组存入b.data中。
  - 每找转置后矩阵的一个三元组，需从头至尾扫描整个三元组表a.data 。找到之后自然就成为按行优先的转置矩阵的压缩存储表示。

按方法二求转置矩阵的算法如下：

```
Status TransposeMatrix(TSMatrix M , TSMatrix &T){
    T.mu=M.nu； T.nu=M.nu； T.tu=M.tu；
    //置三元组表T.data的行、列数和非0元素个数
    if (M.tu==0)   printf( “ The Matrix A=0\n”  );
    else{
        q=1;
        for (col=1; col<=M.nu；col++)
        //每循环一次找到转置后的一个三元组
            for (p=1；p<M.tu；p++)   //循环次数是非0元素个数
                if (M.data[p].j==col){
                    T.data[q].i=M.data[p].j；
                    T.data[q].j=M.data[p].i;
                    T.data[q].e=M.data[p].e;
                    q++；
                }
    }
    return OK;
}
```

算法分析：本算法主要的工作是在p和col的两个循环中完成的，故算法的时间复杂度为O(nu×tu)，即矩阵的列数和非0元素的个数的乘积成正比。

而一般传统矩阵的转置算法为：

```
1. for(col=1; col<=n ;++col)
2.     for(row=0 ; row<=m ;++row)
3.         b[col][row]=a[row][col] ;
```

其时间复杂度为O(nu×mu)。当非零元素的个数tu和mu×nu同数量级时，算法TransMatrix的时间复杂度为O(mu×nu^2)。

由此可见，虽然节省了存储空间，但时间复杂度却大大增加。所以上述算法只适合于稀疏矩阵中非0元素的个数tu远远小于mu×nu的情况。

三元组表稀疏矩阵的转置——方法三：快速转置

- **方法三(快速转置) 算法思想：**
  - 直接按照稀疏矩阵A的三元组表a.data的次序依列次顺序转换，并将转换后的三元组放置于三元组表b.data的恰当位置。
  - 前提：若能预先确定原矩阵A中每一列的(即B中每一行)第一个非0元素在b.data中应有的位置，则在作转置时就可直接放在b.data中恰当的位置。因此，应先求得A中每一列的非0元素个数。
  - 附设两个辅助向量num[ ]和cpot[ ] 。
    - num[col]：统计A中第col列中非0元素的个数；
    - cpot[col] ：指示A中第一个非0元素在b.data中的恰当位置。

显然有位置对应关系：

$$\begin{cases} cpot[1]=1 \\ cpot[col]=cpot[col-1]+num[col-1] \qquad 2 \leq col \leq a.nu \end{cases}$$

例图5-8中的矩阵A和表5-9(a)的相应的三元组表可以求得num[col]和cpot[col]的值如表5-1：

表5-1　num[col]和cpot[col]的值表

col	1	2	3	4	5	6	7
num[col]	2	2	2	1	0	1	0
cpot[col]	1	3	5	7	8	8	9

快速转置算法part1

```
void  FastTransMatrix(TMatrix a, TMatrix b) {
        int p, q, col, k;
        int num[MAX_SIZE], copt[MAX_SIZE];
        b.rn = a.cn; b.cn = a.rn; b.tn = a.tn;
        /*   置三元组表b.data的行、列数和非0元素个数  */
        if (b.tn == 0)    printf( “ The Matrix A = 0\n” );
        else {
                for (col = 1; col <= a.cn; ++col)    num[col] = 0;
                /*  向量num[]初始化为0   */
                for (k = 1; k <= a.tn; ++k)
                        ++num[a.data[k].col];
```

快速转置算法part2

```
        /*   求原矩阵中每一列非0元素个数   */
        for (cpot[0] = 1, col = 2; col <= a.cn; ++col)
                cpot[col] = cpot[col - 1] + num[col - 1];
        /*  求第col列中第一个非0元在b.data中的序号 */
        for (p = 1; p <= a.tn; ++p) {
                col = a.data[p].col;  q = cpot[col];
                b.data[q].row = a.data[p].col;
                b.data[q].col = a.data[p].row;
                b.data[q].value = a.data[p].value;
                ++cpot[col];      /*至关重要!!当本列中  */
        }
    }
```

作业预告

- 稀疏矩阵存储及转置

总结

- **稀疏矩阵**
  - 存储（三元组表）
  - 转置（三种方法：排序、BF法、快速转置）
- **之后内容，请自学**

5.3.2 行逻辑链接的三元组顺序表

将上述方法二中的辅助向量cpot[ ]固定在稀疏矩阵的三元组表中，用来指示“行”的信息。得到另一种顺序存储结构：行逻辑链接的三元组顺序表。其类型描述如下：

```
#define MAX_ROW 100
typedef struct {
        Triple data[MAX_SIZE + 1];        //非0元素的三元组表
        int    rpos[MAX_ROW + 1];         //各行第一个非0位置表
        int    mu, nu, tu;                //矩阵的行、列数和非0元个数
}RLSMatrix;
```

5.3.2 稀疏矩阵的乘法

设有两个矩阵：$A=(a_{ij})_{m\times n}$， $B=(b_{ij})_{n\times p}$

则： $C=(c_{ij})_{m\times p}$ 其中 $c_{ij}=\sum a_{ik}\times b_{kj}$

$1\leqq k\leqq n$， $1\leqq i\leqq m$， $1\leqq j\leqq p$

经典算法是三重循环：

```
for (i = 1; i <= m; ++i)
for (j = 1; j <= p; ++j) {
        c[i][j] = 0;
        for (k = 1; k <= n; ++k)
                c[i][j] = c[i][j] + a[i][k]□ b[k][j];
}
```

此算法的复杂度为$O(m\times n\times p)$。

5.3.2 稀疏矩阵的乘法

设有两个稀疏矩阵$A=(a_{ij})_{m\times n}$， $B=(b_{ij})_{n\times p}$， 其存储结构采用行逻辑链接的三元组顺序表。

算法思想：对于A中的每个元素a.data[p](p=1, 2, … , a.tu)，找到B中所有满足条件：

a.data[p].j=b.data[q].i的元素b.data[q]，求得a.data[p].e×b.data[q].e，该乘积是c_{ij}中的一部分。求得所有这样的乘积并累加求和就能得到c_{ij}。

为得到非0的乘积，只要对a.data[1…a.tu] 中每个元素(i，k，a_{ik})(1≦i≦a.rn，1≦k≦a.cn)， 找到b.data中所有相应的元素(k，j，b_{kj})(1≦k≦b.rn，1≦j≦b.cn) 相乘即可。则必须知道矩阵B中第k行的所有非0元素，而b.rpos[ ]向量中提供了相应的信息。

b.rpos[row]指示了矩阵B的第row行中第一个非0元素在b.data[ ]中的位置(序号)，显然，b.rpos[row+1]-1指示了第row行中最后一个非0元素在b.data[ ]中的位置(序号)。最后一行中最后一个非0元素在b.data[ ]中的位置显然就是b.tu。

两个稀疏矩阵相乘的算法如下:

```
Status MultSMatrix(RLSMatrix M, RLSMatrix N, RLSMatrix& Q) {
    if (M.nu != N.mu) return ERROR;
    Q.mu = M.mu; Q.nu = N.nu; Q.tu = 0;
    if (M.tu * N.tu != 0) { // Q是非零矩阵
        for (arow = 1; arow <= M.mu; ++arow) {
                // 处理M的每一行
        } // for arow
    } // if
    return OK;
} // MultSMatrix
```

```
ctemp[] = 0; // 当前行各元素累加器清零 Q.rpos[arow] = Q.tu+1;
if (arow < M.mu)tp = M.rpos[arrow + 1];
for (p = M.rpos[arow]; p < tp; ++p) { //对当前行中每一个非零元处理
        brow = M.data[p].j;
        if (brow < N.nu)
                t = N.rpos[brow + 1];
        else { t = N.tu + 1 }
        for (q = N.rpos[brow]; q < t; ++q) {
                ccol = N.data[q].j;
                ctemp[ccol] += M.data[p].e * N.data[q].e;
        } // for q
}
```

```
for (ccol = 1; ccol <= Q.nu; ++ccol)
if (ctemp[ccol]) {
        if (++Q.tu > MAXSIZE) return ERROR;
        Q.data[Q.tu] = { arow, ccol, ctemp[ccol] };
} // if
```

5.3.2 十字链表

- 对于稀疏矩阵，当非0元素的个数和位置在操作过程中变化较大时，采用链式存储结构表示比三元组的线性表更方便。
- 矩阵中非0元素的结点所含的域有：行、列、值、行指针(指向同一行的下一个非0元)、列指针(指向同一列的下一个非0元)。其次，十字交叉链表还有一个头结点，结点的结构如图5-10所示。

row	col	value
down	right	

(a) 结点结构

rn	cn	tn
down	right	

(b) 头结点结构

图5-10 十字链表结点结构

由定义知，稀疏矩阵中同一行的非0元素的由right指针域链接成一个行链表，由down指针域链接成一个列链表。则每个非0元素既是某个行链表中的一个结点，同时又是某个列链表中的一个结点，所有的非0元素构成一个十字交叉的链表。称为十字链表。

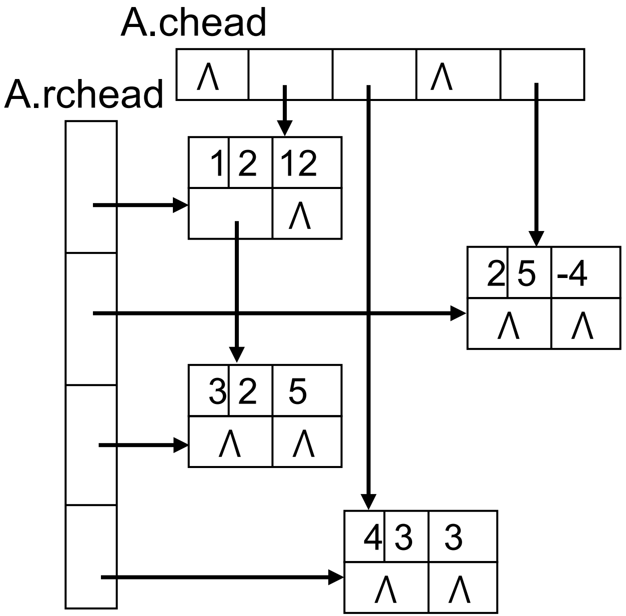
此外，还可用两个一维数组分别存储行链表的头指针和列链表的头指针。对于图5-11(a)的稀疏矩阵A，对应的十字交叉链表如图5-11(b)所示，结点的描述如下：

```
typedef struct  OLNode  {
        int  i,   j;   /*  行号和列号  */
        ElemType e;    /*  元素值  */
        struct  OLNode  *down , *right ;
}OLNode, *OLink ;   /*  非0元素结点  */
```

```
typedef struct {
        int   mu;        /*  矩阵的行数  */
        int   bu;        /*  矩阵的列数  */
        int   tu;        /*  非0元素总数  */
        OLNode *rhead ;
        OLNode *chead ;
    } CrossList ;
```

$$A=\begin{pmatrix} 0 & 12 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -4 \\ 0 & 5 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 \end{pmatrix}$$

(a) 稀疏矩阵

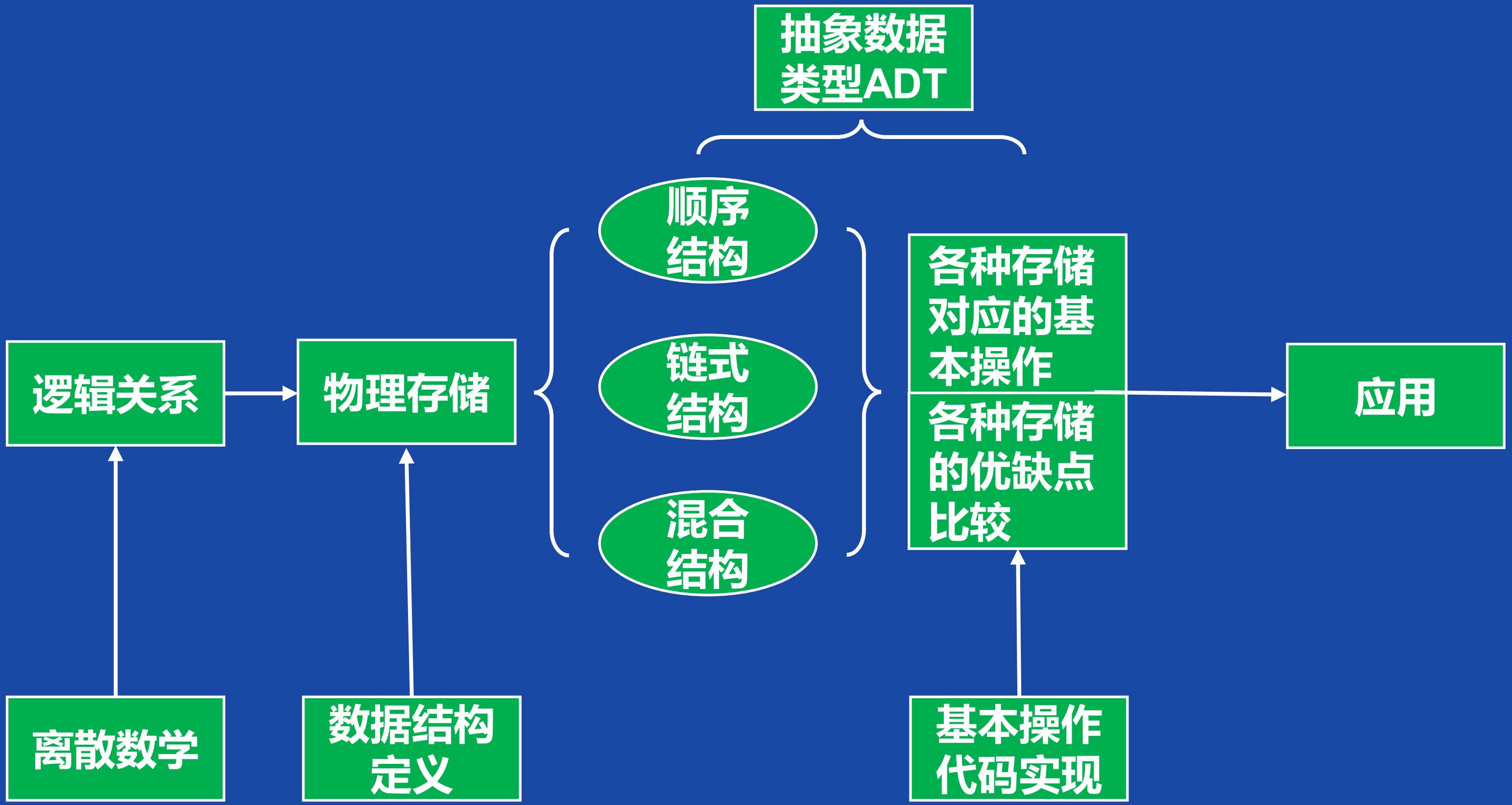


(b) 稀疏矩阵的十字交叉链表

图5-11 稀疏矩阵及其十字交叉链表

本章内容

- 树


抽象数据类型ADT
顺序结构
链式结构
混合结构
逻辑关系
物理存储
各种存储对应的基本操作
各种存储的优缺点比较
应用
离散数学
数据结构定义
基本操作代码实现

本章内容

第五章 树和二叉树

- 5.1 树的定义和基本术语
- 5.3-5.4 二叉树定义、性质、存储结构
- 5.5 遍历二叉树和线索二叉树
- 5.6 树和森林
- 5.7 哈夫曼树及其应用
- 5.2 5.8 案例

重点：二叉树的性质、遍历；
难点：基于二叉树遍历的算法设计

本节内容

- **树**
  - 树的定义和基本术语
  - 二叉树定义、性质

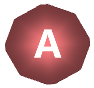
本节内容

- 树的定义和基本术语

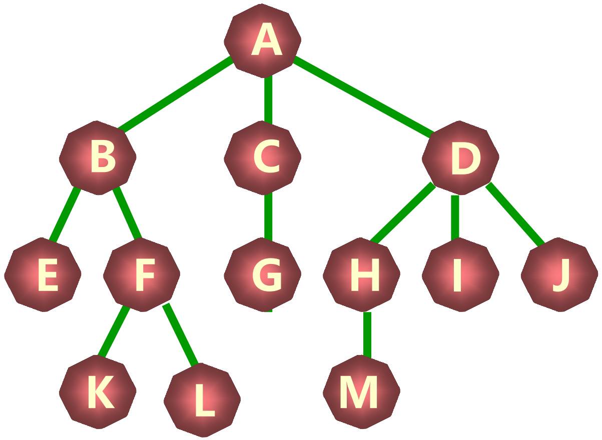
树的定义(递归定义)

- 有树是n(n≥0)个结点(元素)的有限集。
- 若n=0，称为空树。
- 若n > 0，则
  - 且仅有一个特定的称为根的结点root；
  - 当n > 1时，除根以外的其他结点划分为m(m>0) 个互不相交的有限集T1, T2,… Tm，其中每一个集合本身又是一棵树，并且称为根的子树。
  - 且对任意的i(m≥i≥1), T_i存在惟一的结点x_i , 有<root, x_i >∈H H为树中元素之间的二元关系集

- **树的表示**



只有根结点的树

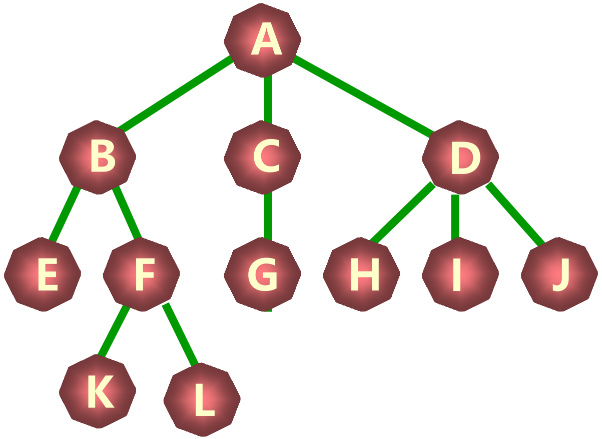


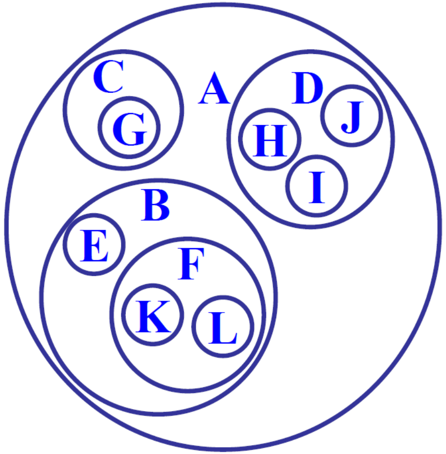
有13个结点的树

树的定义-其他表示

- **树的其他表示**

嵌套集合、广义表表示、文氏图表示、凹入表示





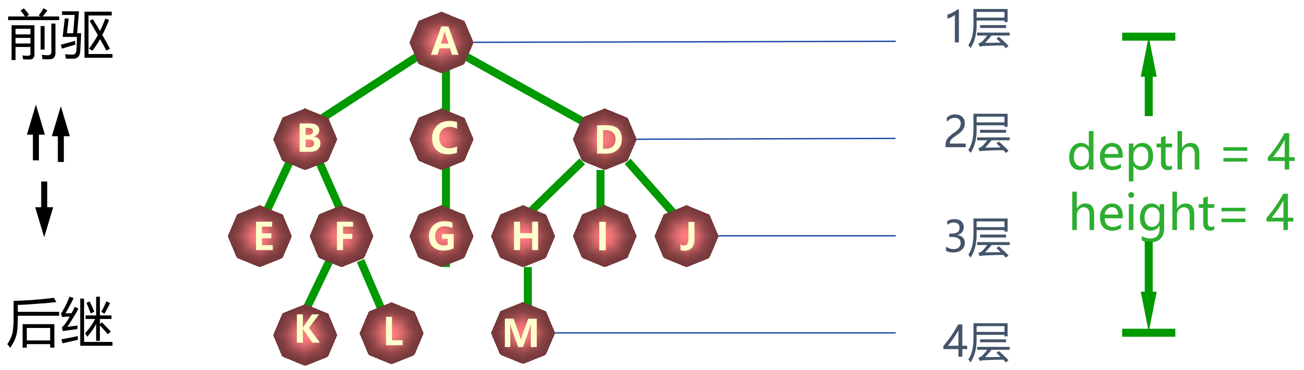

```
A*****************
  B****************
    E***************
    F***************
      K**************
      L**************
  C****************
    G***************
  D****************
    H***************
    I***************
    J***************
```

{A,{B,{E, F,{K,L}}, C,{G}, D,{H, I, J}}}

(A(B(E, F(K,L)), C(G), D(H, I, J)))

树的定义-基本术语

基本术语



- 结点
- 结点的**度**
- 结点的层次
- 终端结点（叶子）
- 非终端结点（分支结点）
- 内部结点

- 孩子
- 双亲
- 兄弟
- 祖先
- 子孙
- 堂兄弟

- 树的度
- 树的深度/高度
- 有序树
- 无序树
- 森林

有序树无序树与森林

- **有序树**
  - 如果将树中结点看成是从左到右有次序的，即不能互换，则称该树为有序树。
  - 有序树最左边子树的根称为第一个孩子，最右边的称为最后一个孩子。
- **森林（Forest）**
  - 森林（Forest）是m棵互不相交的树的集合。（m>=0）树的每个结点的子树集合也是森林。
  - 递归定义。

树的定义-树的抽象数据类型定义

树的特有操作

- 查找：双亲、最左的孩子、右兄弟

  结点的度不定，给出这两种操作可以查找到一个结点的全部孩子
- 插入、删除：孩子
- 遍历：存在一对多的关系，给出一种有规律的方法遍历（有且仅访问一次）树中的结点

树的定义-ADT Tree

ADT Tree{

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

数据关系：若D为空集，则称为空树;

若D仅含一个数据元素，则R为空集，否则R={H}，H是如下二元关系：

(1) 在D中存在唯一的称为根的数据元素root，它在关系H下无前驱;

(2) 若D-{root}≠Φ，则存在D-{root}的一个划分D1, D2, …, Dm (m>0) （Di 表示构成第i棵子树的结点集），对任意j≠k (1≤j, k≤m) 有Dj∩Dk=Φ，且对任意的i (1≤i≤m)，唯一存在数据元素xi∈Di, 有<root, xi>∈H（H表示结点之间的父子关系）；

(3) 对应于D-{root}的划分，H-{<root, x1>,…, <root, xm>}有唯一的一个划分H1, H2, …, Hm(m>0)（Hi表示第i棵子树中的父子关系），对任意j≠k(1≤j,k≤m)有Hj∩Hk=Φ，且对任意i(1≤i≤m)，Hi是Di上的二元关系，(Di, {Hi})是一棵符合本定义的树，称为根root的子树。

}

树的定义-ADT Tree

基本操作：

InitTree(&T)
　　操作结果：构造空树T

DestroyTree(&T)
　　初始条件：树T已存在
　　操作结果：销毁树T

ClearTree(&T)
　　初始条件：树T已存在
　　操作结果：将树T清为空树

TreeEmpty(T)
　　初始条件：树T已存在
　　操作结果：若T为空树，则返回TRUE，否则返回FALSE

TreeDepth(T)
　　初始条件：树T已存在
　　操作结果：返回树T的深度

Root(T)
　　初始条件：树T已存在
　　操作结果：返回T的根

Value(T, cur_e)
　　初始条件：树T已存在，cur_e是T中某个结点
　　操作结果：返回cur_e的值

Assign(T, &cur_e, value)
　　初始条件：树T已存在，cur_e是T中某个结点
　　操作结果：结点cur_e赋值为value

Parent(T, cur_e)
　　初始条件：树T已存在，cur_e是T中某个结点
　　操作结果：若cur_e是T的非根结点，则返回它的双亲，否则函数值为“空”

LeftChild(T, cur_e)
　　初始条件：树T已存在，cur_e是T中某个结点
　　操作结果：若cur_e是T的非叶子结点，则返回它的最左孩子，否则返回“空”

RightSibling(T, cur_e)
　　初始条件：树T已存在，cur_e是T中某个结点
　　操作结果：若cur_e有右兄弟，则返回它的右兄弟，否则返回“空”

InsertChild(&T, p, i, c)
　　初始条件：树T已存在，p指向T中某个结点，1≤i≤p所指结点的度+1，非空树c与T不相交
　　操作结果：插入c为T中p所指结点的第i棵子树。

DeleteChild(&T, p, i)
　　初始条件：树T已存在,p指向T中某个结点，1≤i≤p所指结点的度
　　操作结果：删除T中p所指结点的第i棵子树。

TraverseTree(T)
　　初始条件：树T已存在，visit是对结点操作的应用函数
　　操作结果：按某种次序对T的每个结点访问一次

}ADT Tree

定义-ADT Tree

- **ADT Tree**
  - 查找：Parent(T,cur_e) LeftChild(T, cur_e) RightSibling(T, cur_e)
  - 插入：InsertChild(&T, &p, i, c)
  - 删除：DeleteChild(&T, &p, i)
  - 遍历：TraverseTree(T)

本节内容

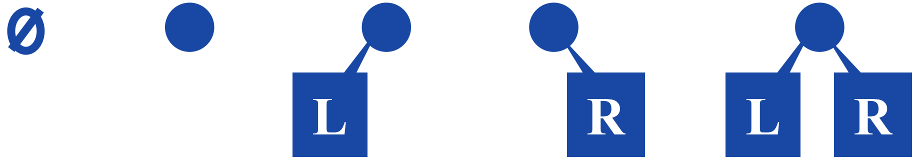
- **二叉树**
  - 定义和性质

- **二叉树**
  - 定义、ADT

二叉树

一般树的度不定，直接考虑其操作比较困难，故首先考虑度为二的树。这里引入二叉树。

- **二叉树的定义（递归定义）**
  - 一棵二叉树是结点的一个有限集合，该集合或者为空，或者是由一个根结点加上两棵分别称为左子树和右子树的、互不相交的二叉树组成。



二叉树的五种不同形态

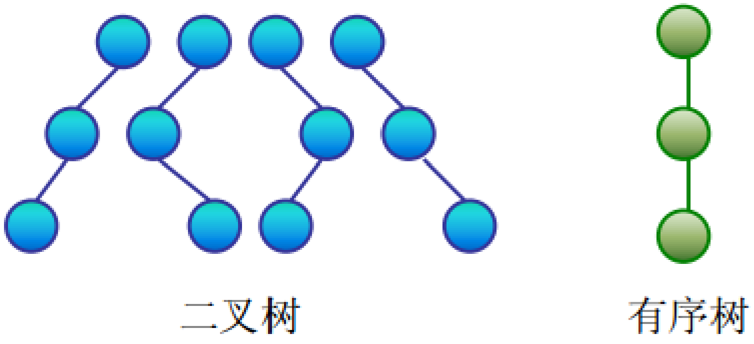
二叉树

➢ 二叉树的特殊性

- 0≤度≤2
- 子树有左右之分（子树的个数 = 1 或2时）注意：0≤度≤2的有序树≠二叉树

注意：0≤度≤2的有序树≠二叉树

当某个结点只有一棵子树时，不存在序的概念



二叉树-ADT BinaryTree

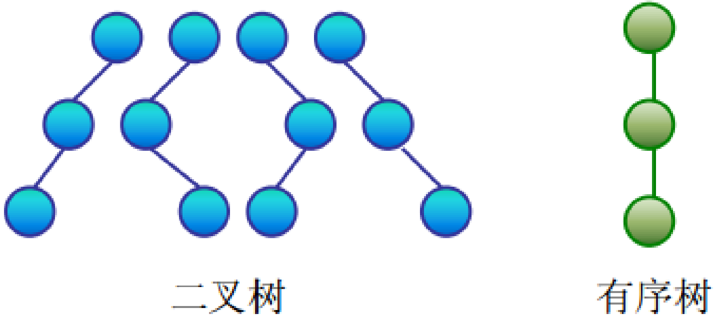
二叉树的特殊性

- 0≤度≤2
- 子树有左右之分（子树的个数 = 1 或2时）注意：0≤度≤2的有序树≠二叉树

注意：0≤度≤2的有序树≠二叉树

当某个结点只有一棵子树时，

不存在序的概念



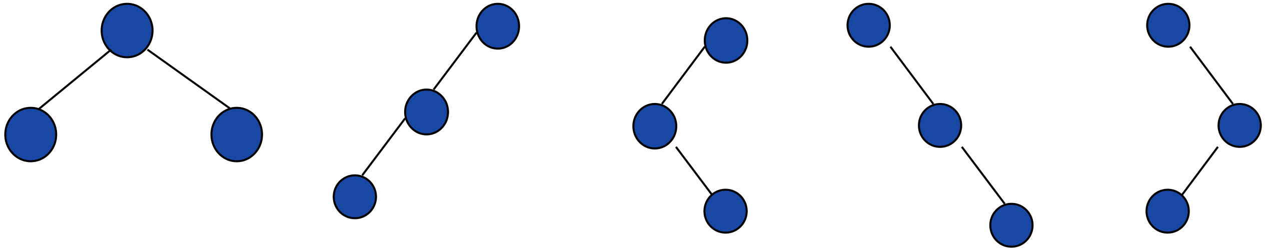
- 最左的孩子、右兄弟 ->左孩子、右孩子
- 遍历的规律性：　　L(左子树)、D(根)、R(右子树)的排列上限定为L在R前访问

为什么要研究二叉树?

- **理由一**
  - 普通树（多叉树）若不转化为二叉树，则运算很难实现。
- **理由二**
  - 二叉树的结构最简单，规律性最强；
- **理由三**
  - 可以证明，所有树都能转为唯一对应的二叉树，不失一般性。

练习

- **问题：**
  - 具有3个结点的二叉树可能有几种不同形态？普通树呢？



5种/2种

- **二叉树**
  - 性质

二叉树-性质（1,2,3）

二叉树的性质

- **性质1**：二叉树的第i层至多有2^{i-1}个结点(i≥1)
- **性质2**：深度为k的二叉树至多有2^{k}-1个结点(k≥1)

  思考： 性质1和性质2推广到 k叉树，结果会如何?

  k^{i-1} $(k^{h}-1)/(k-1)$

- **性质3**：对任何一棵二叉树, 如果其叶结点有 n_0 个, 度为2的非叶结点有 n_2 个, 则有

  $$n_0 = n_2 + 1$$

  结点总数 $n= n_0 + n_1 + n_2$

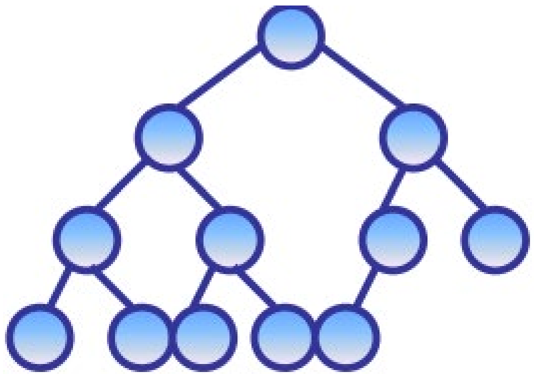
  分支数 n-1(**去掉根**) = $n_1+2\times n_2$

  思考： 若包含有n个结点的树中只有叶子结点和度为 k的结点，则该树中有多少叶子结点?

  $n = n_0+n_k,\ n-1 = kn_k \Rightarrow n_0 = n-(n-1)/k$

二叉树-性质

- **满二叉树**：一棵深度为k且有$2^k - 1$个结点的二叉树(k≥0)
- **完全二叉树**：对于深度为k的完全二叉树，则
  1) 前k-1层为满二叉树；
  2) 第 k层结点依次占据最左边的位置；
  3) 一个结点有右孩子，则它必有左孩子；
  4) 度为 1的结点个数为 0或1
  5) 叶子结点只可能在层次最大的两层上出现；
  6) 对任一结点，若其右分支下的子孙的最大层次为 l, 则其左分支下的子孙的最大层次必为 l 或 l + 1。



二叉树的性质

- **性质1**：二叉树的第i层至多有2^{i-1}个结点(i≥1)
- **性质2**：深度为k的二叉树至多有2^k-1个结点(k≥1)

至多➔至少?

第i层上至少有_1_个结点?
深度为k时至少有_k_个结点?

二叉树-性质（4）

- **性质4**：具有n个结点的完全二叉树的深度为 $\lfloor \log_2 n \rfloor + 1$

  由性质2

  $2^{k-1}-1 < n \le 2^k-1$ 或 $2^{k-1} \le n < 2^k$

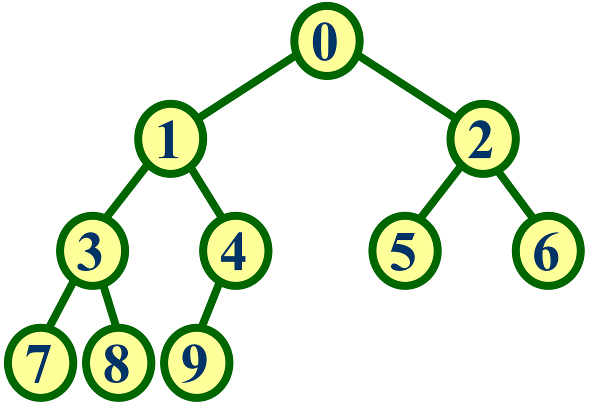
  于是 $k-1 \le \log_2 n < k$

例：若一个完全二叉树有**1450**个结点，则度为 **1**的结点个数为　　，度为 **2**的结点个数为　　　，叶子结点的个数为　　　，有　　　个结点有左孩子，有　　　个结点有右孩子；该树的高度为　　　。 **(性质3、性质4以及完全二叉树的特征)**

二叉树-性质（5）

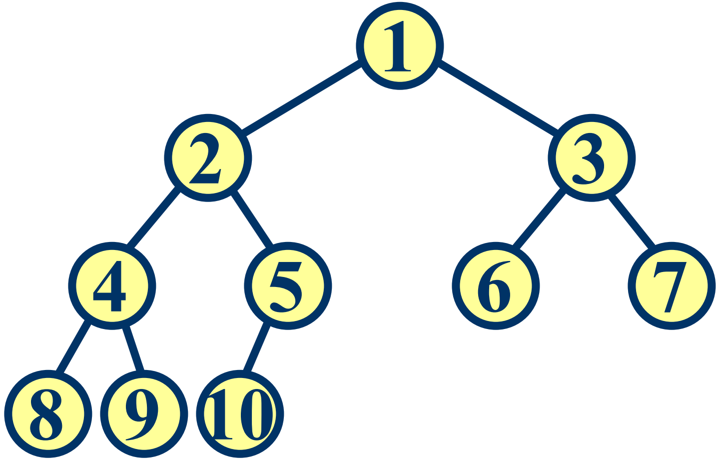
- **性质5**：如果对一棵有n个结点的完全二叉树的结点按层序从0开始编号(从第1层到第 $\lfloor \log_2 n \rfloor + 1$ 层，每层从左到右)，则对任一结点i ($1 \le i \le n$)，有
  (1) 如果i=0，则结点i是二叉树的根，无双亲；如果i >0,则其双亲是结点 $\lfloor (i-1)/2 \rfloor$；
  (2) 如果$2i$ +1>= n，则结点i无左孩子(结点i为叶子结点)；否则其左孩子是结点$2i$+1；
  (3) 如果$2i$ +2> =n，则结点i无右孩子；否则其右孩子是结点$2i$ + 2。
- (4) 如果i为偶数且i !=0，则其左兄弟为i-1；
  (5) 如果i为奇数且i !=n-1，则其右兄弟为i+1；

思考： 性质5推广到 k叉树，结果会如何?



二叉树-性质（5）

- **性质5**：如果对一棵有n个结点的完全二叉树的结点按层序从1开始编号(从第1层到第$\lfloor \log_2 n \rfloor + 1$ 层，每层从左到右)，则对任一结点i ($1 \le i \le n$)，有
  (1) 如果i=1，则结点i是二叉树的根，无双亲；如果i >1,则其双亲是结点 $\lfloor i/2 \rfloor$;
  (2) 如果$2i > n$，则结点i无左孩子(结点i为叶子结点)；否则其左孩子是结点$2i$;
  (3) 如果$2i + 1 > n$，则结点i无右孩子；否则其右孩子是结点$2i + 1$。
- (4) 如果i为奇数且i !=1，则其左兄弟为i-1;
  (5) 如果i为偶数且i !=n，则其右兄弟为i+1;



思考：　性质5推广到 k叉树，结果会如何?

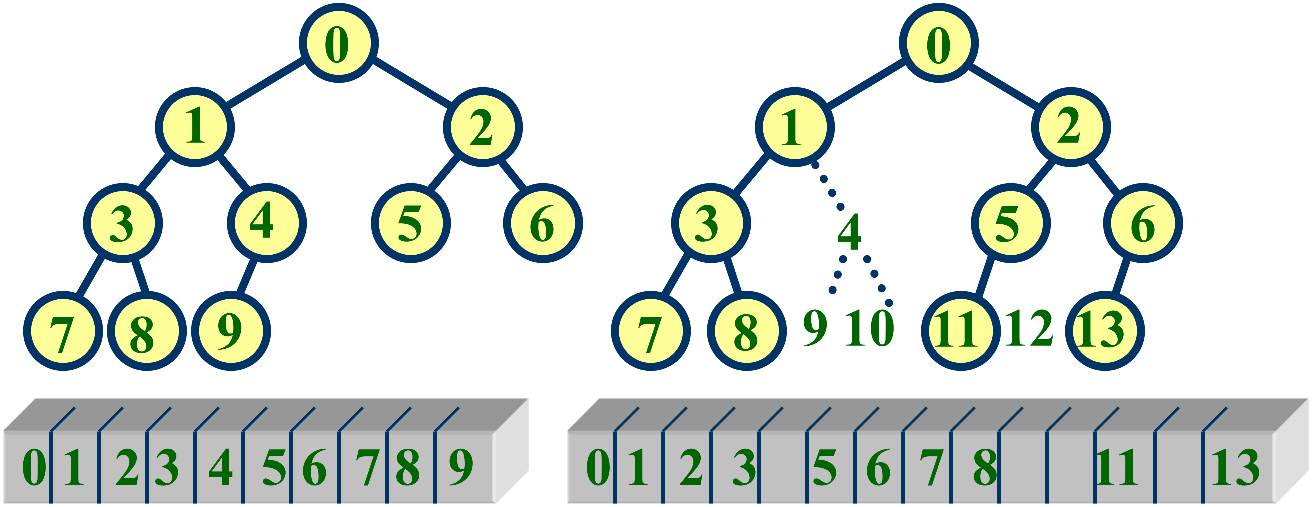
本节内容

- **二叉树**
  - 存储

二叉树的顺序存储结构

- 类型定义
  - 通过补虚结点，将一般的二叉树变成完全二叉树空间开销大！
  - typedef ElemType SqBiTree[MAX_TREE_SIZE];//0号单元存储根结点
  - 1) 依据性质5，用一组地址连续的存储单元依次自上而下、自左至右存储完全二叉树上的结点元素；——结点在存储区中的相对位置反映它们逻辑上的关系
  - 2) 仅适用于完全二叉树
- 一般二叉树的顺序存储方法
  - 通过补虚结点，将一般的二叉树变成完全二叉树空间开销大！

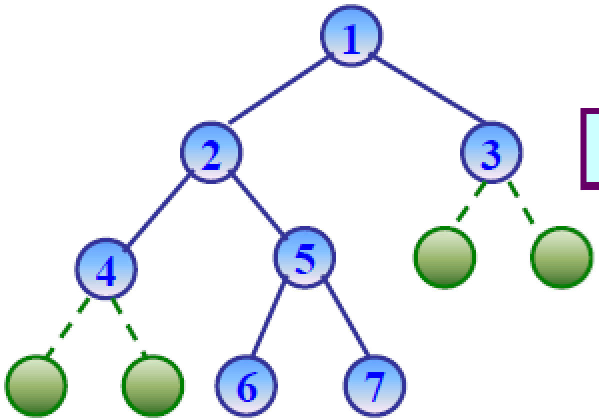
二叉树-顺序存储结构



完全二叉树的顺序表示

一般二叉树的顺序表示

二叉树-顺序存储结构



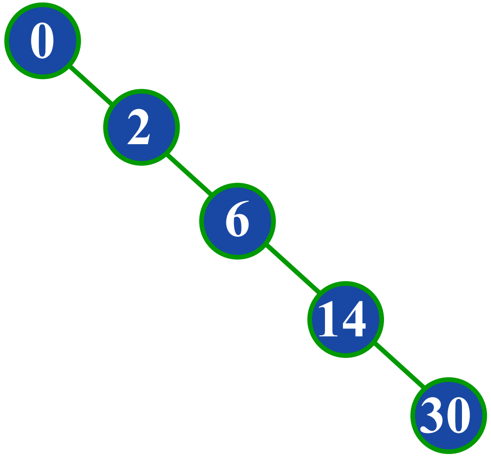
1	2	3	4	5	0	0	0	0	6	7

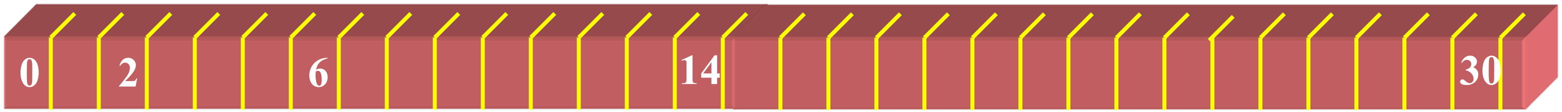
- 空间利用率问题
  - 在最坏情况下，一个深度为k且只有k个结点的单支树(树中不存在度为2的结点)，则需要长度为2k-1的一维数组。

二叉树-顺序存储结构

极端情形: 只有右单支的二叉树

- 对于完全二叉树，因结点编号连续，数据存储密集，适于用顺序表示;
- 对于一般二叉树，用链表表示较好;





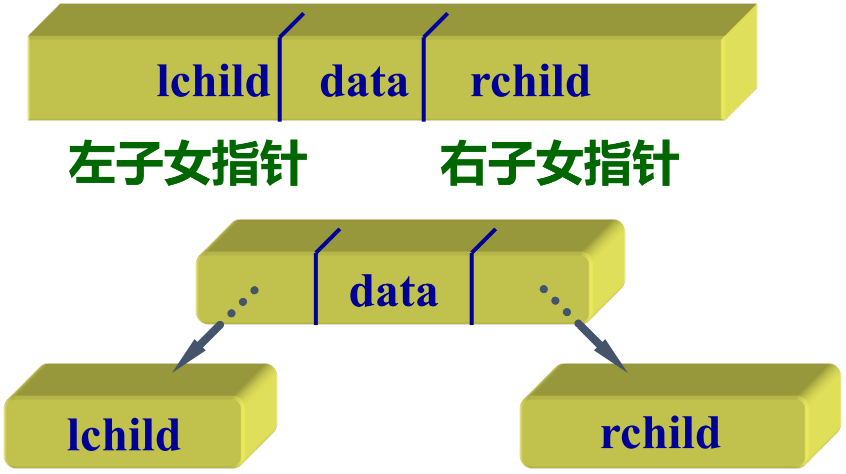
二叉树的链式存储结构

- 引入辅助空间表示结点之间的关系：双亲-孩子
  - 二叉链表(左、右孩子链域)
  - 三叉链表(双亲及左、右孩子链域)
- 二叉链的类型定义(动态链表)

```
typedef struct BiTNode{
    ElemType data;
    struct BiTNode *lchild, *rchild; // 左右孩子指针
}BiTNode, *BiTree;
```

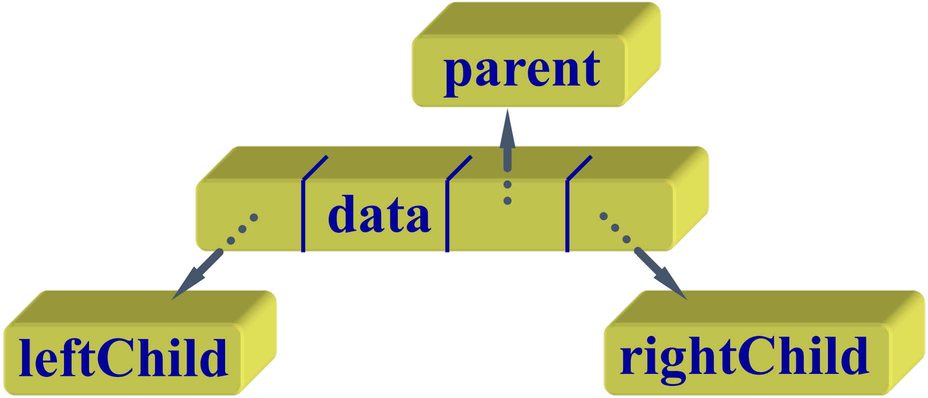
  - 若有n个结点，则共有2n个链域；其中n-1不为空，指向孩子；另外n+1个为空链域

二叉树的二叉链表表示



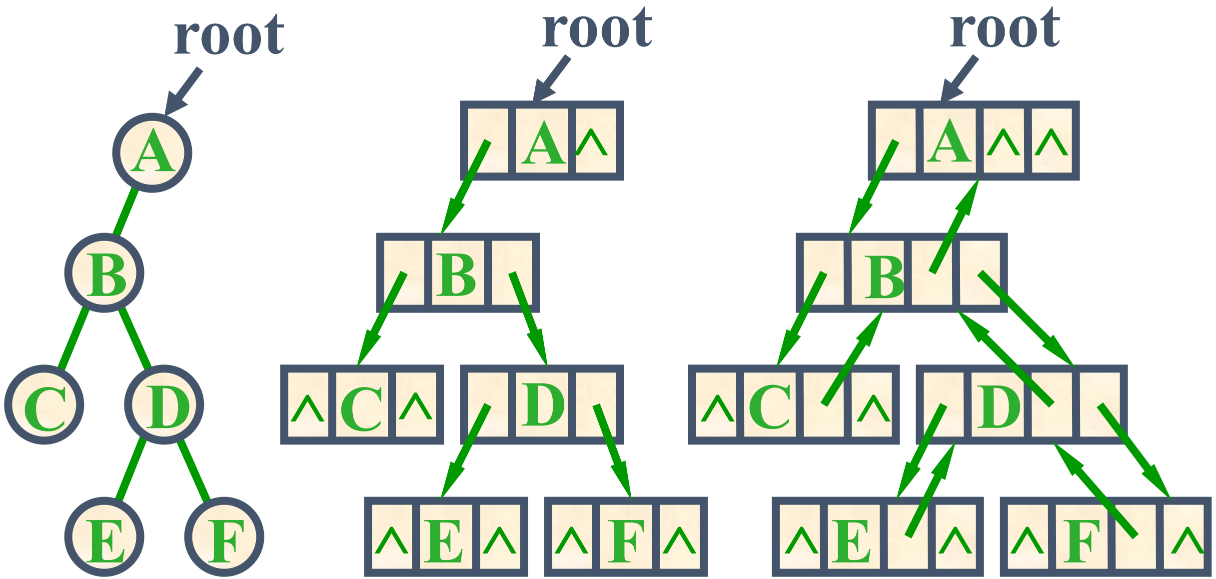
- 使用二叉链表，找子女的时间复杂度为O(1)，找双亲的时间复杂度为$O(\log_2 i) \sim O(i)$，其中，i 是该结点编号。

二叉树的三叉链表表示



- 使用三叉链表，找子女、双亲的时间都是O(1)。

二叉树链表表示的示例



二叉树　二叉链表　三叉链表