数据结构与算法

多维数组 & 广义表

主讲：赵海燕

北京大学信息科学技术学院

“数据结构与算法”教学组

国家精品课“数据结构与算法”

http://www.jpk.pku.edu.cn/pkujpk/course/sjjg/

张铭，王腾蛟，赵海燕

高等教育出版社，2008. 6, “十一五”国家级规划教材

主要内容

- 多维数组
  - 基本概念
  - 数组的空间结构
  - 数组的存储
  - 用数组表示特殊矩阵
  - 稀疏矩阵
- 广义表和存储管理
- Trie结构和Patricia树

基本概念

- 数组（Array）是元素数量和类型**固定**的有序序列
  - 静态数组必须在定义时指定其大小和类型
  - 动态数组可在程序运行时分配内存空间
- 多维数组（Multi-array）是向量的扩充
  - 向量的向量组成了多维数组
  - 可表示为

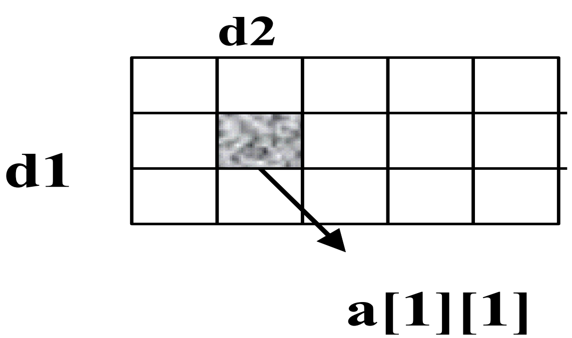
$$\text{ELEM A}[c_1..d_1][c_2..d_2]\ldots[c_n..d_n]$$

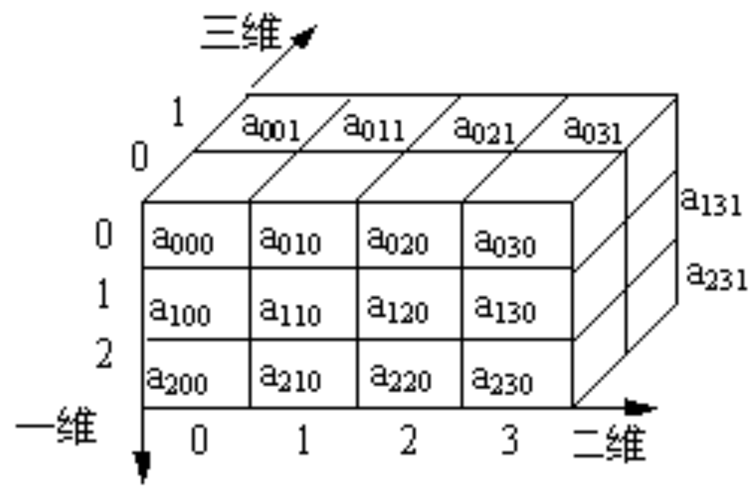
c_i 和 d_i 是各维下标的下界和上界。其元素个数为：

$$\prod_{i=1}^{n}(d_i - c_i + 1)$$

数组的空间结构

d1=3,d2=5





二维数组

三维数组

d1[0..2],d2[0..3],d3[0..1]分别为3个维

数组的存储

- 内存是一维的，故数组的存储也只能一维
  - 以行为主序（也称“**行优先**”）
  - 以列为主序（也称“**列优先**”）

$$X = \begin{vmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{vmatrix}$$

列优先存储

- FORTRAN等语言采用列优先
  - 先排最左的下标
  - 从左向右
  - 最后最右的下标

列优先存储 a[1..k, 1..m, 1..n]

a_{111} a_{211} a_{311} … a_{k11} a_{*11}

a_{121} a_{221} a_{321} … a_{k21} a_{*21}

a_{**1}

…………………………

a_{1m1} a_{2m1} a_{3m1} … a_{km1} a_{*m1}

a_{112} a_{212} a_{312} … a_{k12}

a_{122} a_{222} a_{322} … a_{k22}

a_{**2}

…………………………

a_{1m2} a_{2m2} a_{3m2} … a_{km2}

⋮

a_{11n} a_{21n} a_{31n} … a_{k1n}

a_{12n} a_{22n} a_{32n} … a_{k2n}

…………………………

a_{1mn} a_{2mn} a_{3mn} … a_{kmn}

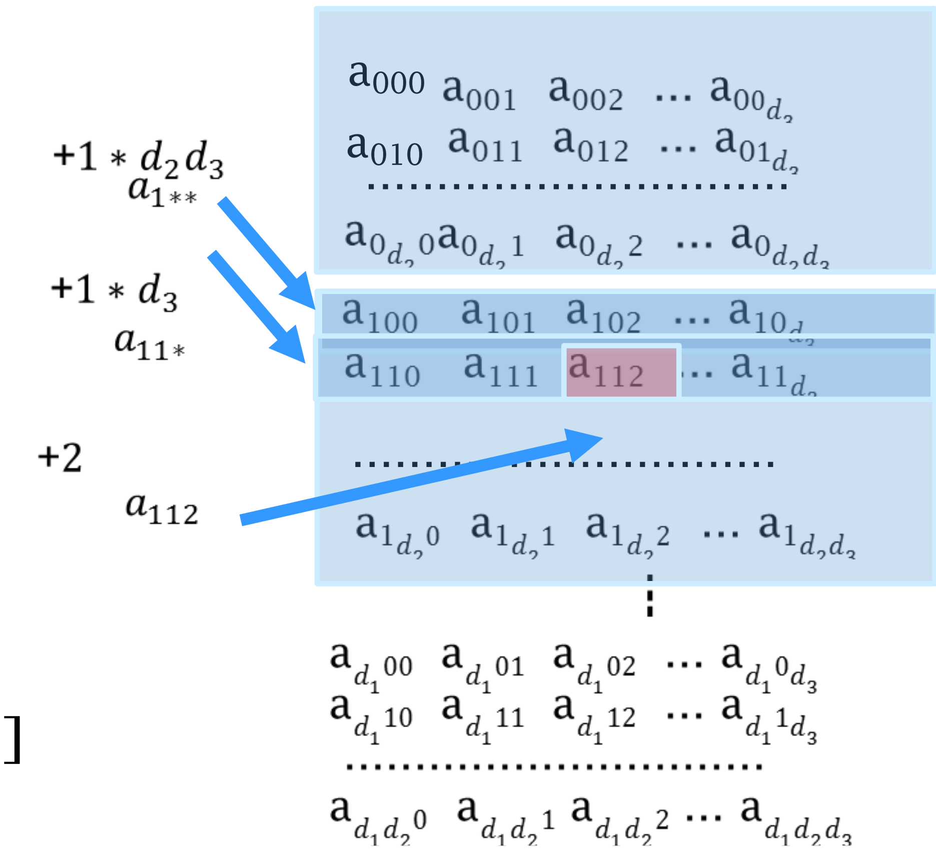
行优先存储

- C/C++、 Pascal等采用行优先
  - 先排最右的下标
  - 从右向左
  - 最后最左的下标

行优先存储

- C++ 多维数组ELEM A[d_1][d_2]…[d_n];

$$
\begin{aligned}
& loc(A[j_1, j_2, \ldots, j_n]) \\
&= loc(A[0,0,\ldots,0]) \\
&\qquad + d \cdot [j_1 \cdot d_2 \cdot \ldots \cdot d_n \\
&\qquad + j_2 \cdot d_3 \cdot \ldots \cdot d_n \\
&\qquad + \ldots \\
&\qquad + j_{n-1} \cdot d_n + j_n] \\
&= loc(A[0,0,\ldots,0]) \\
&\qquad + d \cdot [\sum_{i=1}^{n-1} j_i \prod_{k=i+1}^{n} d_k + j_n]
\end{aligned}
$$



用数组表示特殊矩阵

- 三角矩阵
  - 上三角
  - 下三角
- 对称矩阵
- 对角矩阵
- 稀疏矩阵

下三角矩阵示例

- 一维数组 list[0.. $(n^2+n)/2-1$]
  - 矩阵元素 $a_{i,j}$ 与线性表相应元素的对应位置为 list[$(i^2+i)/2 + j$] ($i>=j$)

$$
\begin{pmatrix}
0 \\
0 & 0 \\
7 & 5 & 0 \\
0 & 0 & 1 & 0 \\
9 & 0 & 0 & 1 & 8 \\
0 & 6 & 2 & 2 & 0 & 7
\end{pmatrix}
$$

对称矩阵

$$\begin{bmatrix} 0 & 3 & 0 & 15 \\ 3 & 0 & 4 & 0 \\ 0 & 4 & 0 & 6 \\ 15 & 0 & 6 & 0 \end{bmatrix}$$

- 元素满足 $a_{i,j}= a_{j,i}$，0 <= （i, j） <n
  - e.g.，无向图的相邻矩阵
- 只存储其下三角的值，对称关系映射
  - 存储于一维数组s[0..n(n+1)/2-1]
  - s[k]和矩阵元素$a_{i,j}$之间存在着一一对应的关系：

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

对角矩阵

- 所有非零元素都集中在主对角线及以其为中心的其他对角线上
  - e.g.，三对角矩阵:

    若|i-j|>1，则数组元素a[i][j] = 0

$$
\begin{pmatrix}
a_{0,0} & a_{0,1} & \cdots\cdots & & 0 \\
a_{1,0} & a_{1,1} & a_{1,2} & \cdots\cdots & \\
 & & & & \\
\cdots\cdots & & & & a_{n-2,n-1} \\
0 & & & a_{n-1,n-2} & a_{n-1,n-1}
\end{pmatrix}
$$

稀疏矩阵

- 非零元素**极少**，且分布不规律的矩阵

$$\mathbf{A}_{6\times 7} = \begin{pmatrix} 0 & 0 & 0 & 7 & 0 & 0 & 5 \\ 0 & 15 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 17 & 0 \\ 0 & 78 & 0 & 0 & 0 & 22 & 0 \\ 11 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 42 & 0 & 0 & 0 & 0 \end{pmatrix}$$

稀疏矩阵

- 稀疏因子
  - 在$m \times n$的矩阵中，有 t 个非零元素，则稀疏因子δ：

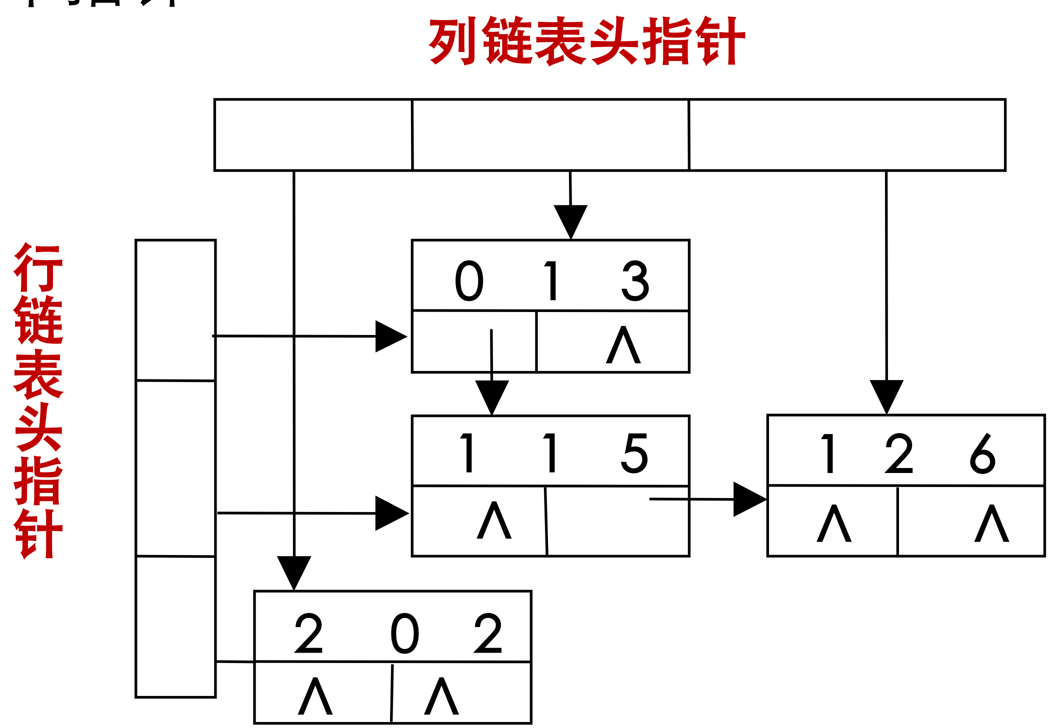
$$\delta = \frac{t}{m \times n}$$

  - 当 $\delta < 0.05$ 时，可认为是稀疏矩阵
- 采用**三元组**(i, j, a_{ij})表示矩阵元素
  - i为元素所在**行号**
  - j为元素所在**列号**
  - a_{ij}是元素的**值**

稀疏矩阵的十字链表

- 由两组链表组成
  - 行和列的指针序列
  - 每个结点都包含两个指针：
    - 同一行的后继
    - 同一列的后继

$$\begin{bmatrix} 0 & 3 & 0 \\ 0 & 5 & 6 \\ 2 & 0 & 0 \end{bmatrix}$$



经典矩阵乘法

- A[c1..d1][c3..d3], B[c3..d3][c2..d2], C[c1..d1][c2..d2]
- C＝A×B，其中

$$C_{ij} = \sum_{k=c3}^{d3} A_{ik} \bullet B_{kj}$$

```
for (i=c1; i<=d1; i++)
        for (j=c2; j<=d2; j++) {
                sum = 0;
                for (k=c3; k<=d3; k++)
                        sum += A[i, k]*B[k, j];
                C[i, j ] = sum;
        }
```

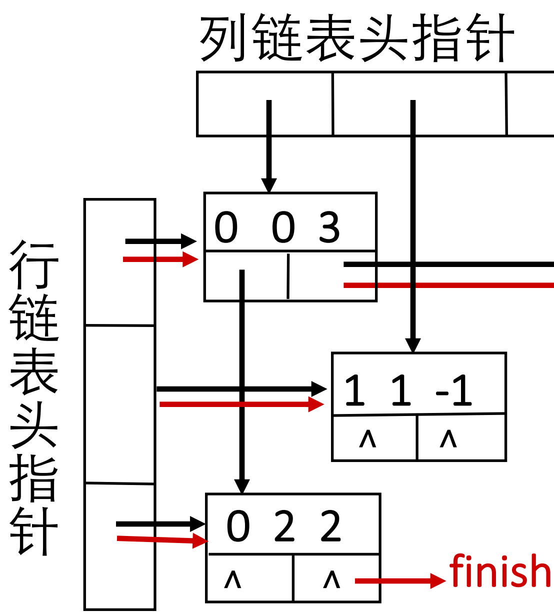
经典矩阵乘法时间代价

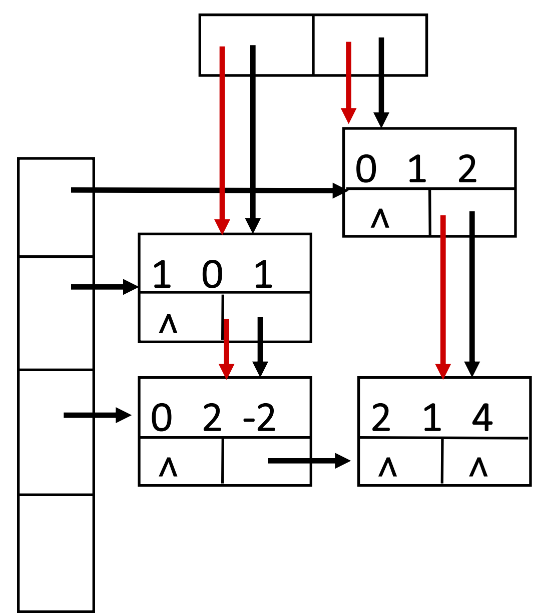
- $p = d1 - c1 + 1$， $m = d3 - c3 + 1$， $n = d2 - c2 + 1$；
- A 为 $p \times m$ 的矩阵，B 为 $m \times n$ 的矩阵，乘得的结果 C为 $p \times n$ 的矩阵
- 经典矩阵乘法所需要的时间代价为 O (p×m×n)

```
for (i=c1; i<=d1; i++)
   for (j=c2; j<=d2; j++){
      sum = 0;
      for (k=c3; k<=d3; k++)
         sum = sum + A[i,k]*B[k,j];
      C[i， j] = sum;
   }
```

稀疏矩阵乘法

$$\begin{bmatrix} 3 & 0 & 0 & 5 \\ 0 & -1 & 0 & 0 \\ 2 & 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 0 & 2 \\ 1 & 0 \\ -2 & 4 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} \mathbf{0} & \mathbf{6} \\ \mathbf{-1} & \mathbf{0} \\ \mathbf{0} & \mathbf{4} \end{bmatrix}$$





稀疏矩阵乘法时间代价

- 时间代价降为

  $O((t_a+t_b)\times p\times n)$

- 其中，
  - A为p×m的矩阵，其行向量的非零元素个数最多为t_a
  - B为m×n的矩阵，其列向量的非零元素个数最多为t_b
  - 乘积结果C为p×n的矩阵

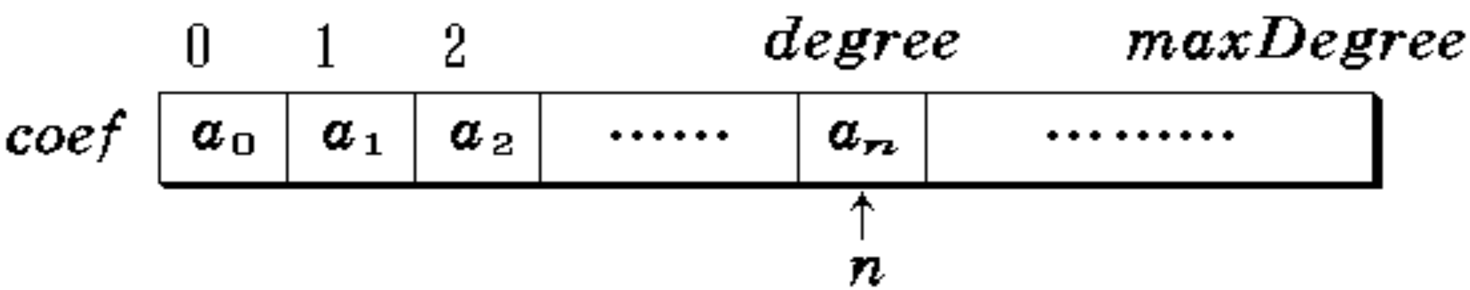
- 经典矩阵乘法所需要的时间代价为O(p×m×n)

稀疏矩阵的应用

- 一元多项式

$$P_n(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

$$= \sum_{i=0}^{n} a_i x^i$$



主要内容

- 多维数组
  - 基本概念
  - 数组的空间结构
  - 数组的存储
  - 用数组表示特殊矩阵
  - 稀疏矩阵
- 广义表和存储管理
- Trie结构和Patricia树

广义表和存储管理

- 广义表
  - 基本概念
  - 广义表的各种类型
  - 广义表的存储
  - 广义表的周游算法
- 储存管理

基本概念

- 线性表回顾
  - 由*n*（n≥0）个数据元素组成的**有限有序**序列
  - 线性表的每个元素都具有相同的数据类型
- 若一个线性表存在一个或多个子表，则被称为**广义表**（Generalized Lists，也称 Multi-lists），一般记作：

  $L = (x_0, x_1, \ldots, x_i, \ldots, x_{n-1})$

广义表

$L = (x_0, x_1, \ldots, x_i, \ldots, x_{n-1})$

- L 为广义表的名称
- n 为其长度
- x_i （$0 \le i \le n-1$） 是 L 的成员
  - 或为 单个元素，即 原子（atom ）
  - 或为 一个广义表，即 子表（sublist）
- 深度
  - 表中元素都化解为原子后的括号层数

广义表

L = （x_0， x_1， …， x_i， …， x_{n-1}）

- 表头 *head* = x_0
- 表尾 *tail* = （x_1， …， x_{n-1}）
- **规模更小**的表

有利于存储和实现

如何利用head & tail 访问广义表中的每个元素?

广义表

- E.g.， 给定两个广义表S=((a，(b)，c)，((d)，e))，T=(f，(g，((h))，i，j))

d = ______________?______________

h = = ______________?______________

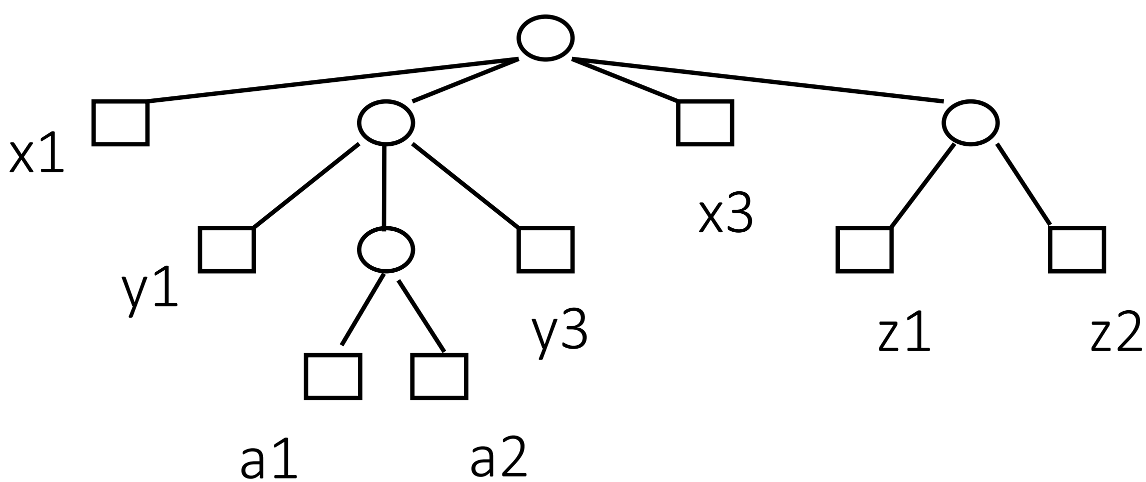
d = _____Head(Head(Head(Head(Tail(S)))))____

h = _____Head(Head(Head(Tail(Head(Tail(T))))))_____

广义表的分类

- 纯表（pure list）
  - 根结点到任一叶结点**只有一条路径**
  - 也即，任一元素（原子、子表）广义表中只出现一次

(x1, (y1 ,(a1 ,a2), y3), x3 ,(z1 ,z2))

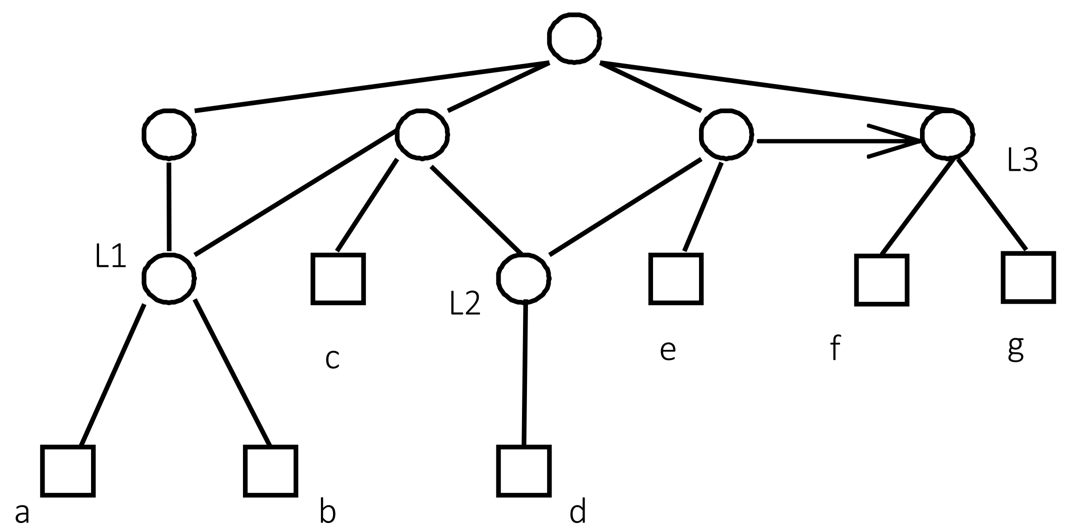


广义表的分类

- 可重入表
  - 其元素（包括原子和子表）可能会在表中多次出现
  - 若没有回路，图示对应于一个DAG
- 对子表和原子赋以标号

特例：循环表（即递归表）

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

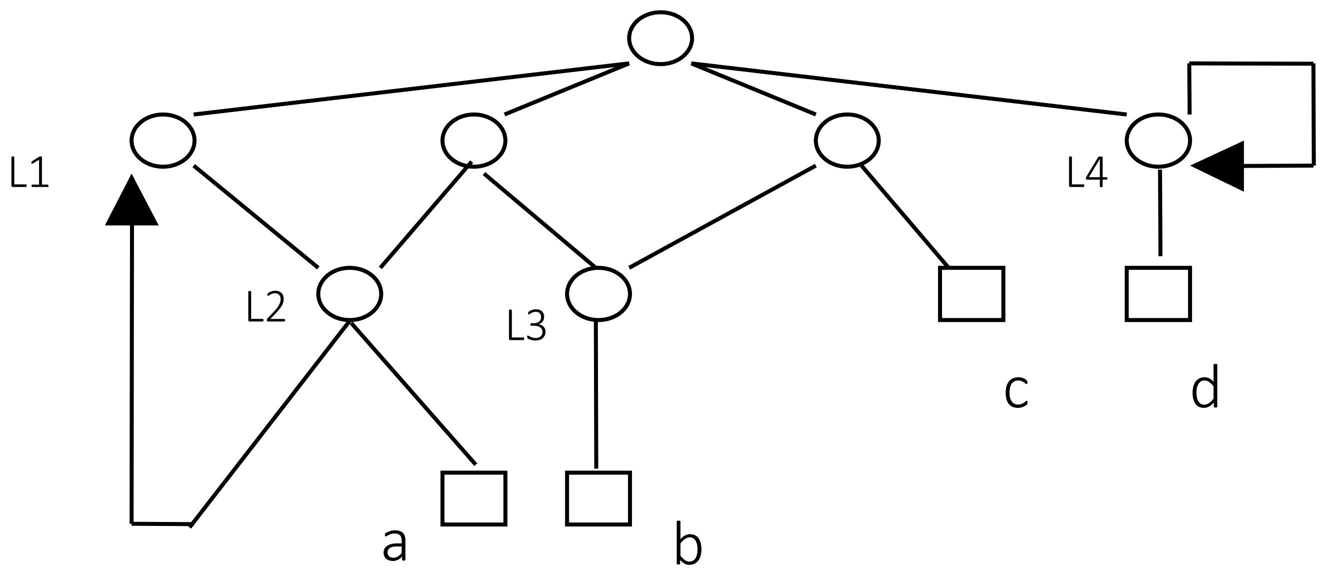


((L1: (a, b)), (L1, c, L2:(d)), (L2, e, L3:(f, g)), L3)

广义表的分类

- 循环表
  - 包含回路
  - 循环表的深度为无穷大

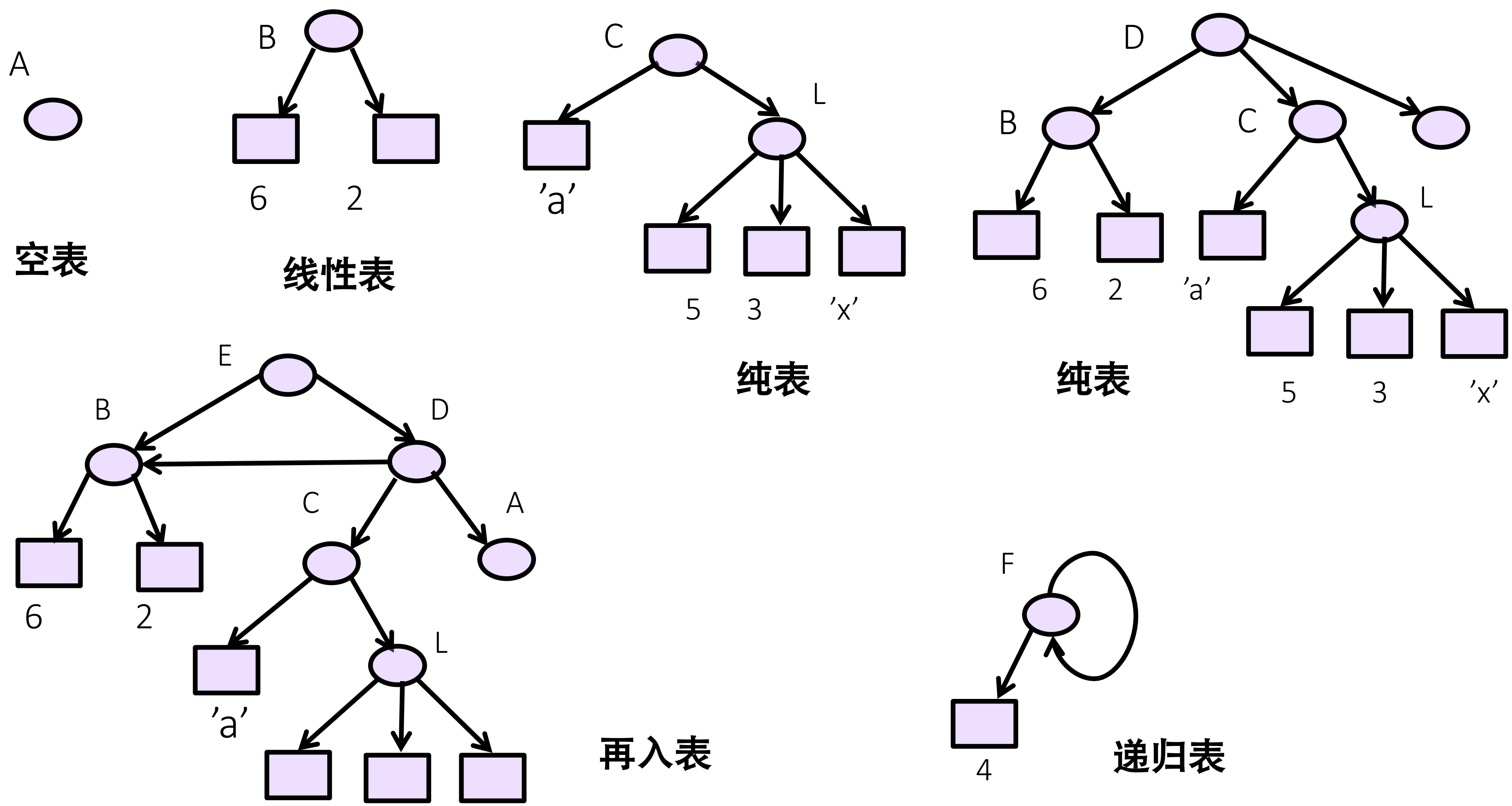
(L1:(L2:(L1, a)), (L2, L3:(b)), (L3, c), L4:(d,L4))



  - L1、L4调用自身

广义表的分类

E: (B: (6, 2) , D: (B, C:('a', L: (5, 3, 'x')) , A: ( )))



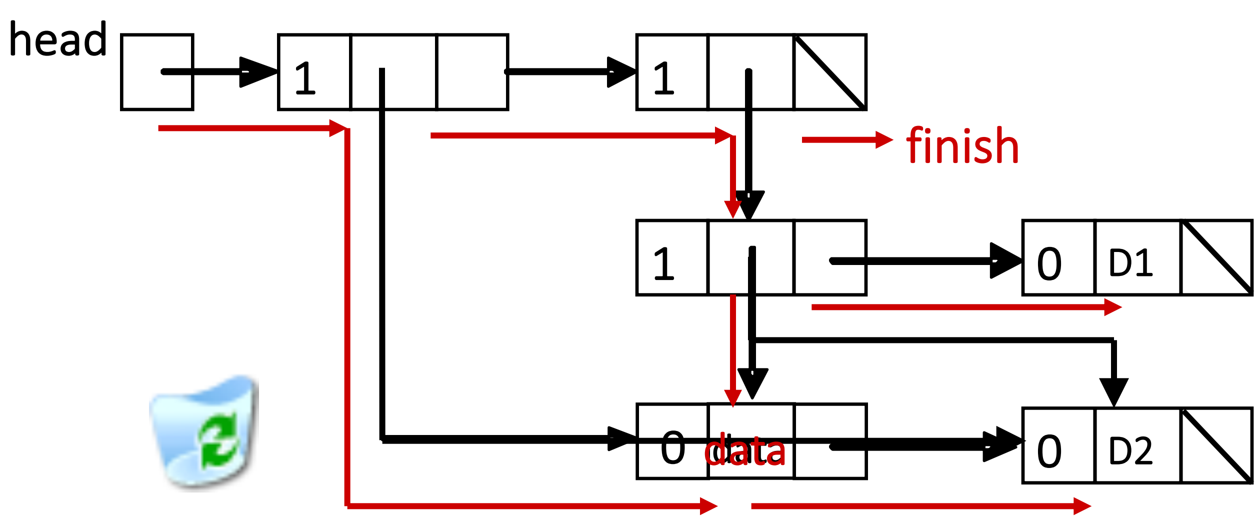
广义表的分类

- 图 ⊇ 再入表 ⊇ 纯表(树) ⊇ 线性表
  - 广义表是线性与树形结构的推广
- 递归表是有回路的再入表

- 广义表应用
  - 函数的调用关系
  - 内存空间的引用关系
  - LISP/Functional programming languages的处理对象

广义表存储ADT

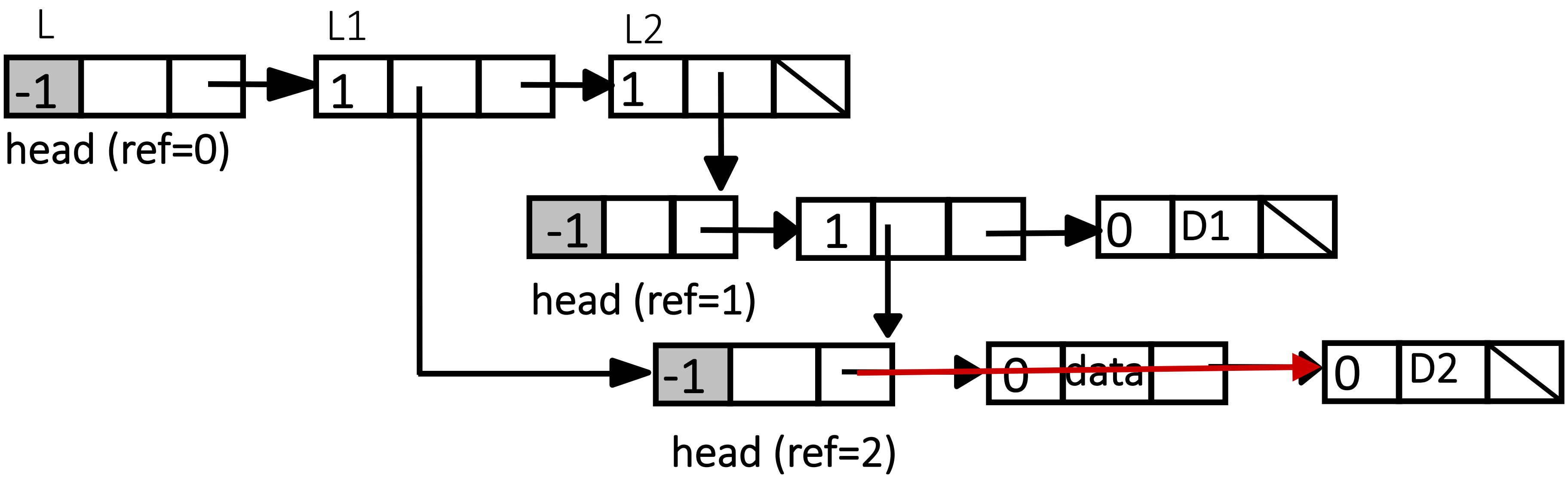
- 不带头结点的广义表链
  - 在删除结点的时候会出现问题
  - 删除结点data时链的调整费事



广义表存储ADT

- 增加头指针，简化删除、插入操作

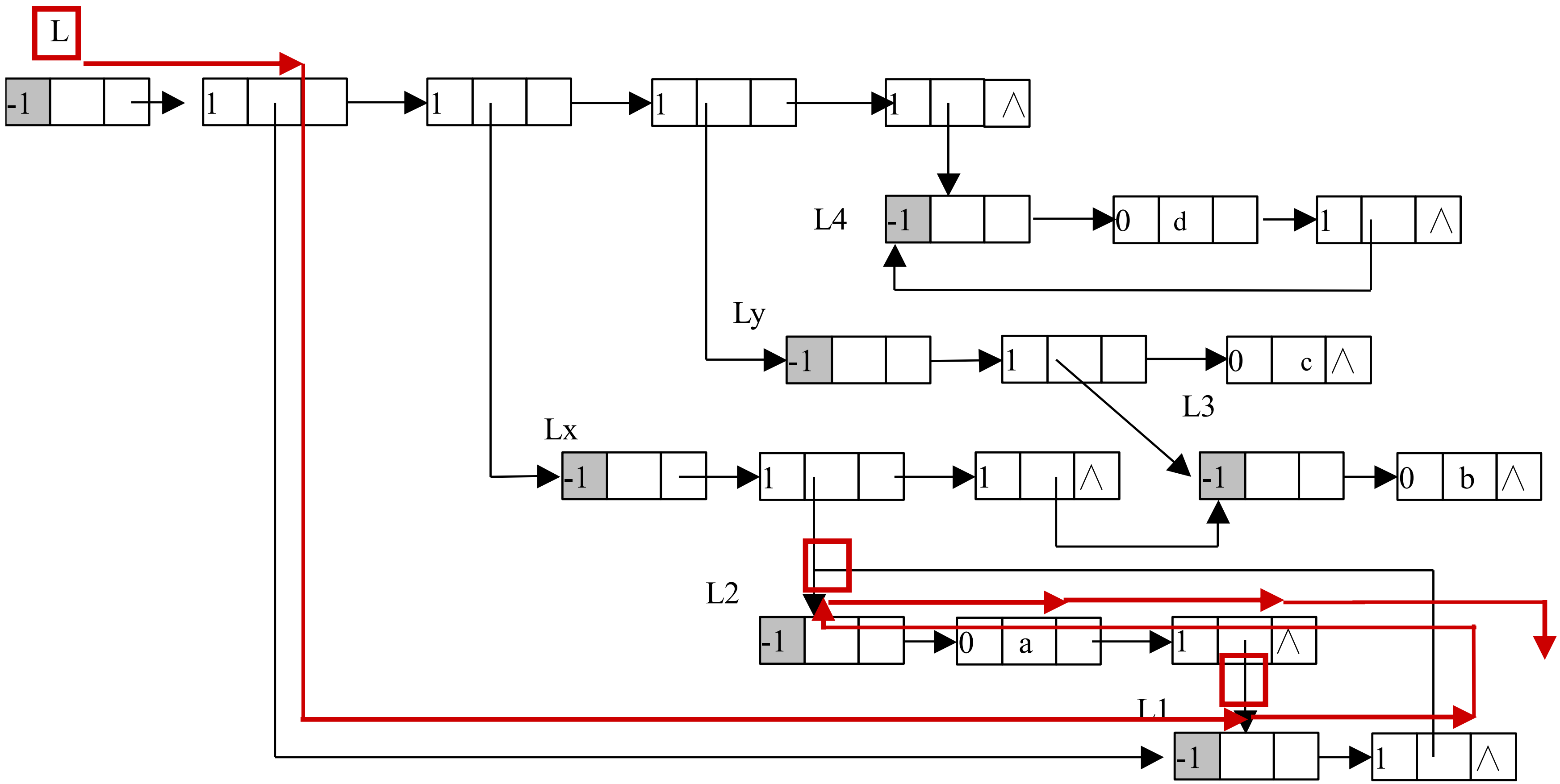
L: (L1: (data, D2) , L2: (L1, D1))



- 重入表，尤其是循环表
  - mark标志位 —— 图的因素

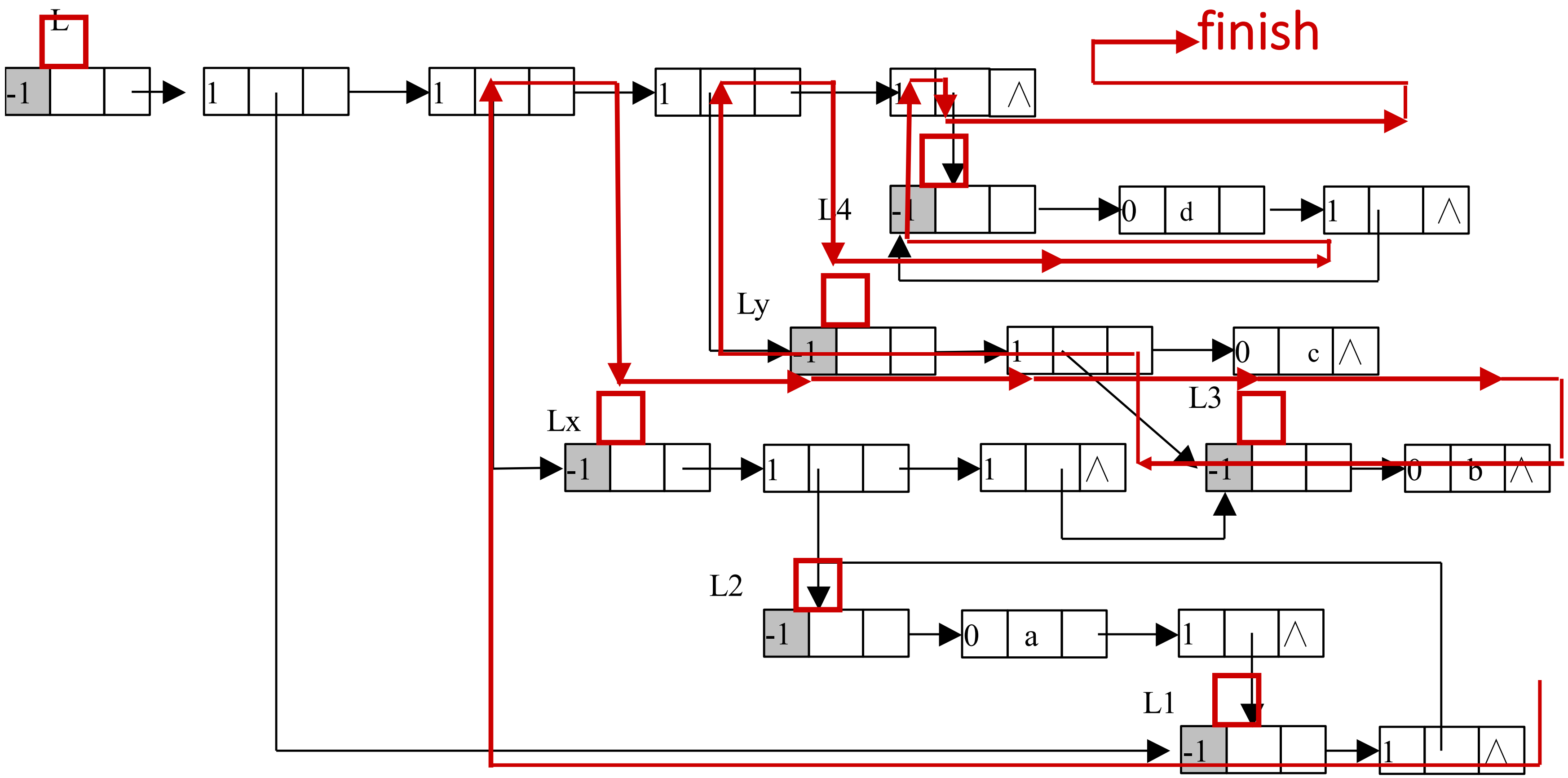
带头结点的循环广义表

(L1: (L2: (a,L1))



带头结点的循环广义表

(L1: (L2: (a ,L1)) , Lx :(L2 , L3 : (b)), Ly :(L3 , c) , L4 : (d , L4))



思考

- 广义表与树、图各有什么区别与联系?
- 怎么实现广义表遍历的算法?

广义表的应用：存储管理技术

- 内存管理存在的问题
- 可利用空间表
- 存储的动态分配和回收
- 内存管理技术
  - 链表、广义表
  - 伙伴系统
  - 失败处理策略和无用单元回收

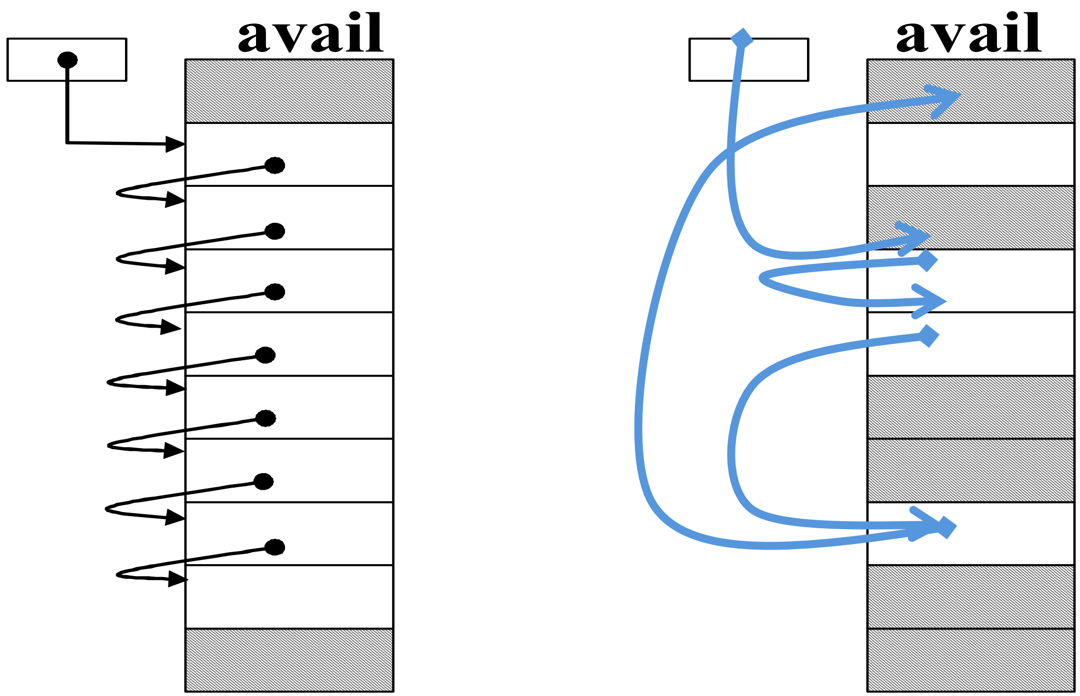
分配与回收

- 内存管理最基本的问题
  - 存储空间的分配
  - 回收被“释放”的存储空间
- 碎片问题
  - 存储压缩
- **无用单元收集**
  - 无用单元：可回收而未回收的空间
  - 内存泄漏(memory leak)
  - 程序员忘记delete不再使用的指针

可利用空间表

- 把存储器看成一组变长块数组，包括
  - 已分配的
  - 尚未分配的
    - 链接空闲块，形成可利用空间表(freelist)
- 存储分配和回收
  - new p 从可利用空间分配
  - delete p 把 p 指向的数据块返回可利用空间表
- 空间不够，则求助于失败处理策略

可利用空间表



（1） 初始状态的可利用空间表

（2） 系统运行一段时间后的可利用空间表

结点等长的可利用空间表

可利用空间表的函数重载

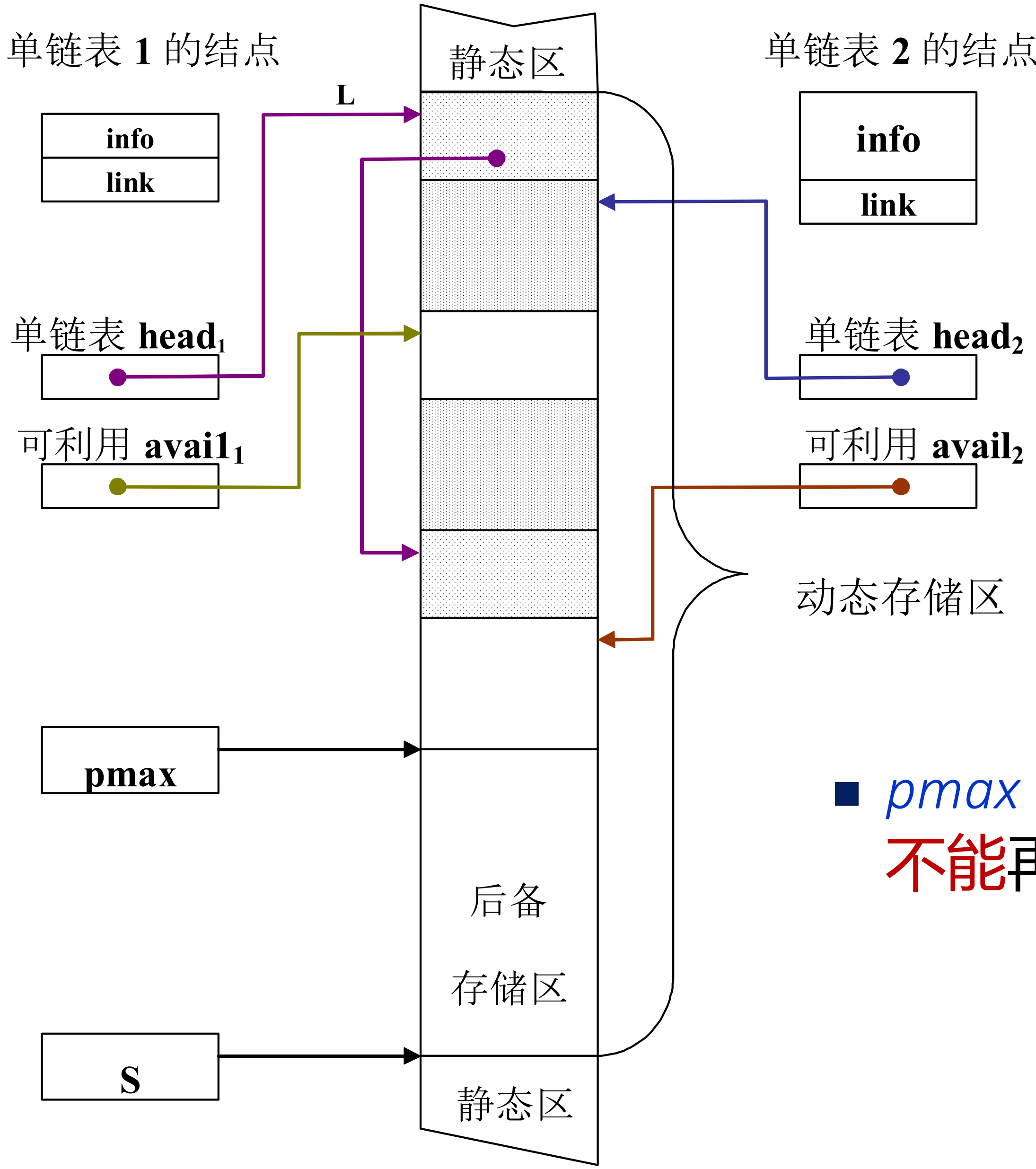
```
template <class Elem> class LinkNode{
private:
   static LinkNode *avail;              // 可利用空间表头指针
public:
   Elem value;                          // 结点值
   LinkNode * next;                     // 指向下一结点的指针
   LinkNode (const Elem & val, LinkNode * p) ;
   LinkNode (LinkNode * p = NULL) ;// 构造函数
   void * operator new (size_t) ;       // 重载new运算符
   void operator delete (void * p) ;    // 重载delete运算符
};
```

可利用空间表的函数重载

```
// 重载new运算符实现
template <class Elem>
void * LinkNode<Elem>::operator new (size_t) {
   if (avail == NULL)                              // 可利用空间表为空
     return ::new LinkNode;                        // 利用系统的new分配空间
   LinkNode<Elem> * temp = avail;                  // 从可利用空间表中分配
   avail = avail->next;
   return temp;
}
// 重载delete运算符实现
template <class Elem>
void LinkNode<Elem>::operator delete (void * p) {
   ( (LinkNode<Elem> *)  p) ->next = avail;
   avail =  (LinkNode<Elem> *) p;
}
```

可利用空间表

- 单链表栈
  - new， 即栈的删除操作
  - delete ，即栈的插入操作
- 直接引用系统的new和delete操作符，需要强制用 "::new p“ 和 "::delete p"
  - 程序运行完毕时，把avail所占用的空间都交还给系统（真正释放空间）



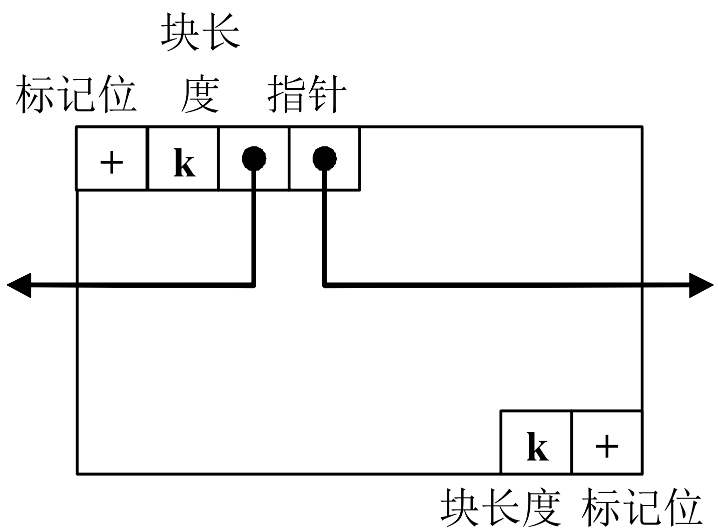
- *pmax* 已达到或超过 s，则不能再分配空间

存储的动态分配和回收

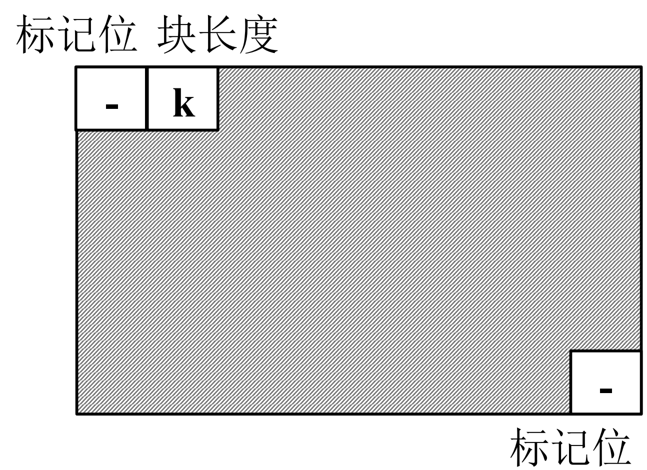
变长可利用块

- 分配
  1. 找到其长度大于等于申请长度的结点
  2. 从中截取合适的长度
- 回收
  - 考虑刚刚被删除的结点空间能否与邻接块合并
  - 以满足后续较大长度结点的分配请求

空闲块的数据结构

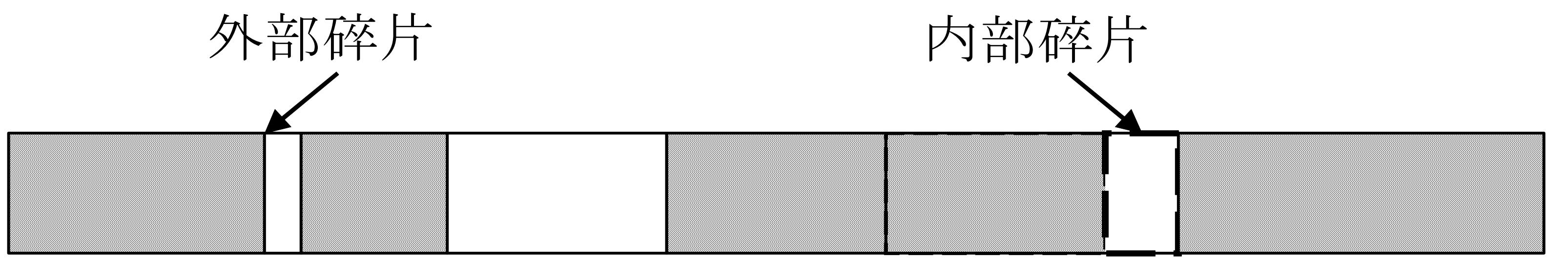


（a）空闲块的结构



（b）已分配块的结构

碎片问题



外部碎片和内部碎片

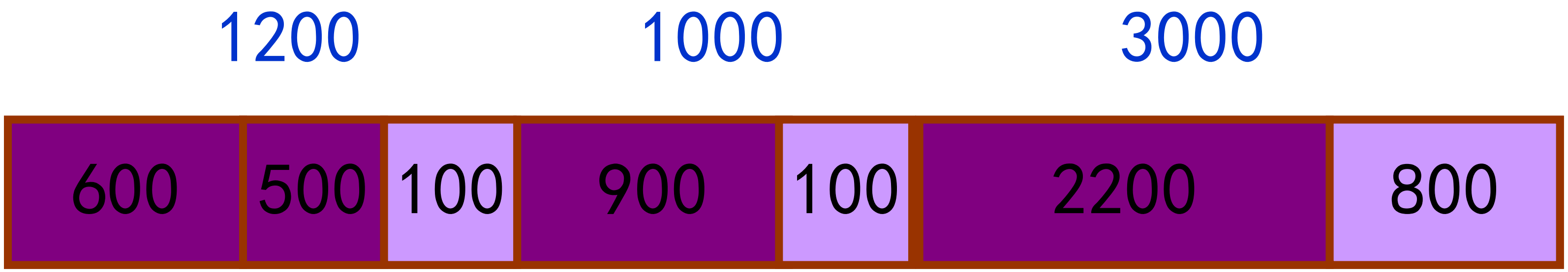
- 内部碎片
  - 因多于请求字节数的空间（剩余空间）
- 外部碎片
  - 小空闲块

顺序适配(sequential fit)

- 空闲块的分配方策略
  - 首先适配(first fit)
  - 最佳适配(best fit)
  - 最差适配(worst fit)

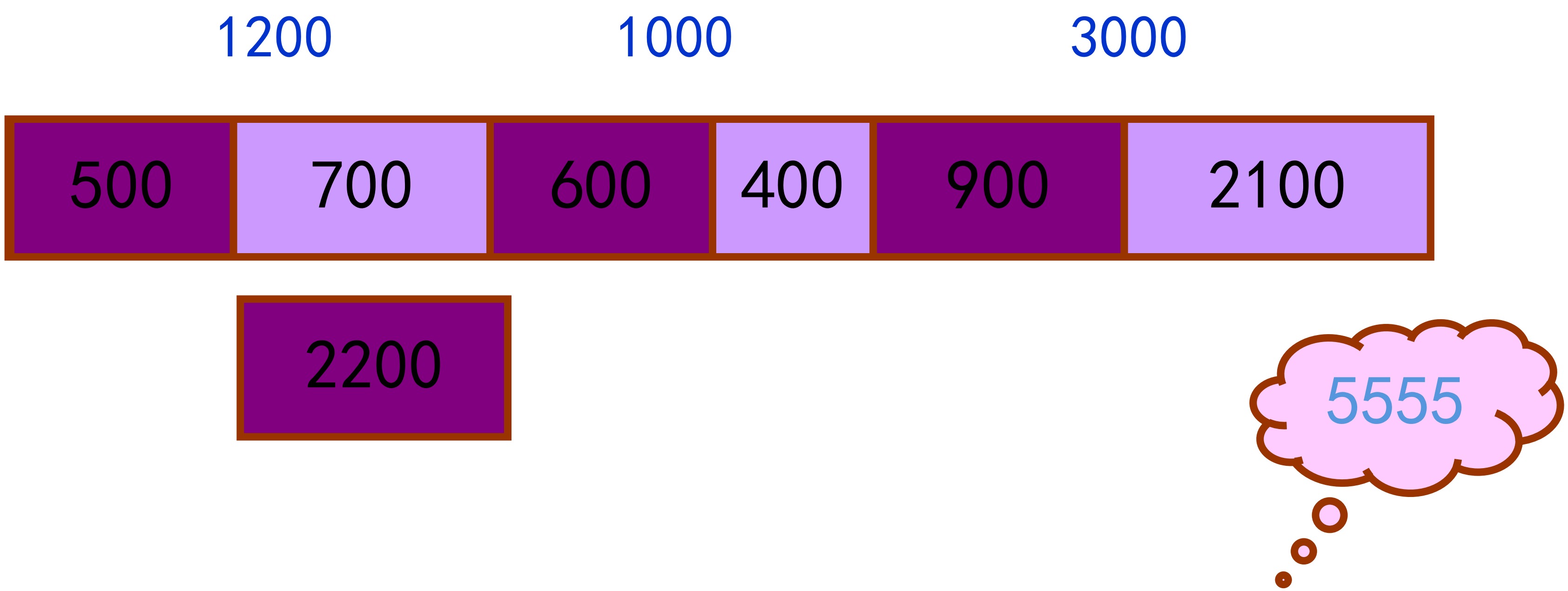
顺序适配

- 若有 三个可利用块，大小分别为 1200，1000，3000
  分配请求序列：600，500，900，2200
- 首先适配



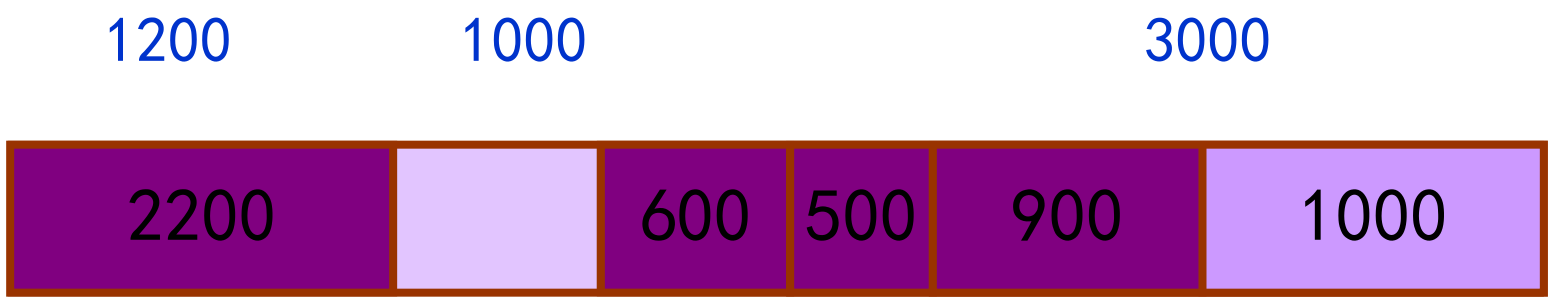
顺序适配

- **请求序列：**600，500，900，2200
- 最佳适配



顺序适配

- **请求序列：** 600，500，900，2200
- 最差适配



适配策略选择

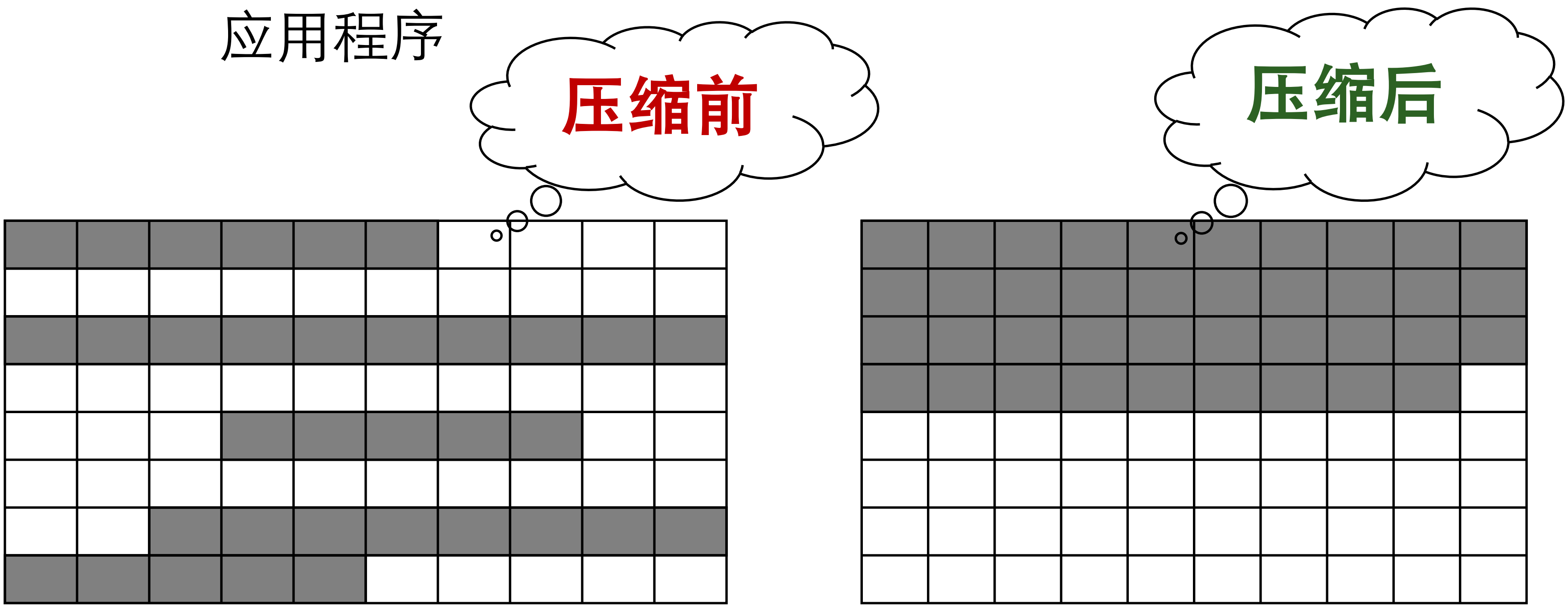
- 很难笼统地讲哪种适配策略最好
- 需考虑因素包括
  - 分配或回收效率对系统的重要性
  - 所分配空间的长度变化范围
  - 分配和回收的频率
- 在实际应用中，首先适配最常用
  - 分配和回收的速度比较快
  - 支持比较随机的存储请求

失败处理策略和无用单元回收

- 若内存不足而无法满足一个存储请求时，存储管理器可有两种行为
  - 直接返回一个系统错误信息
  - 使用失败处理策略(failure policy)来满足请求
    - 存储压缩（compact）
    - 无用单元收集和回收（Garbage Collection)

存储压缩

- 把内存中的所有**碎片集中**起来
  - 采用句柄使得存储地址相对化
    - 移动存储块位置，只需要修改句柄值，不需要修改应用程序



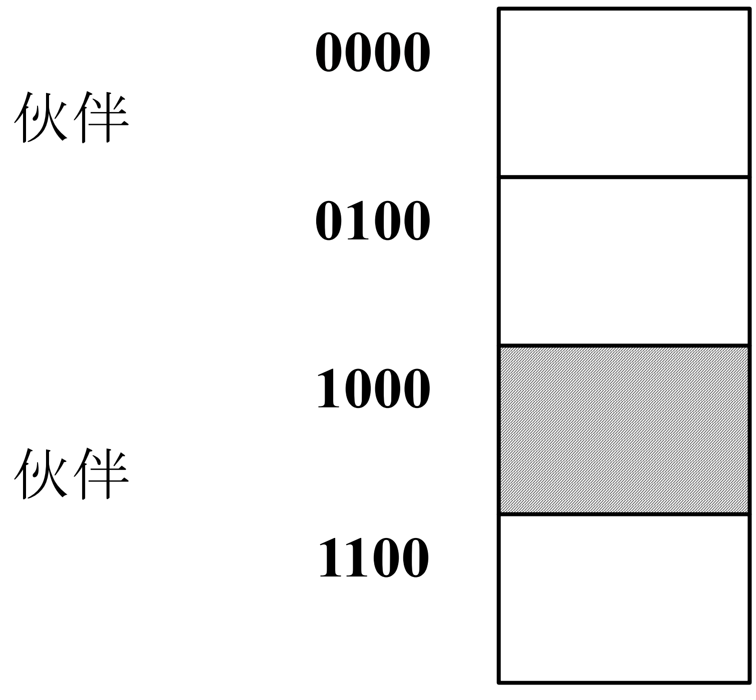
无用单元收集和回收

- 彻底的失败处理策略
  - 普查内存，标记那些不属于任何链的结点
  - 将被标记结点收集到可利用空间表中
  - 回收过程通常还可与存储压缩一起进行

伙伴系统（Buddy）

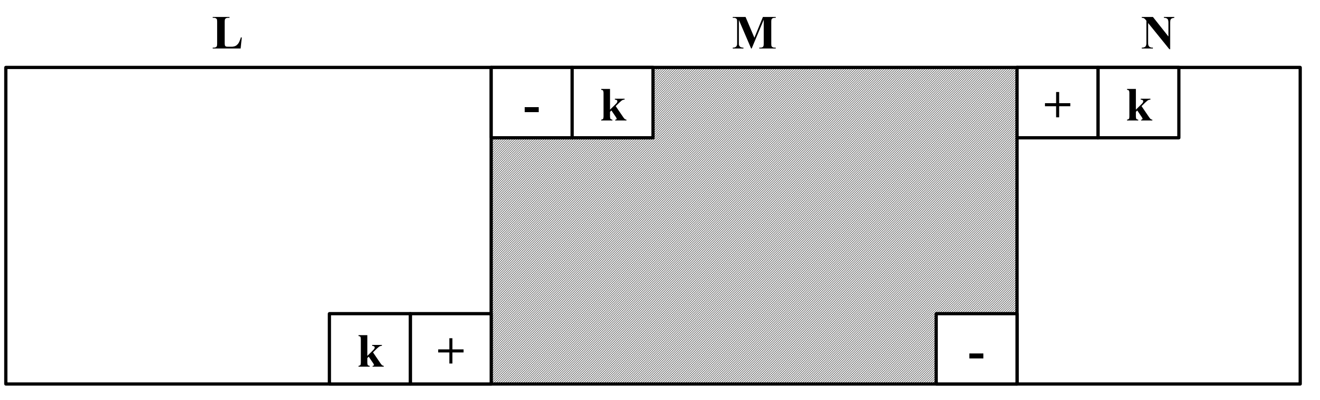
- 假设存储空间的大小为2^M，M为正整数
- 每个空闲块和已分配块的大小均为2^k，$0< k\leq M$
  - 2^k大小的块首地址与其伙伴的首地址相比，除了第k位外（最右为0位），其他所有位都相同
- 空闲列表
  - 长度相同空闲块在一个列表中
  - 列表数目不超过M个

伙伴系统



伙伴系统的回收

伙伴系统：回收考虑合并相邻块



把块 M 释放回可利用空间表

主要内容

- 多维数组
  - 基本概念
  - 数组的空间结构
  - 数组的存储
  - 用数组表示特殊矩阵
  - 稀疏矩阵
- 广义表和存储管理
- Trie结构和Patricia树

Trie结构

- 关键码对象空间分解
  - “trie”这个词来源于“retrieval”
  - 又称 前缀树 或 字典树
    - 由Edward Fredkin发明
- 字符树——26叉Trie
- 主要应用
  - 信息检索（information retrieval）
  - 大量字符串的统计和排序（不仅限于字符串）
  - 常被搜索引擎系统用于文本词频统计
  - 自然语言大规模的英文词典

Trie结构的基本特性

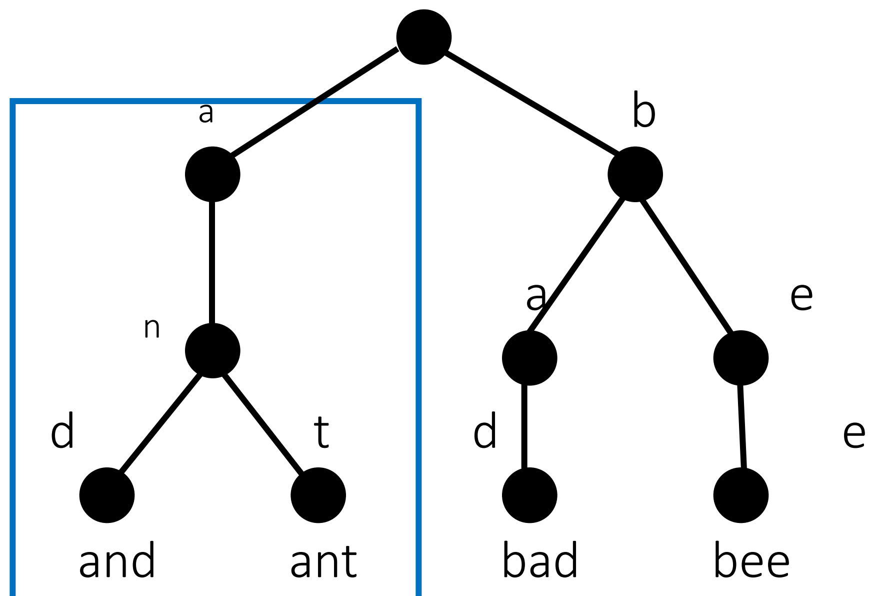
- 根结点不包含字符，除根结点外每个结点都只包含一个字符
  - 根结点对应空字符串
- 从根结点到某一结点，**路径**所经字符连接起来，为**该结点对应的字符串**
  - 每个结点的所有子结点包含的字符都不相同
- 一个结点的所有子孙都有相同的前缀
  - 前缀树
- 基于原则
  - 关键码集合固定
  - 可对结点进行分层标记

英文字符树：26叉Trie

- 一棵子树代表具有相同前缀的关键码的集合

“an”子树代表
具有相同前缀an-
的关键码集合
{and，ant}

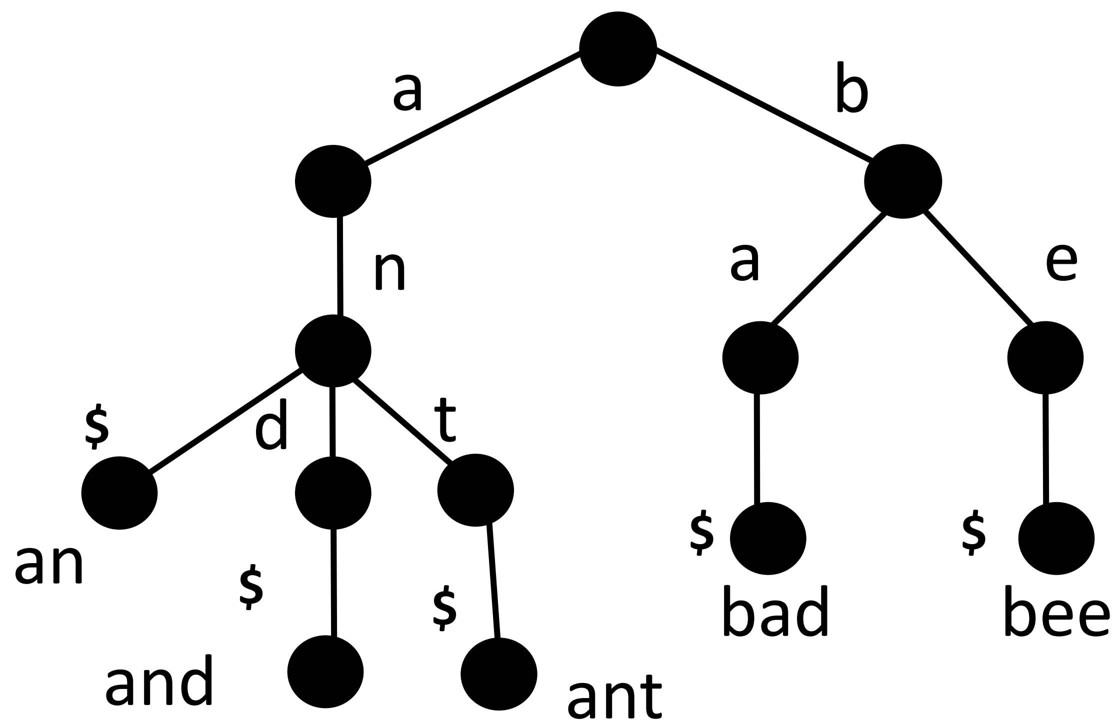
存单词 and, ant, bad, bee



- 常用于存储字典中的单词: 字符树
  - 层次与单词长度相关

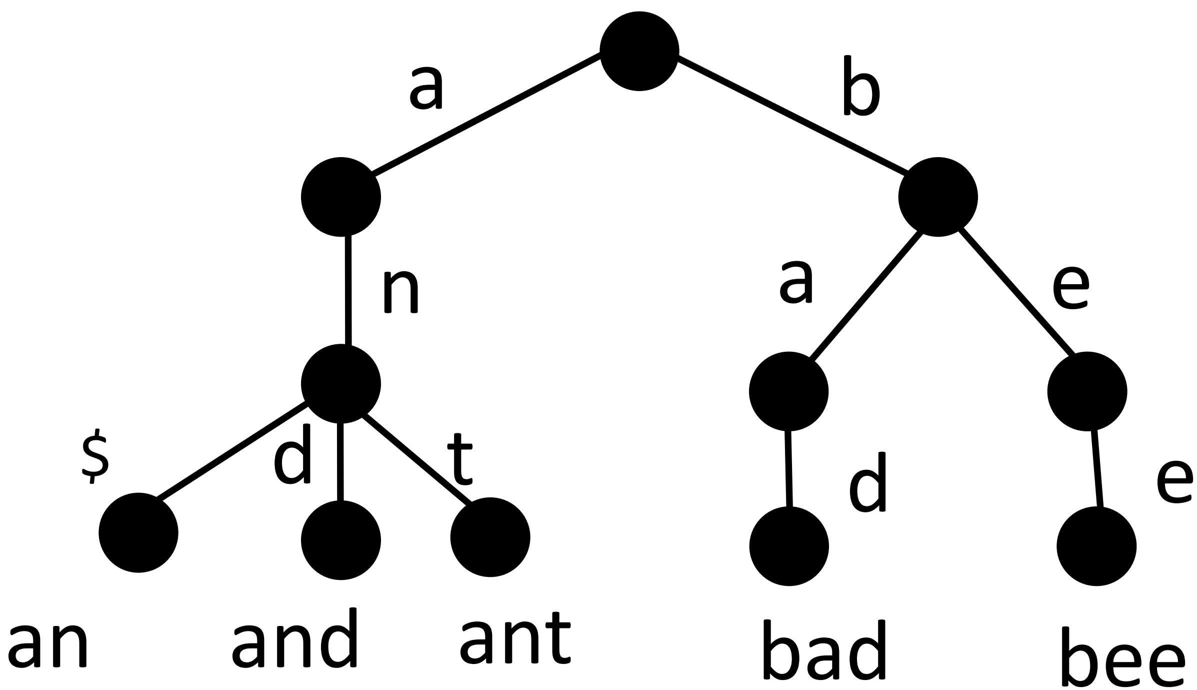
不等长的字符树：“$”标记

- 增加特殊的结束符$
- 叶结点$上存储单词：an，and，ant，bad, bee



压缩靠近叶结点的单路径

- 存储单词 an, and, ant, bad, bee



Trie字符树的特点

- 一个结点的所有子孙都有相同的前缀
- 根结点对应空字符串

- Trie 结构非平衡
  - t 子树下的分支比 z 子树下的多
  - 26个分支因子 —— 庞大的26叉树

二叉Trie结构：PATRICIA 树

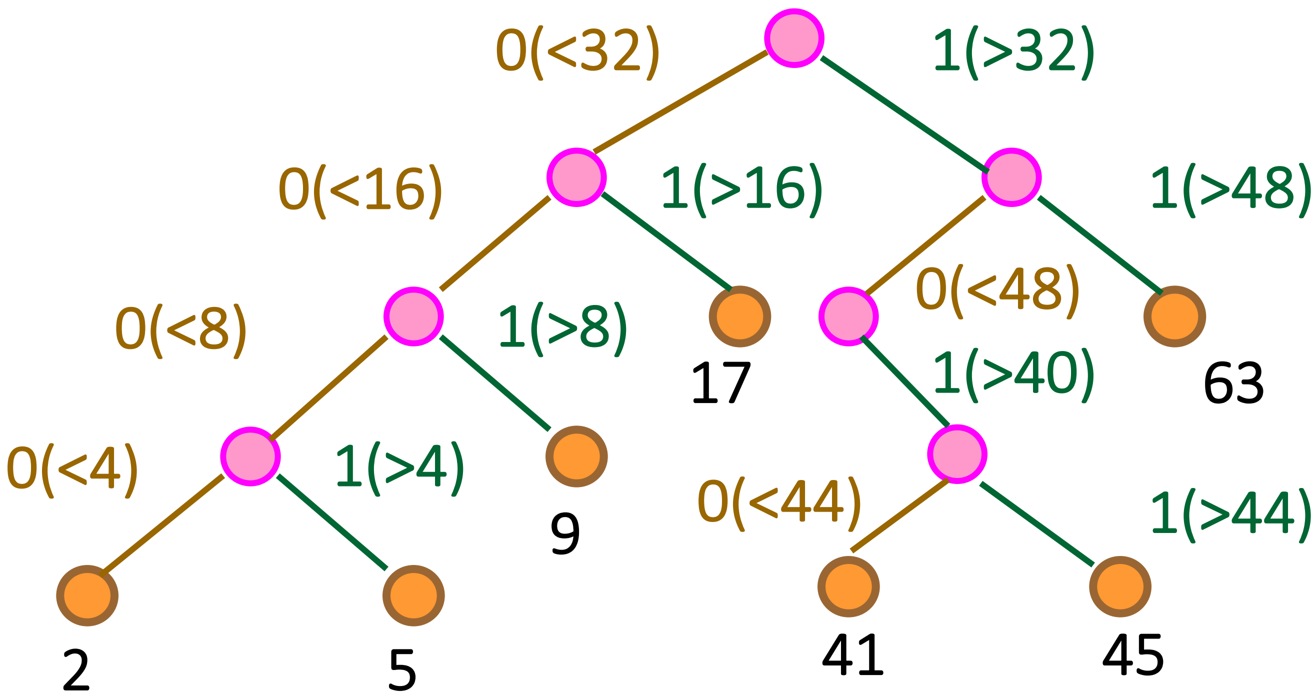
- Practical Algorithm To Retrieve Information Coded In Alphanumeric
- D. Morrision 发明的 Trie 结构变体
  - 根据关键码的二进制位编码来划分（而非根据关键码的大小 范围 划分）
  - 比Trie树更为平衡
  - 二叉Trie树
    - 用每个字符的二进制编码来代表
    - 编码只有0和1
    - Huffman
  - 可适用于诸如中文等基本构成单位较多的情况

PATRICIA 的特点

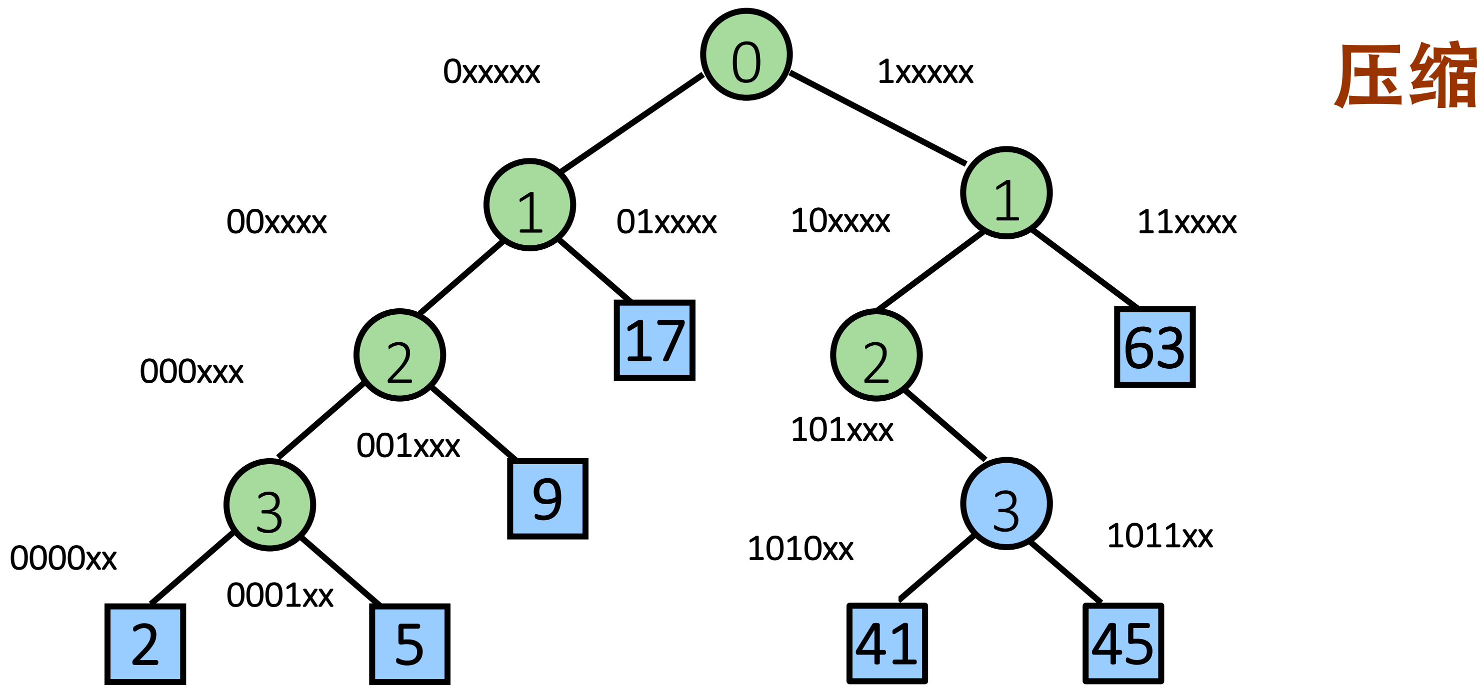
- 改进后的压缩PATRICIA树是满二叉树
  - 每个内部结点都代表一个位的比较
  - 必然产生两个子结点
- 一次检索不超过关键码的位个数

二叉Trie结构

- 元素为 2, 5, 9, 17, 41, 45, 63



二叉Trie结构



编码： 2： 000010 5： 000101 9： 001001 17： 010001
41: 101001 45： 101101 63： 111111