第六章　树

学习要点:

- 理解树的定义和与树相关的结点、度、路径等术语。
- 理解树是一个非线性层次数据结构。
- 掌握树的前序遍历、中序遍历和后序遍历方法。
- 了解树的父结点数组表示法。
- 了解树的儿子链表表示法。
- 了解树的左儿子右兄弟表示法。
- 理解二叉树和ADT二叉树的概念。
- 了解二叉树的顺序存储结构。
- 了解二叉树的结点度表示法。
- 掌握用指针实现二叉树的方法。
- 理解线索二叉树结构及其适用范围。

第六章　树

树是一类重要的非线性数据结构，是以分支关系定义的层次结构

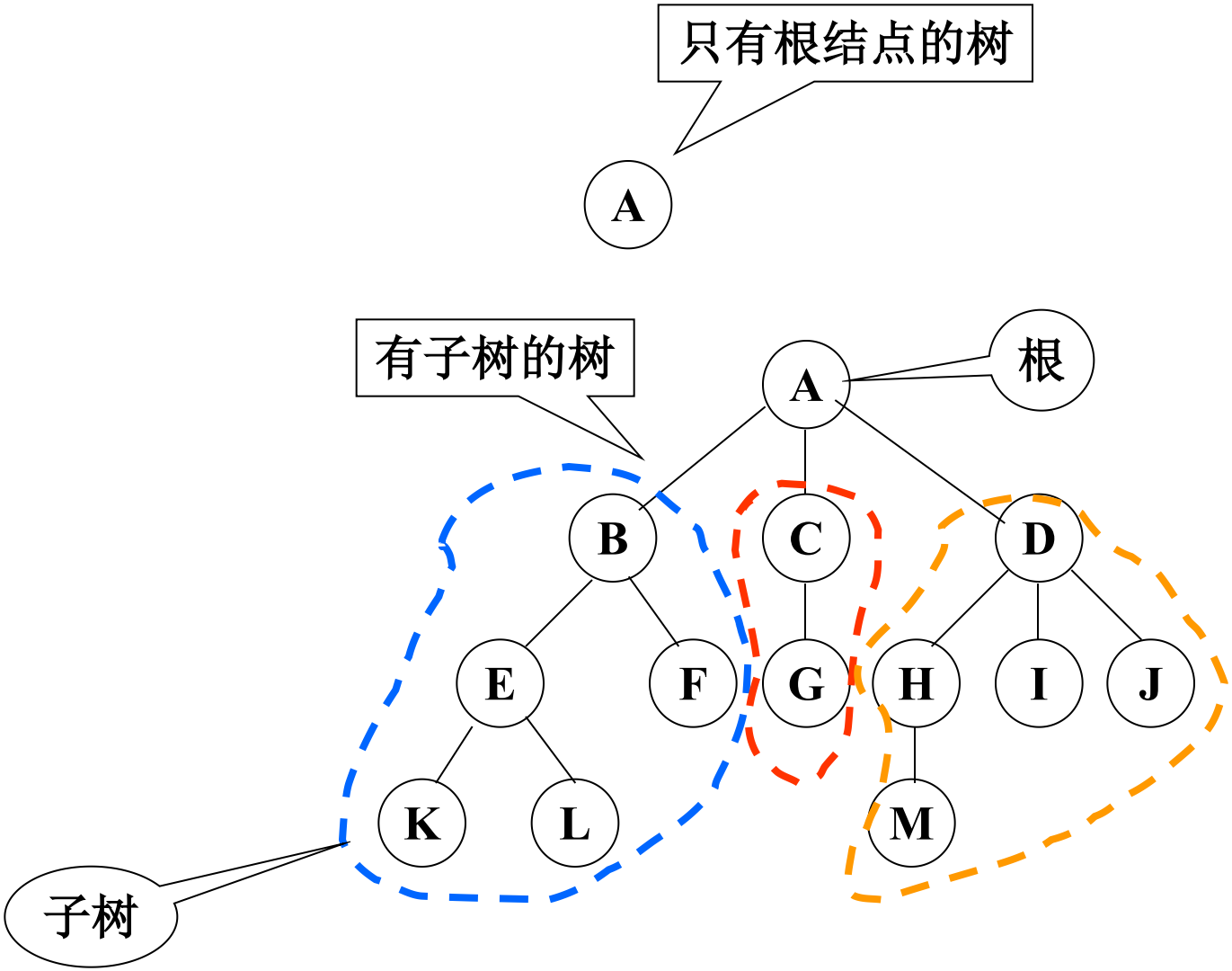
6.1 树的定义

- 定义
  - 递归定义

(1) 单个结点是一棵树，该结点就是树根。

(2) 设$T_1, T_2, \ldots, T_k$都是树，它们的根结点分别为$n_1, n_2 \ldots n_k$，而n是另一个结点且以$n_1, n_2 \ldots n_k$为儿子，则$T_1, T_2, \ldots, T_k$和n构成一棵新树。结点n就是新树的根。称$n_1, n_2 \ldots n_k$为一组兄弟结点。还称$T_1, T_2, \ldots, T_k$为结点n的子树。

为了方便起见，空集合也看作是树，称为空树，并用∧来表示。空树中没有结点。

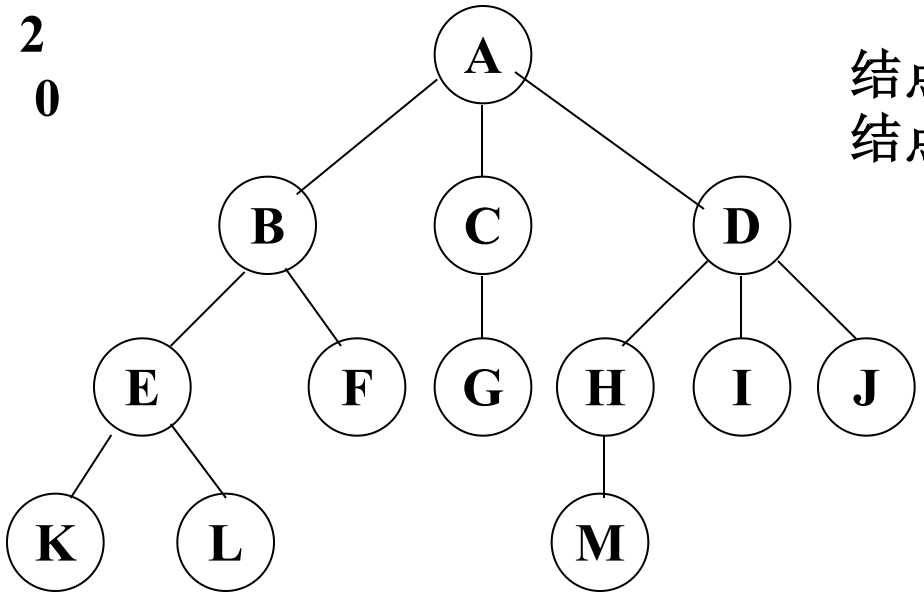

只有根结点的树
A
有子树的树
根
A
B
C
D
E
F
G
H
I
J
K
L
M
子树

- 基本术语
  - 结点——表示树中的元素，包括数据项及若干指向其子树的分支
  - 结点的度——结点的儿子结点个数
  - 树的度——一棵树中最大的结点度数
  - 叶结点——度为0的结点
  - 分枝结点——度不为0的结点
  - 路径——若存在树中的一个节点序列k_1, k_2, …, k_j, 使得结点k_i是k_{i+1}的父结点$(1\le i<j)$，则称该结点序列是树中从结点k_1到结点k_j的一条路径。
  - 路径长度——路径所经过的边的数目。
  - 祖先、子孙
  - 结点的高度——从该结点到各叶结点的最长路径长度
  - 树的高度——根结点的高度
  - 结点的深度(或层数)——从树根到任一结点n有唯一的路径，称该路径的长度为结点n的深度(或层数)。从根结点算起，根为第0层，它的孩子为第1层……
  - 有序树——为树的每一组兄弟结点定义一个从左到右的次序
  - 左儿子、右兄弟
  - 森林——m(m≥0)棵互不相交的树的集合

叶结点：K，L，F，G，M，I，J
分枝节点：A，B，C，D，E，H

结点A的度：3
结点B的度：2
结点M的度：0

树的度：3



结点I的双亲：D
结点L的双亲：E

结点B，C，D为兄弟
结点K，L为兄弟

结点A的孩子：B，C，D
结点B的孩子：E，F

结点A的高度：3
结点D的高度：2
树的高度：3

结点A的层次：0
结点M的层次：3

结点A是结点F，G的祖先
结点B，C，…是结点A的子孙

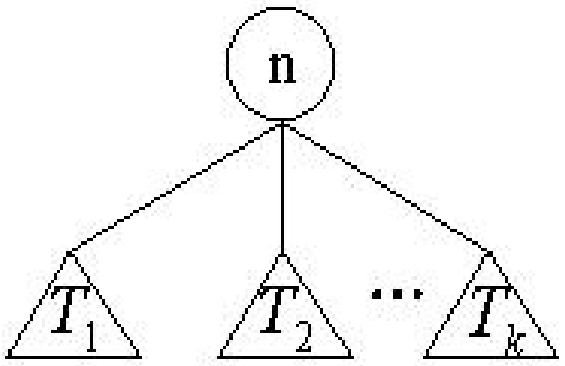
6.2 树的遍历

- 树的遍历
  - 遍历——按一定规律走遍树的各个顶点，且使每一顶点仅被访问一次，即找一个完整而有规律的走法，以得到树中所有结点的一个线性排列。
- 树T的3种遍历方式的递归定义：（T如图所示）

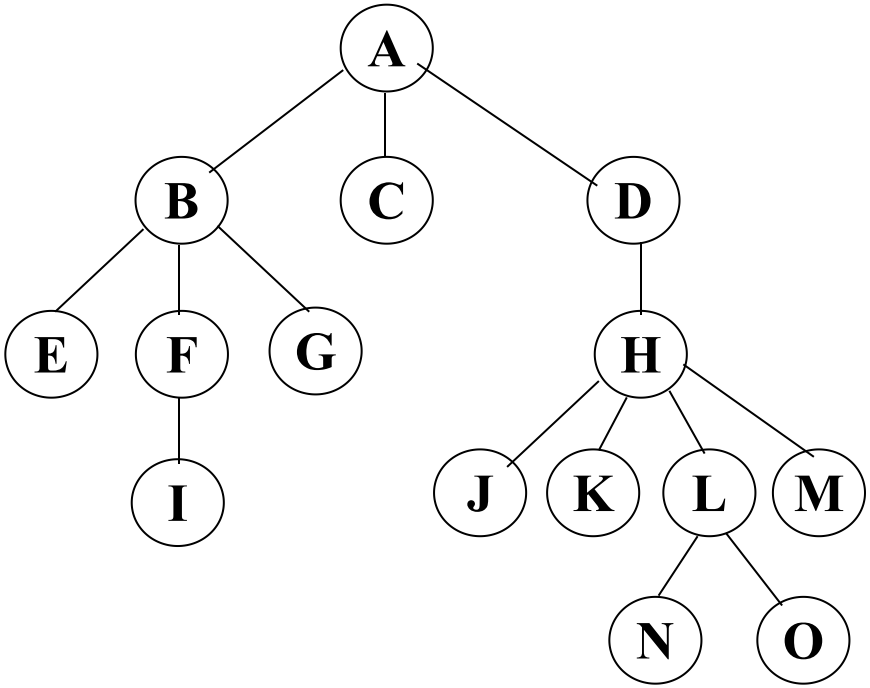
(1) 前序遍历——先访问树根n，然后依次前序遍历$T_1, T_2, \ldots, T_k$。

(2) 中序遍历——先中序遍历T_1，然后访问树根n，接着依次对$T_2, T_3, \ldots, T_k$进行中序遍历。

(3) 后序遍历——先依次对$T_1, T_2, \ldots, T_k$进行后序遍历，最后访问树根n。



树T



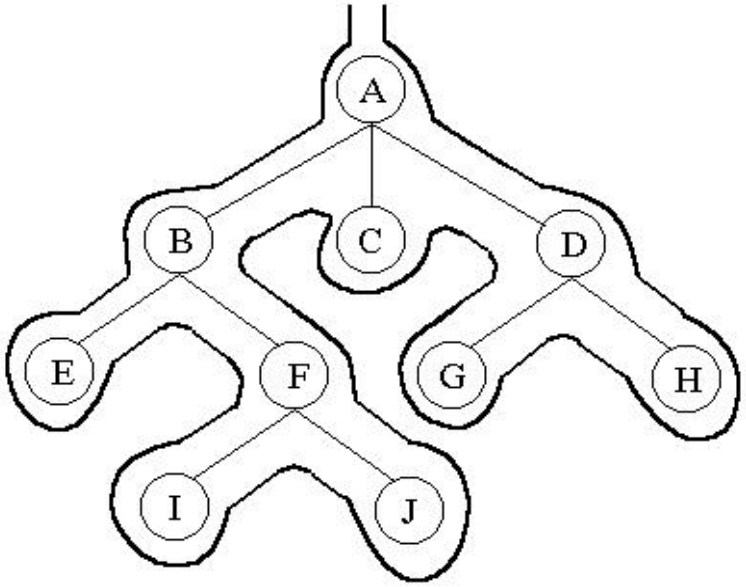
前序遍历：A B E F I G C D H J K L N O M

中序遍历：E B I F G A C J H K N L O M D

后序遍历：E I F G B C J K N O L M H D A

层次遍历：A B C D E F G H I J K L M N O

有序树T的3种遍历的非递归方式



- 按第一次经过的时间次序将各个结点列表:
  A B E F I J C D G H —— 前序列表
- 按最后一次经过的时间次序将各个结点列表:
  E I J F B C G H D A —— 后序列表
- 叶结点在第一次经过时列出，而内部结点在第2次经过时列出:
  E B I F J A C G D H —— 中序列表

- 有序树T的3种遍历能得到什么信息？

(1)在3种不同方式中，各树叶之间的相对次序是相同的，它们都按树叶之间从左到右的次序排列，其差别仅在于内部结点之间以及内部结点与树叶之间的次序有所不同。

(2)后序遍历有助于查询结点间的祖先和子孙关系，因为

postorder(y)-desc(y)≤postorder(x)≤postorder(y)。

其中y是T中的任一结点；x是y的子孙；desc(y)是T中y的真子孙数；postorder(y)是T中y的后序序号。

(3)前序遍历也有助于查询结点间的祖先和子孙关系,因为

preorder(y)≤preorder(x)≤preorder(y)+desc(y) 。

其中y是T中的任一结点；x是y的子孙；desc(y)是T中y的真子孙数；preorder(y)是T中y的前序序号。

6.3 树的表示法

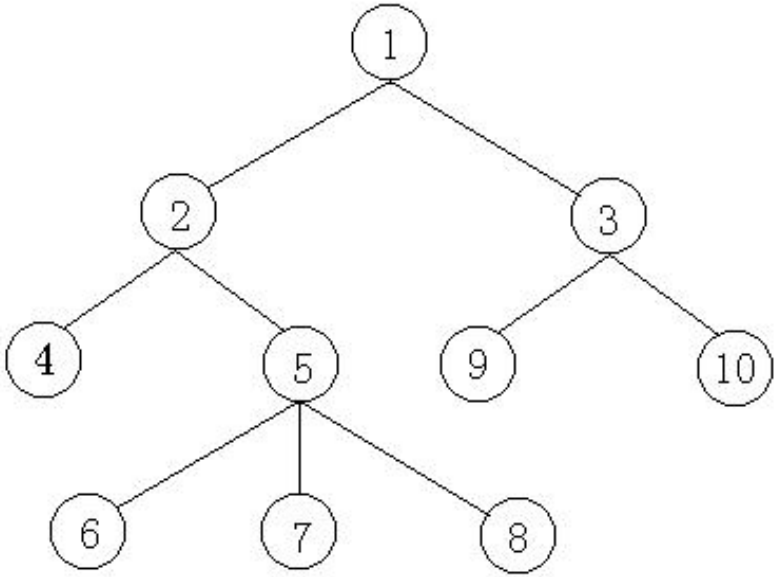
- 树的存储结构
  - 父结点数组表示法

(1)树中的结点数字化为它们的编号1,2,…,n。

(2)用一个一维数组存储每个结点的父结点。即：father[k]中是存放结点k的父结点的编号。

(3)由于树中每个结点的父结点是唯一的，所以父结点数组表示法可以唯一表示任何一棵树。

(4)实例：见下页



(a)

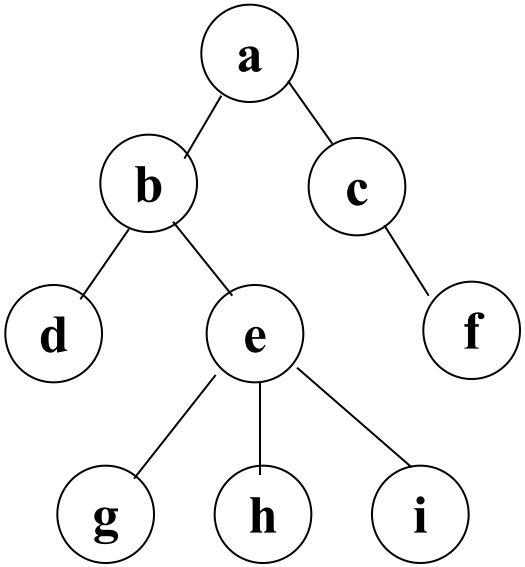
0	1	1	2	2	5	5	5	3	3
1	2	3	4	5	6	7	8	9	10

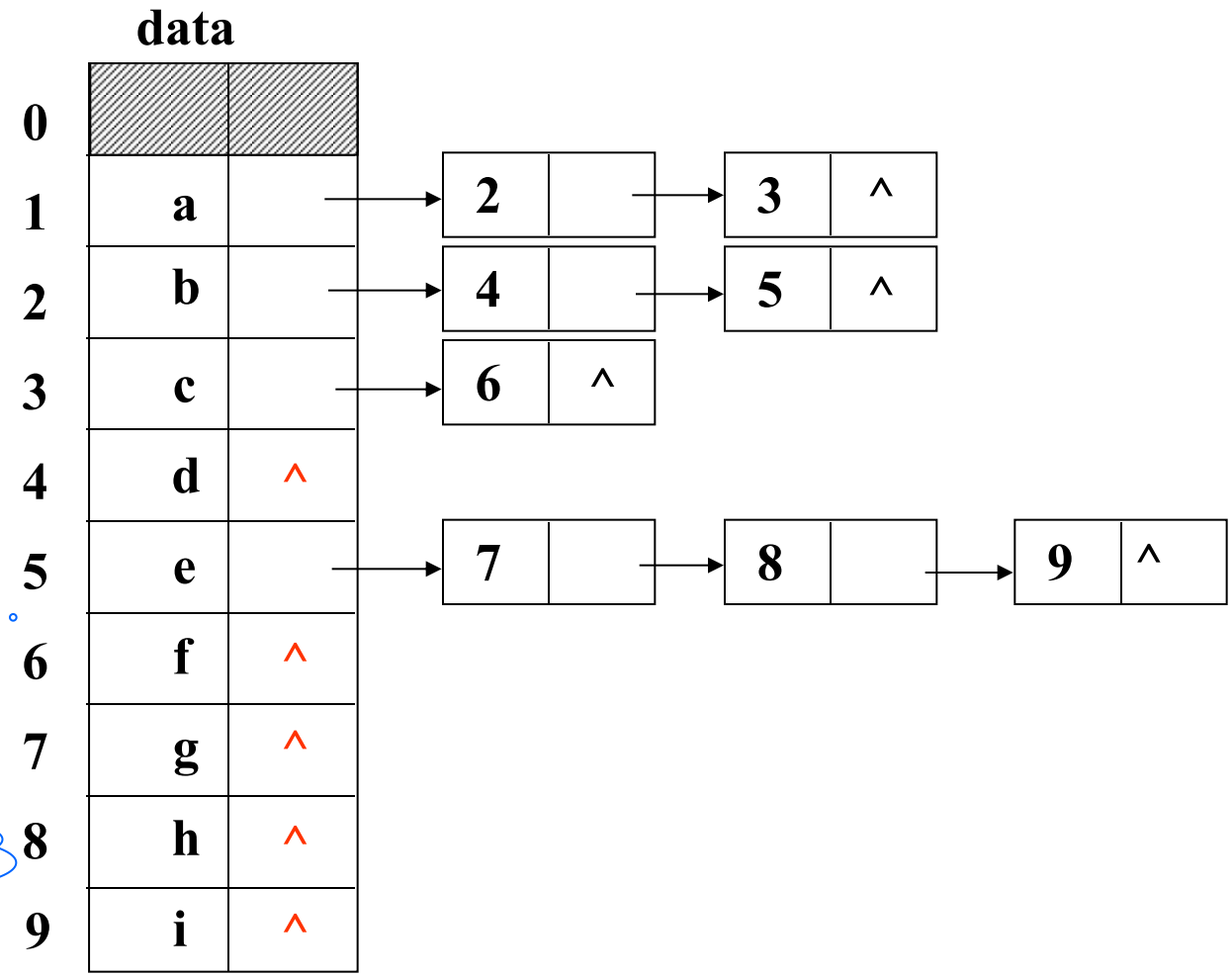
(b)

如何找孩子结点

- 效率分析
  - 寻找一个结点的父结点只需要0(1)时间。
  - 于涉及查询儿子结点和兄弟结点信息的运算，可能要遍历整个数组。

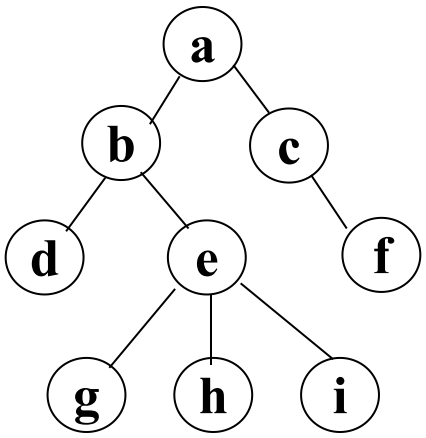
孩子表示法





如何找双亲结点

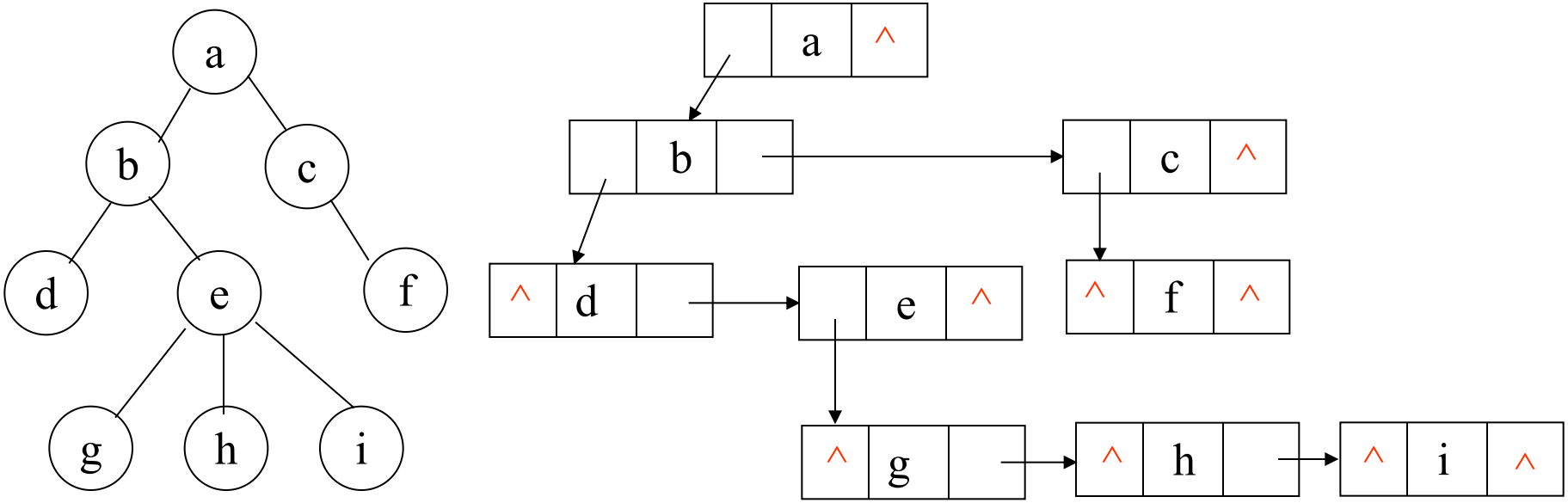
带双亲的孩子链表



	data	parent	
1	a	0	
2	b	1	
3	c	1	
4	d	2	^
5	e	2	
6	f	3	^
7	g	5	^
8	h	5	^
9	i	5	^

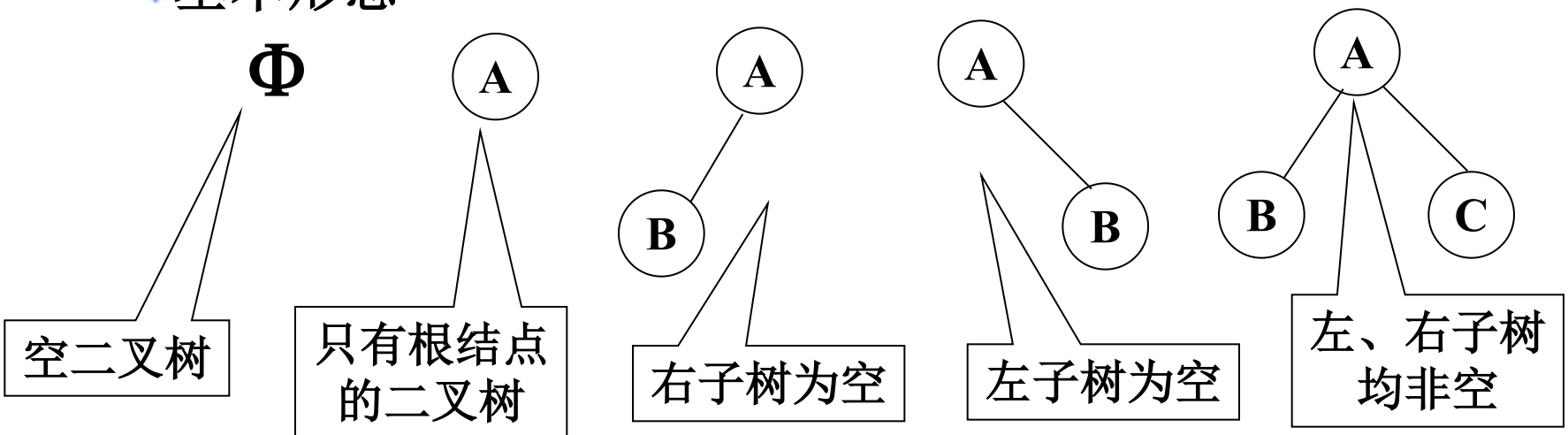
左儿子右兄弟表示法

- 实现：用二叉链表作树的存储结构，链表中每个结点的两个指针域分别指向其最左儿子和右邻兄弟



6.4 二叉树

- 定义：二叉树是n(n≥0)个结点的有限集，它或为空树(n=0)，或由一个根结点和两棵分别称为左子树和右子树的互不相交的二叉树构成。
- 特点
  - 每个结点至多有二棵子树(即不存在度大于2的结点)
  - 二叉树的子树有左、右之分，且其次序不能任意颠倒
- 基本形态



- 二叉树性质
  - 高度为h≥0的二叉树至少有h+1个结点。
  - 高度为h≥0的二叉树至多有$2^{h+1}-1$个结点。
  - 约定空二叉树的高度为-1。
  - 含有n≥1个结点的二叉树的高度至多为n-1。
  - 含有**n≥1**个结点的二叉树的高度至少为$\lfloor \log n \rfloor$，因此为$\Omega(\lfloor \log n \rfloor)$。

- 具有n个结点的不同形态的二叉树的数目即所谓的n阶Catalan数：

$$B_n = \frac{1}{n+1}\binom{2n}{n}$$

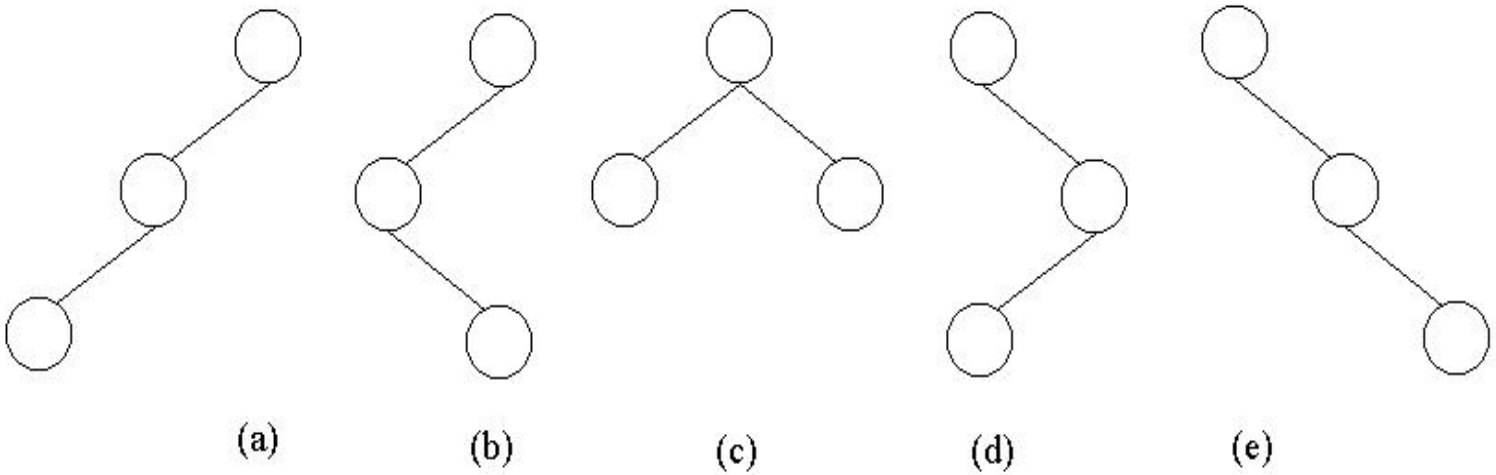
因为一棵具有n个结点的二叉树可以看成是由一个根结点、一棵具有i个结点的左子树和一棵具有n-i-1个结点的右子树所组成，i=0, 1, 2, 3, …, n−1，所以 B_n 可以递归地表达为

$$B_n = \begin{cases} 1 & n = 0 \\ \sum_{i=0}^{n-1} B_i B_{n-i-1} & n \geq 1 \end{cases}$$

解此递归式即是上述的显式表达式。

例如：具有3个结点的不同形态的二叉数的数目$B_3=5$

$$B_n = \frac{1}{n+1}\binom{2n}{n}$$



- 重要性质：对任何一棵二叉树T，如果其终端结点数为n0，度为2的结点数为n2，

则：n0=n2+1

证明：n1为二叉树T中度为1的结点数

因为：二叉树中所有结点的度均小于或等于2

所以：其结点总数n=n0+n1+n2

又二叉树中，除根结点外，其余结点都只有一个分支进入

设B为分支总数，则n=B+1

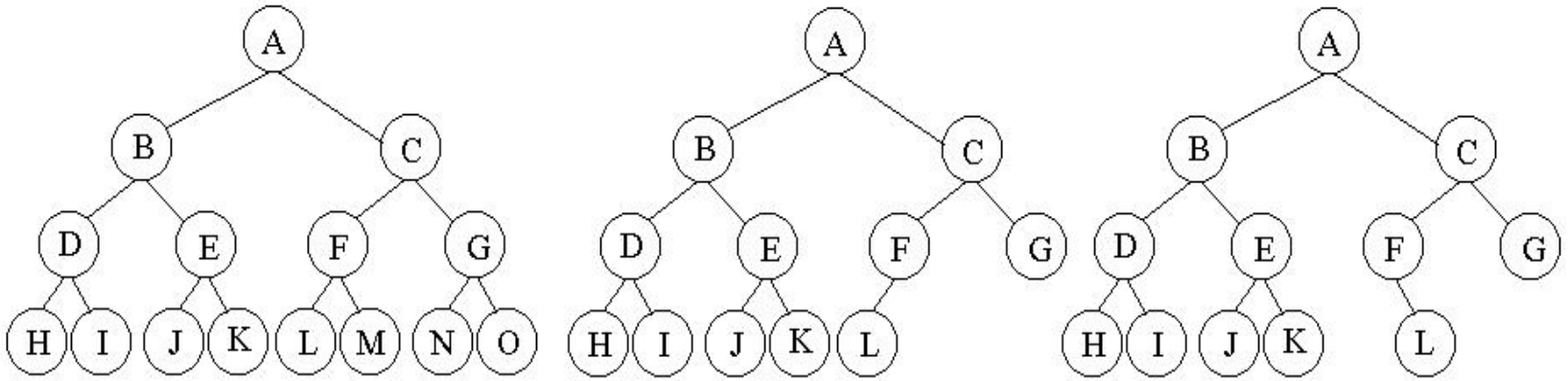
又：分支由度为1和度为2的结点射出

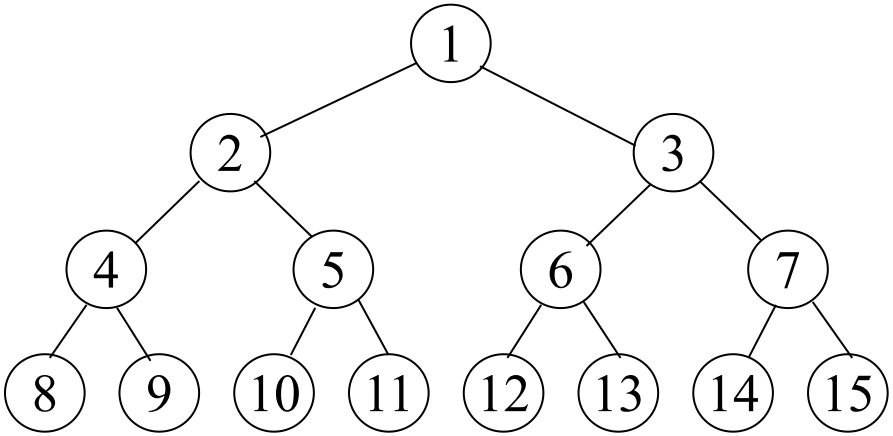
∴B=n1+2n2

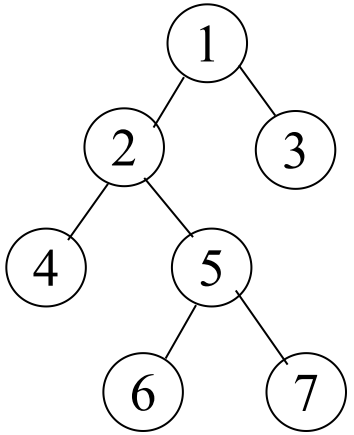
于是，n=B+1=n1+2n2+1=n0+n1+n2

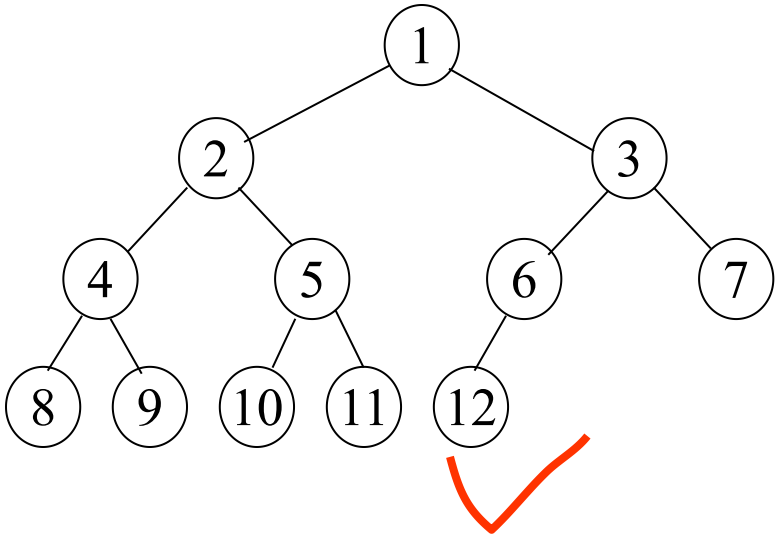
∴n0=n2+1

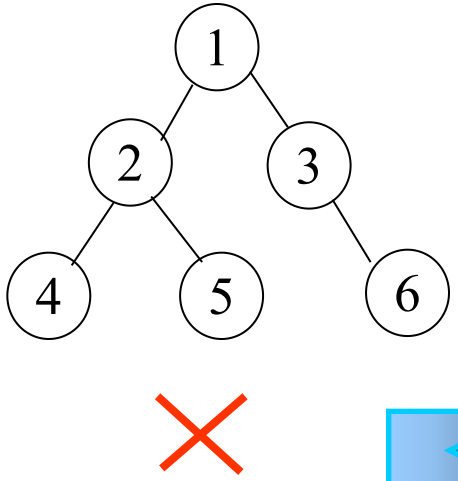
- 几种特殊形式的二叉树
  - 满二叉树
    - 定义：一棵高度为 k 且有 $2^{k+1}-1$ 个结点的二叉树称为 ~
    - 特点：每一层上的结点数都是最大结点数
  - 近似满二叉树（完全二叉树）
    - 定义：若一棵二叉树最多只有最下面的2层上结点的度数可以小于2，并且最下面一层上的节点都集中在该层的最左边，则这种二叉树称为近似满二叉树




1
2
3
4
5
6
7
8
9
10
11
12
13
14
15


1
2
3
4
5
6
7


1
2
3
4
5
6
7
8
9
10
11
12


1
2
3
4
5
6

- 近似满二叉树性质：

如果对一棵有n个结点的完全二叉树的结点按层序编号，则对任一结点i(1≤i≤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⌊i/2⌋

(2) 如果2i>n，则结点i无左孩子；如果2i≤n，则其左孩子是2i

(3) 如果2i+1>n，则结点i无右孩子；如果2i+1≤n，则其右孩子是2i+1

6.5 ADT二叉树

6.5.1 ADT二叉树用来存放有二叉树结构的数据集

ADT二叉树支持的主要基本运算：

(1)BinaryInit() 创建一棵空二叉树。

(2)BinaryEmpty() 判断给定的二叉树是否为空。

(3)Root(T) 返回给定二叉树的根结点标号。

(4)MakeTree(x,T,L,R)以x为根结点元素,以L和R分别为左、右子树,构建一棵新的二叉树T。

(5)BreakTree(T,L,R)函数MakeTree的逆运算，即将二叉树拆分为根结点元素，左子树L和右子树R等3部分。

6.5 ADT二叉树

6.5.1 ADT二叉树用来存放有二叉树结构的数据集

ADT二叉树支持的主要基本运算(续)

(6)PreOrder(visit,t) 前序遍历给定的二叉树。

(7)InOrder(visit,t) 中序遍历给定的二叉树。

(8)PostOrder(visit,t) 后序遍历给定的二叉树。

(9)PreOut (T) 给定的二叉树的前序列表。

(10)InOut (T) 给定的二叉树的中序列表。

(11)PostOut (T) 给定的二叉树的后序列表。

(12)Delete(t) 删除给定的二叉树。

(13)Height(t) 求给定的二叉树的高度。

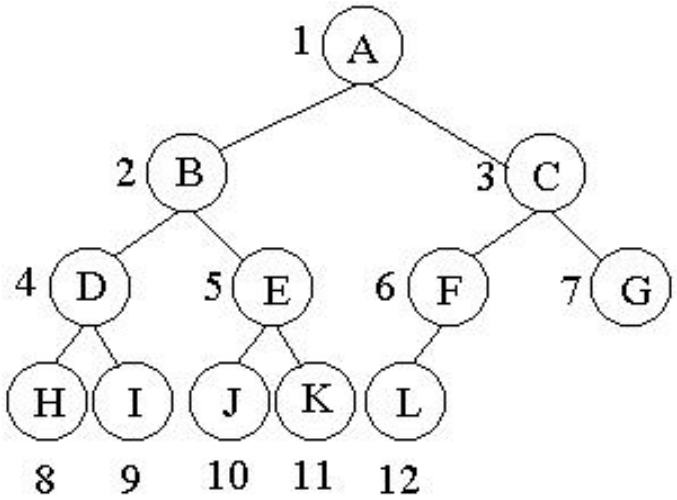
(14)Size(t) 求给定的二叉树的结点数。

6.5 ADT二叉树

6.5.2 ADT二叉树的实现

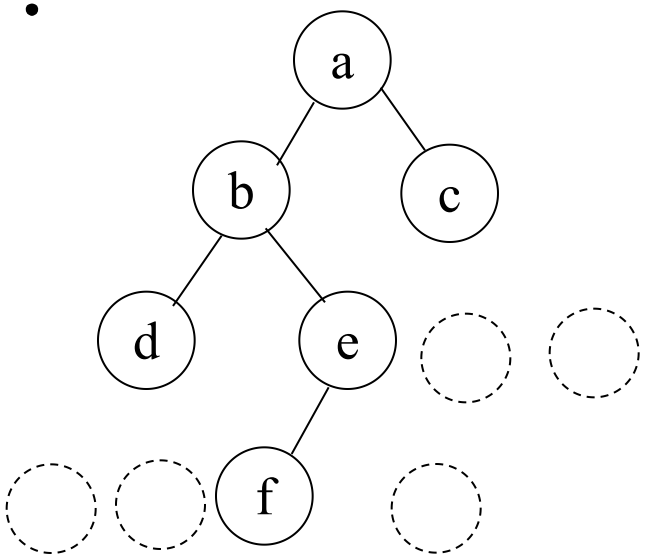
6.5.2.1 用顺序存储结构实现(一种无边表示)

- 适用的对象：近似满二叉树
- 基本想法：若将所有的结点按层自上而下每层自左至右，从1开始编号。



1	2	3	4	5	6	7	8	9	10	11	12
A	B	C	D	E	F	G	H	I	J	K	L

- 不适用的例子：



1	2	3	4	5	6	7	8	9	10	11
a	b	c	d	e	0	0	0	0	f	0

6.5 ADT二叉树

6.5.2 ADT二叉树的实现

6.5.2.2 二叉树的结点度表示法(另一种无边表示)

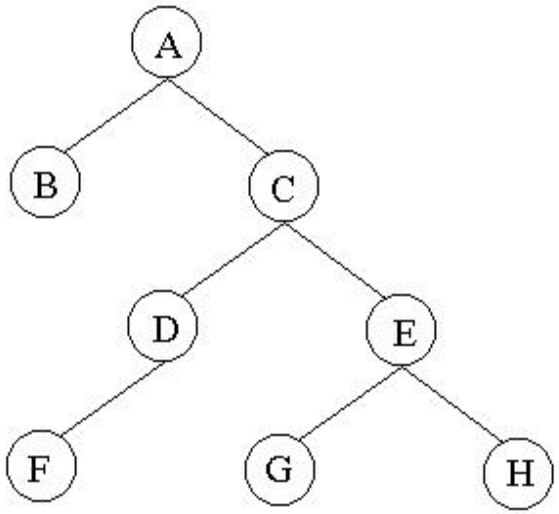
- 基本想法：若将所有的结点按后序列表从1开始编号，并在每个结点中附加一个0到3之间的整数，以表示该结点的分叉特征。该整数为0时，表示相应的结点无儿子；为1时，表示相应结点只有左儿子；为2时，表示相应结点只有右儿子；为3时，表示相应结点既有左儿子又有右儿子。那么，结点之间有如下隐含关系：若结点i有右儿子，则i-1一定是其右儿子结点。另一方面，结点i的左儿子结点必在i之前，而其父结点必在i之后。照此，二叉树可以用记录数组来表示

6.5 ADT二叉树

6.5.2 ADT二叉树的实现

6.5.2.2 二叉树的结点度表示法(另一种无边表示)

- 例



(a)

(B, 0)	(F, 0)	(D, 1)	(G, 0)	(H, 0)	(E, 3)	(C, 3)	(A, 3)

(b)

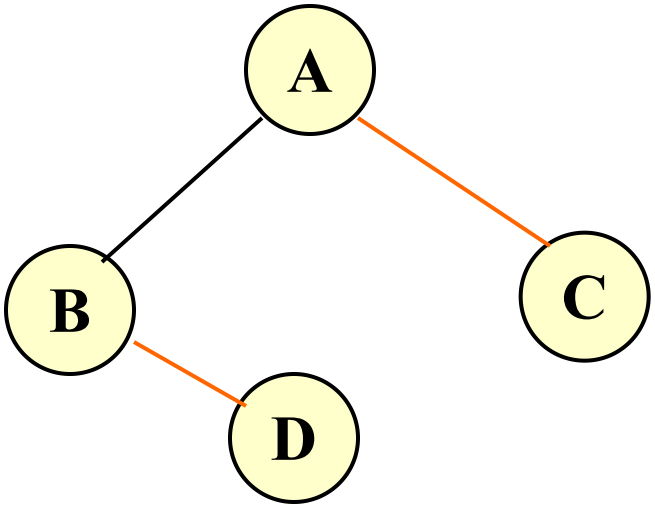
6.5 ADT二叉树

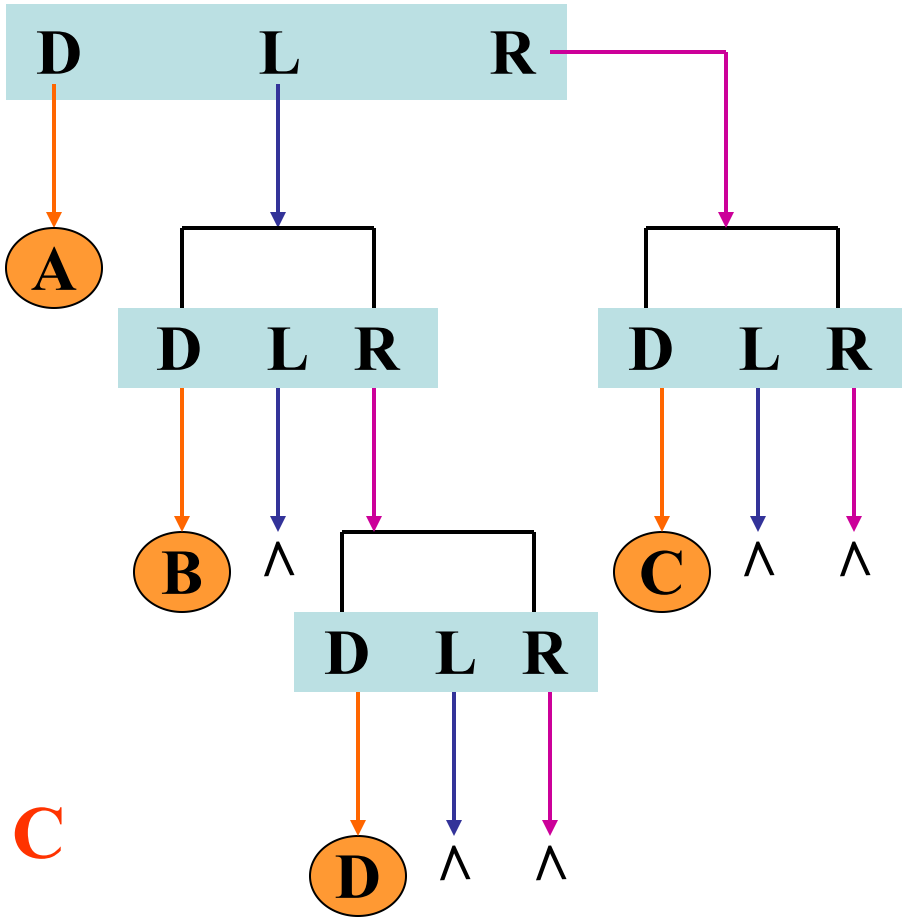
6.5.2 ADT二叉树的实现

6.5.2.3 二叉树的指针实现—带两个指针的结点的实现

P116-120

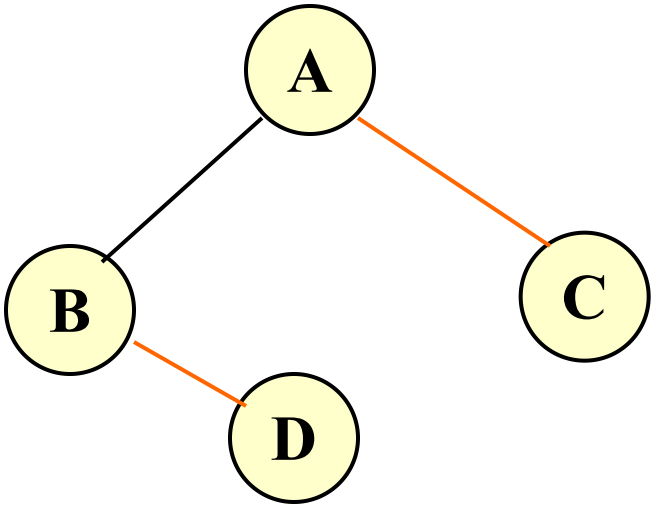
前序遍历：

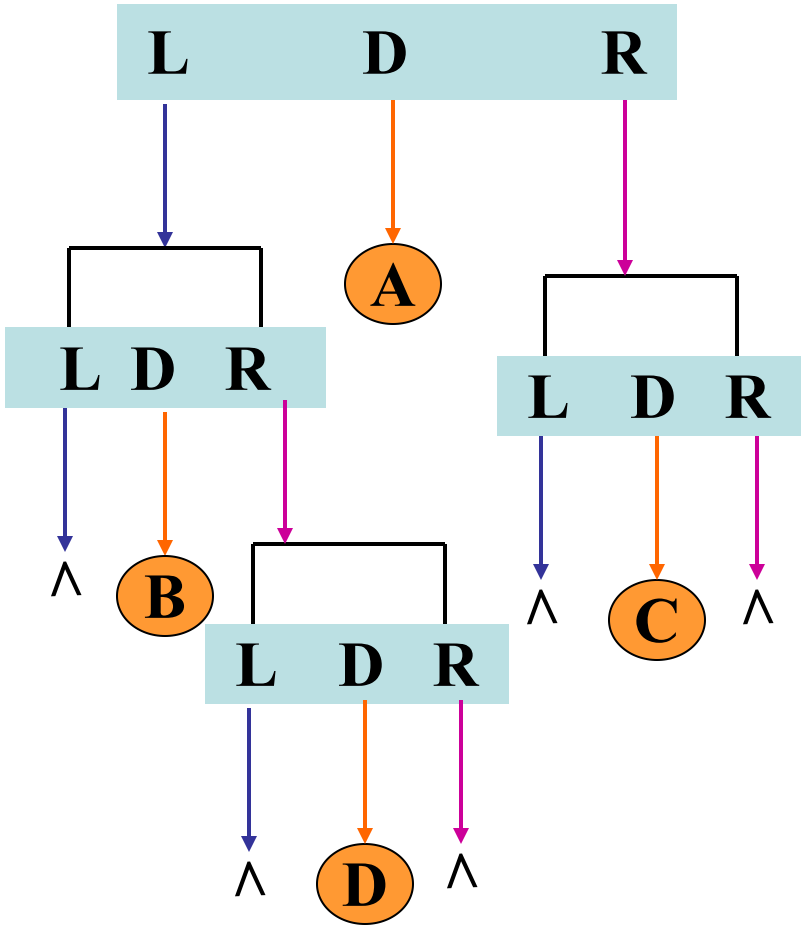




先序遍历序列：A B D C

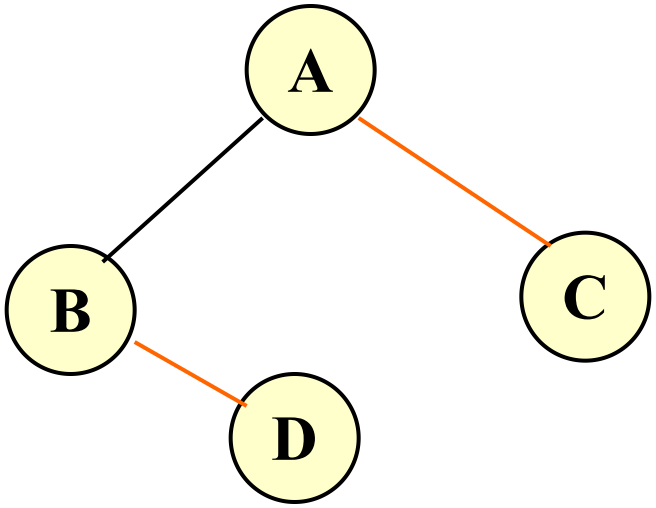
中序遍历：

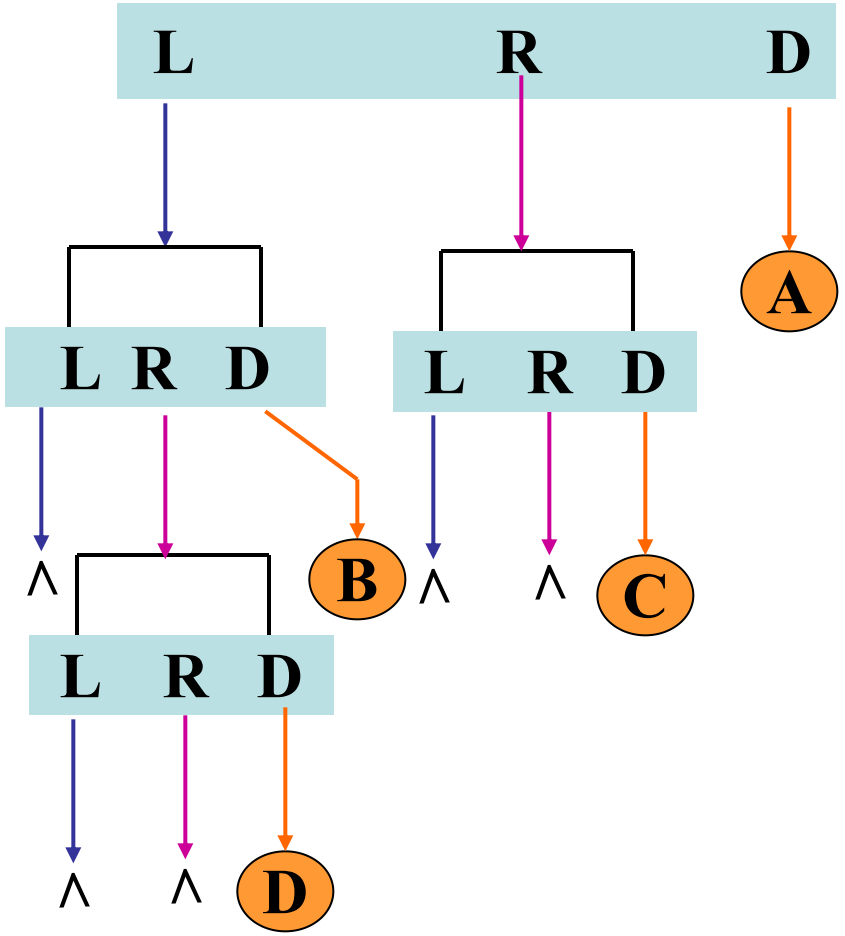




中序遍历序列：B D A C

后序遍历：





后序遍历序列： D B C A

6.5.1 二叉树的应用

- 哈夫曼树(Huffman)——带权路径长度最短的树
  - 定义
    - 路径：从树中一个结点到另一个结点之间的分支构成这两个结点间的路径
    - 路径长度：路径上的分支数
    - 树的路径长度：从树根到每一个结点的路径长度之和
    - 树的带权路径长度：树中所有带权结点的路径长度之和

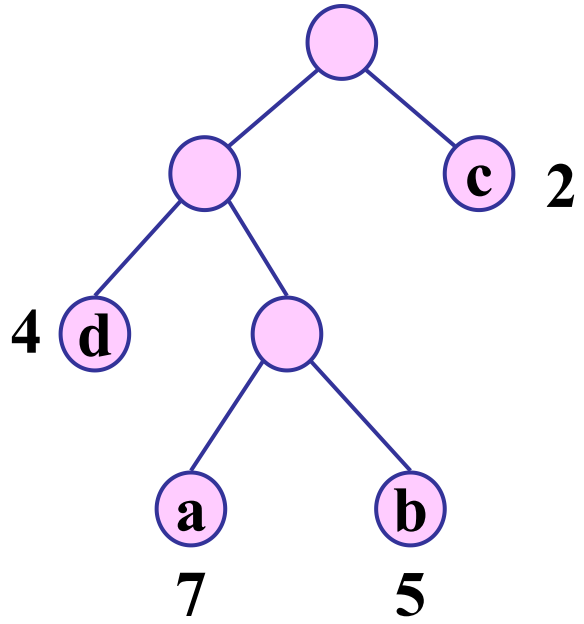
      记作： $wpl = \sum_{k=1}^{n} w_k l_k$

      其中： w_k — 权值

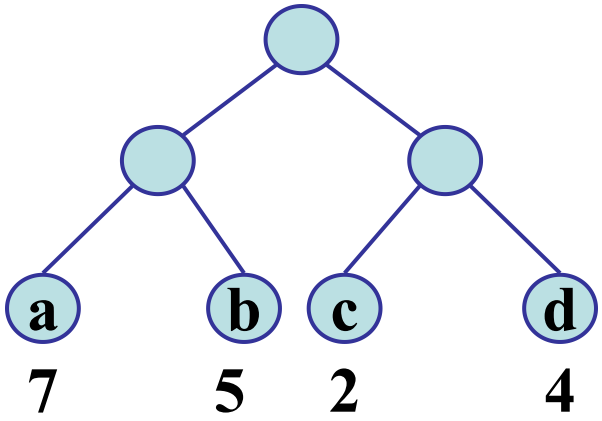
      l_k — 结点到根的路径长度
    - **Huffman树——设有n个权值{w1,w2,……wn}，构造一棵有n个叶子结点的二叉树，每个叶子的权值为wi,则wpl最小的二叉树叫Huffman树**

例 有4个结点，权值分别为7，5，2，4，构造有4个叶子结点的二叉树

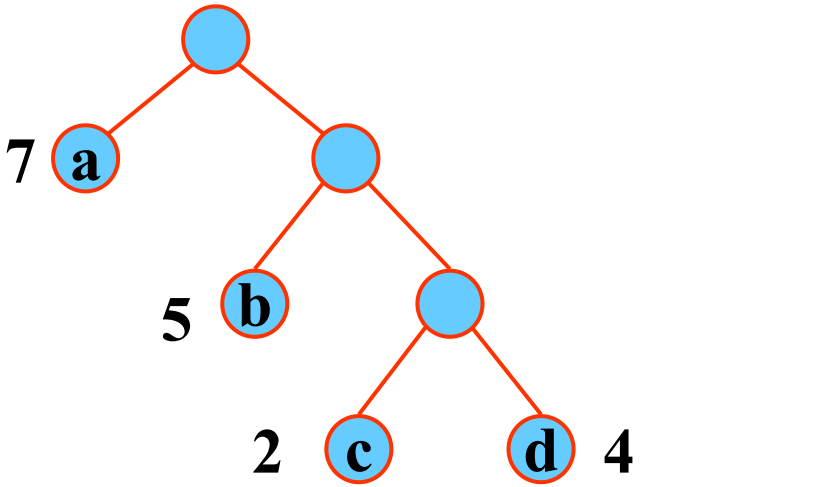
$$WPL=\sum_{k=1}^{n}W_K L_K$$



WPL=7*3+5*3+2*1+4*2=46



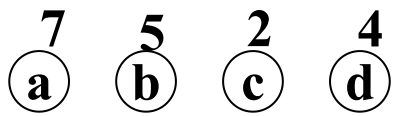
WPL=7*2+5*2+2*2+4*2=36

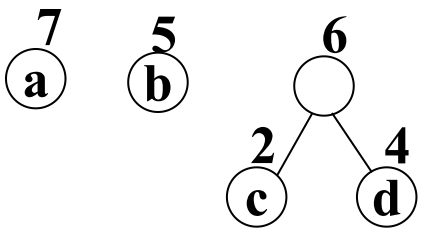


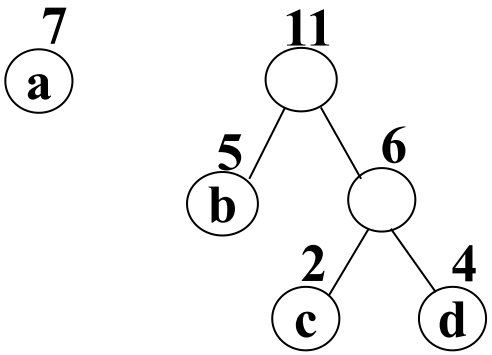
WPL=7*1+5*2+2*3+4*3=35

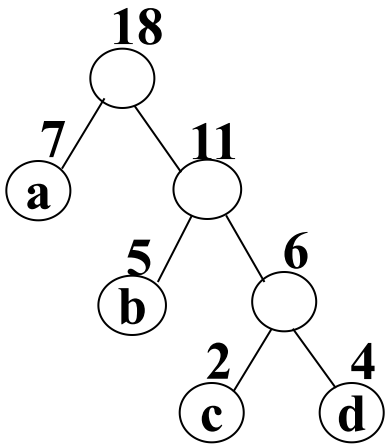
- 构造Huffman树的方法——Huffman算法
  - 构造Huffman树步骤
    - 根据给定的n个权值{w1,w2,……wn}，构造n棵只有根结点的二叉树，令起权值为wj
    - 在森林中选取两棵根结点权值最小的树作左右子树，构造一棵新的二叉树，置新二叉树根结点权值为其左右子树根结点权值之和
    - 在森林中删除这两棵树，同时将新得到的二叉树加入森林中
    - 重复上述两步，直到只含一棵树为止，这棵树即哈夫曼树

例

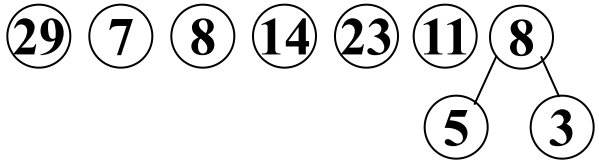


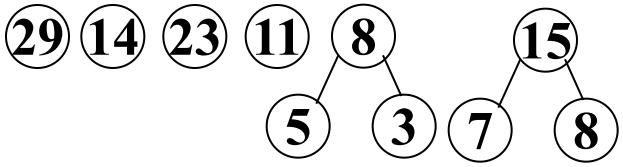


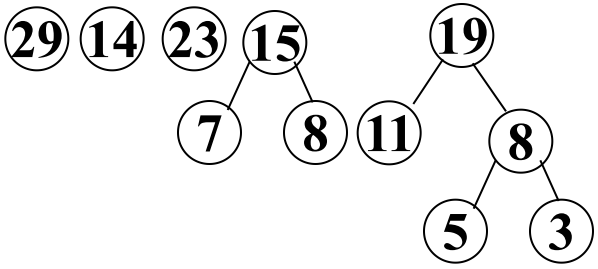


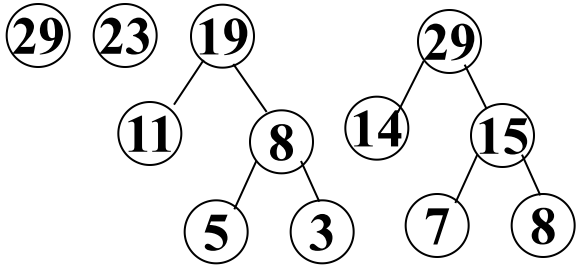


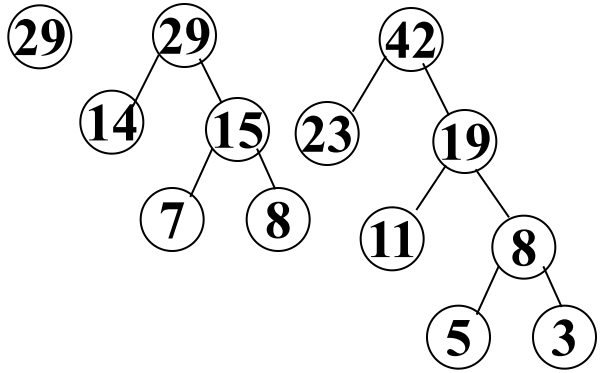
例 w={5, 29, 7, 8, 14, 23, 3, 11}

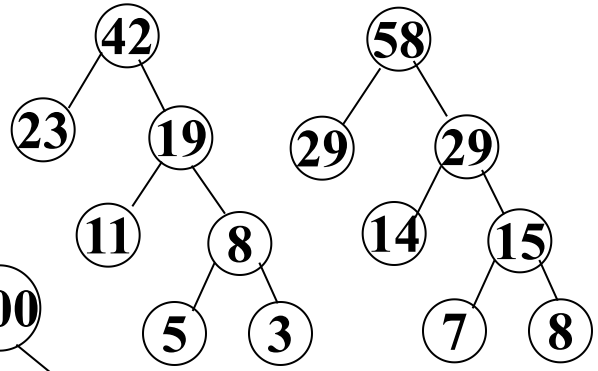


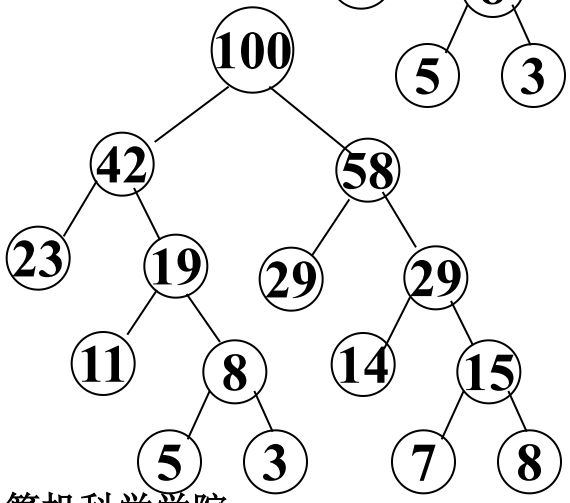












- Huffman算法实现

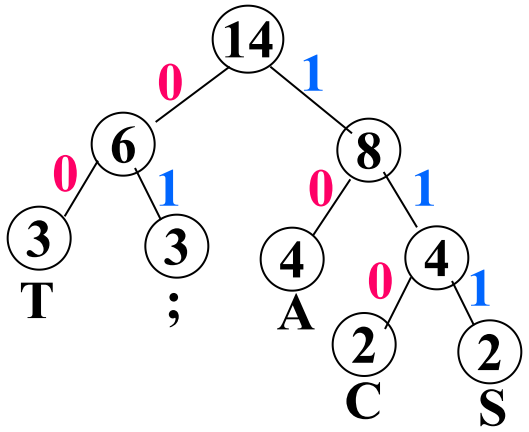

Ch5_8.txt

  - 一棵有n个叶子结点的Huffman树有2n-1个结点
  - 采用顺序存储结构——一维结构数组
  - 结点类型定义

```
typedef  struct
{  int data;
   int pa,lc,rc;
}JD;
```

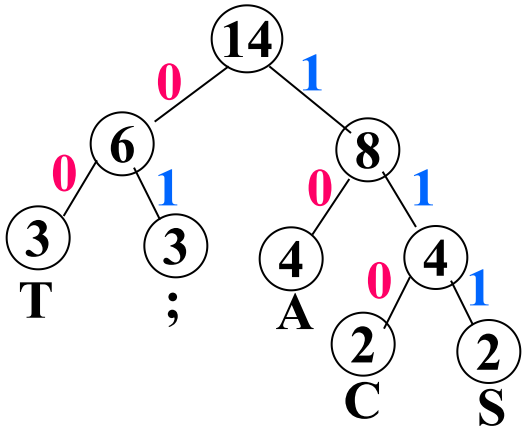
- **Huffman编码：数据通信用的二进制编码**
  - 思想：根据字符出现频率编码，使电文总长最短
  - 编码：根据字符出现频率构造Huffman树，然后将树中结点引向其左孩子的分支标“0”，引向其右孩子的分支标“1”；每个字符的编码即为从根到每个叶子的路径上得到的0、1序列

例 要传输的字符集 D={C,A,S,T, ; }
字符出现频率 w={2,4,2,3,3}



T： 00
; ： 01
A： 10
C： 110
S： 111

- 译码：从Huffman树根开始，从待译码电文中逐位取码。若编码是“0”，则向左走；若编码是“1”，则向右走，一旦到达叶子结点，则译出一个字符；再重新从根出发，直到电文结束



T： 00
; ： 01
A： 10
C： 110
S： 111

例 电文是{CAS;CAT;SAT;AT}
其编码 “1101011101110100001111100011000”
电文为“1101000”
译文只能是“CAT”

6.6 线索二叉树

6.6.1 引入线索二叉树的动因

用指针实现二叉树时，每个结点只有指向其左、右儿子结点的指针，所以从任一结点出发直接只能找到该结点的左、右儿子。在一般情况下无法直接找到该结点在某种遍历序下的前驱和后继结点。若在每个结点中增加指向其前驱和后继结点的指针，虽可提高遍历的效率却降低了存储效率。注意到用指针实现二叉树时，n个结点二叉树中有n+1个空指针。若利用这些空指针存放指向结点在某种遍历次序下的前驱或后继的指针，那么，可以期望提高遍历的效率。

6.6 线索二叉树

6.6.2 有关概念和术语

- “线索”：所引入的非空指针称为“线索”。
- 线索二叉树：加上了线索的二叉树称为线索二叉树。
- 线索标志位：为了区分一个结点的指针是指向其儿子结点的指针，还是指向其前驱或后继结点的线索,在每个结点中增加的2个位—LeftThread、RightThread分别称为左、右线索标志位。
- 线索化：对一棵非线索二叉树以某种次序遍历使其变为一棵线索二叉树的过程称为二叉树的线索化。

6.6 线索二叉树

6.6.3 线索二叉树结点类型定义

P120

6.6 线索二叉树

6.6.4 二叉树线索化：

由于线索化的实质是将二叉树中的空指针改为指向其前驱结点或后继结点的线索(并做上线索标志)，而一个结点的前驱或后继结点只有遍历才能知道，因此线索化的过程是在对二叉树遍历的过程中修改空指针的过程。

为了记下遍历过程中访问结点的先后次序，可引入指针p指引遍历，而引入指针pre跟踪p的前驱。首先将pre和p初始化为遍历的第一结点。然后让p往遍历的方向走找pre的后继。一旦找到，则它们互为前驱和后继，建立相应线索。接着将p赋给pre，重复下去直到遍历结束。

6.6 线索二叉树

6.6.4 二叉树的中序线索化

增加一个头结点，其LeftChild指针指向二叉树 的根结点，其RightChild指针指向中序遍历的最后一个结点。而最后一个结点的RightChild指针指向头结点。

这样一来，就好象为二叉树建立了一个双向线索链表，既可从中序遍历的第一个结点起进行中序的遍历；也可从中序遍历的最后一个结点起进行逆中序的遍历。

6.6 线索二叉树

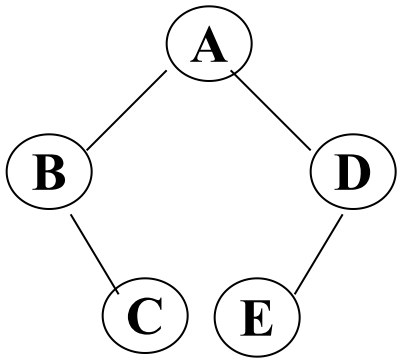
6.6.5 线索二叉树与非线索二叉树比较

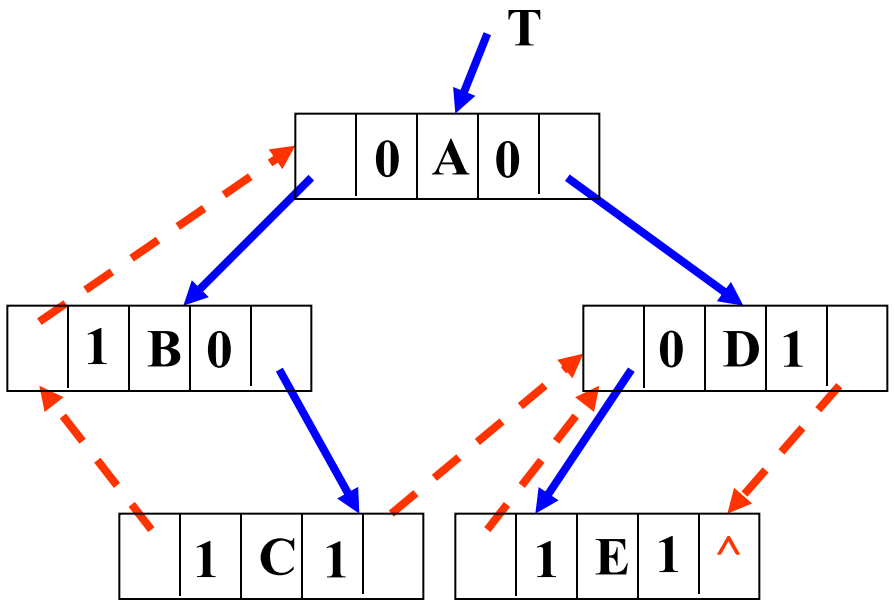
- 优点：

  对于找前驱和后继结点2种运算而言，线索二叉树优于非线索二叉树。

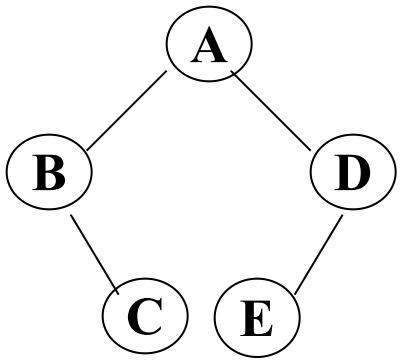
- 缺点：

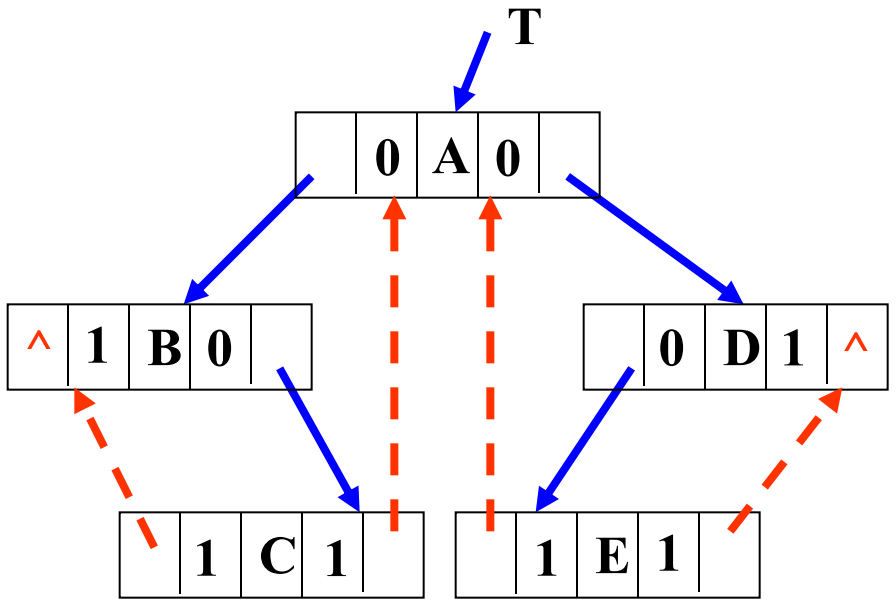
  在进行结点插入和删除运算时，线索二叉树比非线索二叉树的时间开销大。原因在于在线索二叉树中进行结点插入和删除时，除了修改相应指针外，还要修改相应的线索。



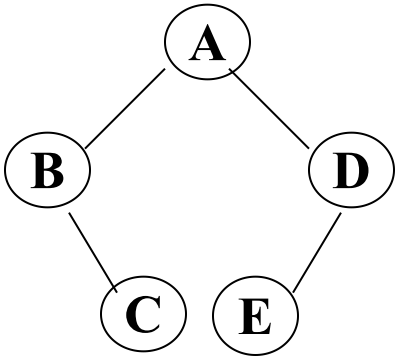


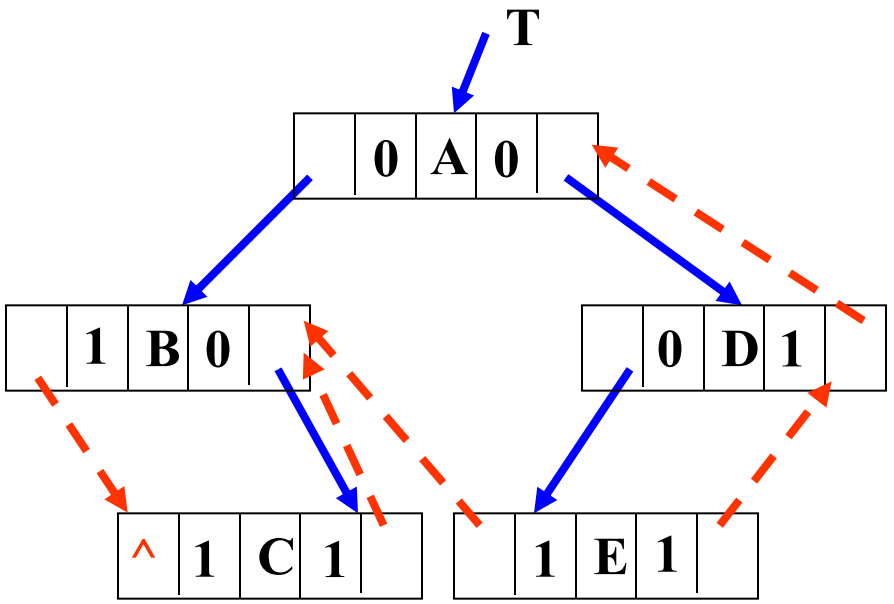
先序序列：ABCDE
先序线索二叉树



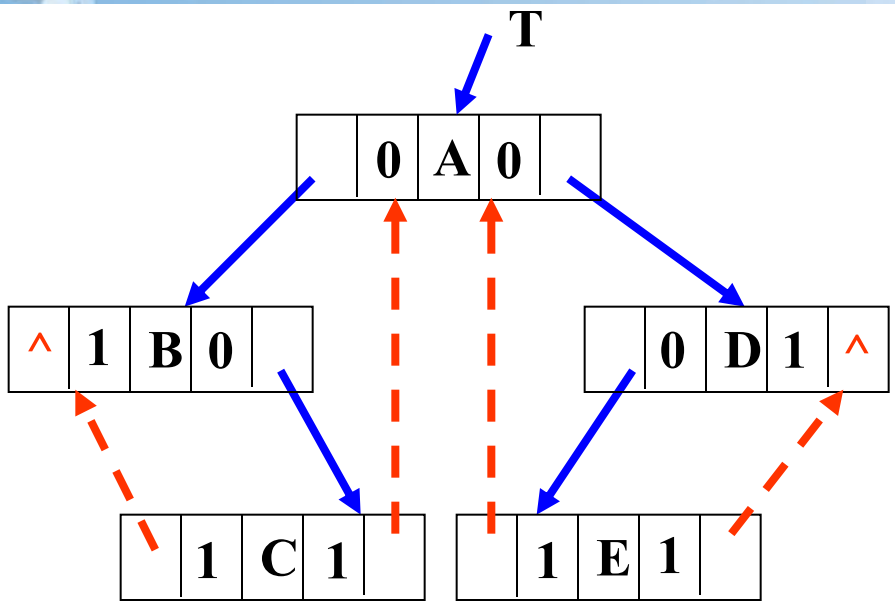


中序序列：BCAED
中序线索二叉树

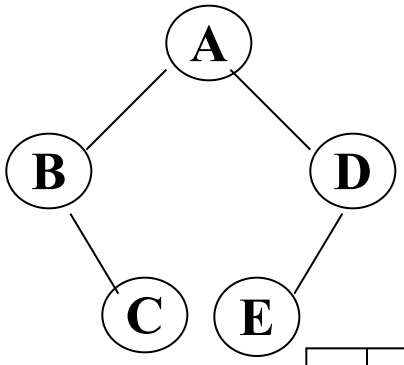


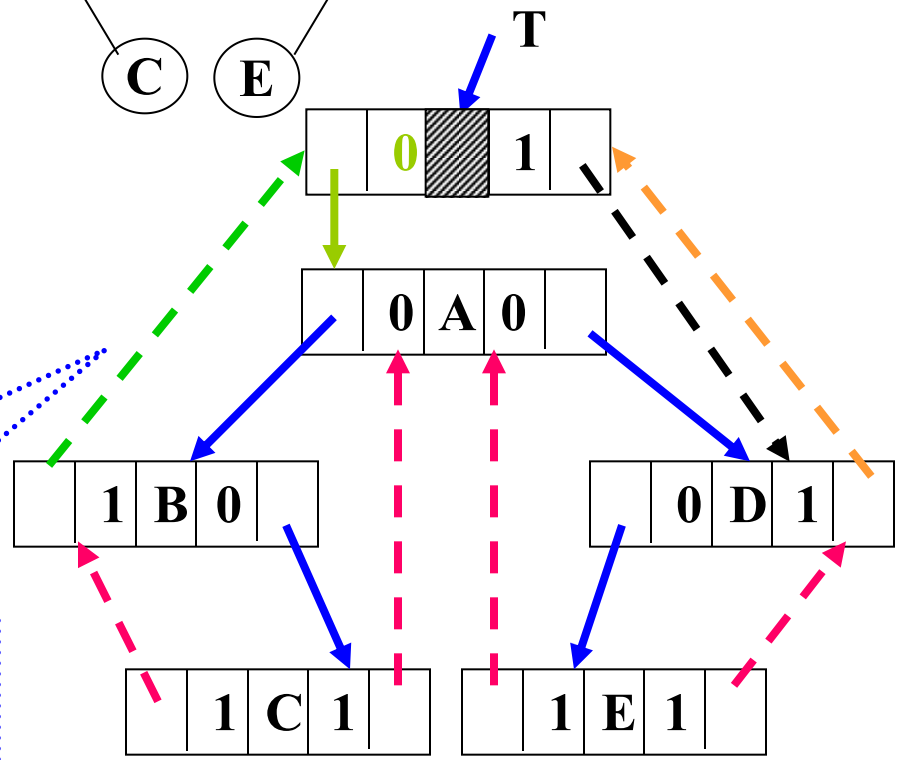


后序序列：CBEDA
后序线索二叉树



中序序列：BCAED
中序线索二叉树



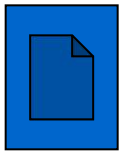


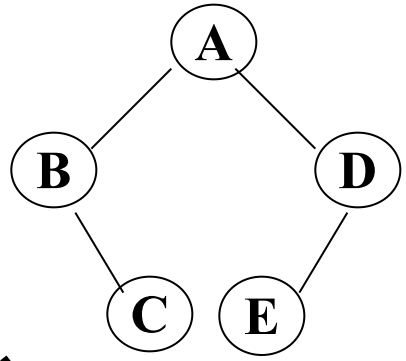
中序序列：BCAED
带头结点的中序线索二叉树

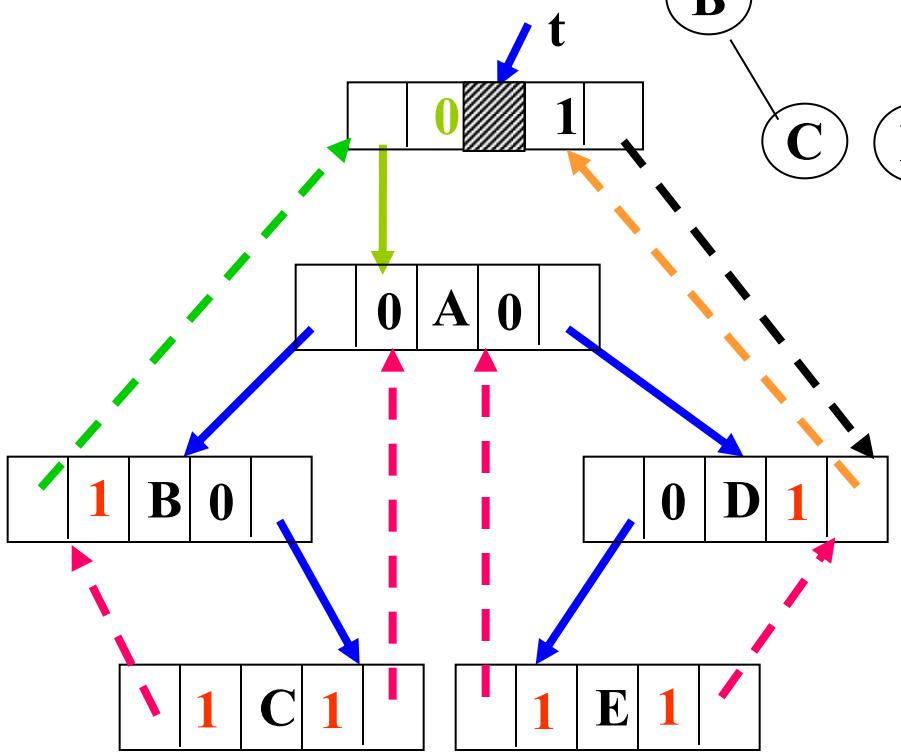
头结点：
lt=0, lc指向根结点
rt=1, rc指向遍历序列中最后一个结点
遍历序列中第一个结点的lc域和最后一个结点的rc域都指向头结点

- 算法
  - 按中序线索化二叉树


Ch5_20.txt

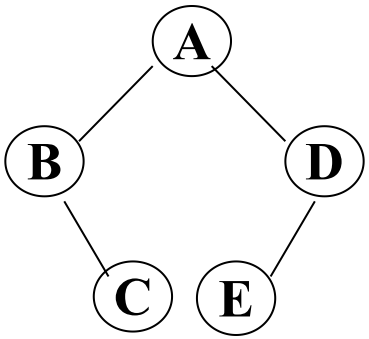

Ch5_20.c

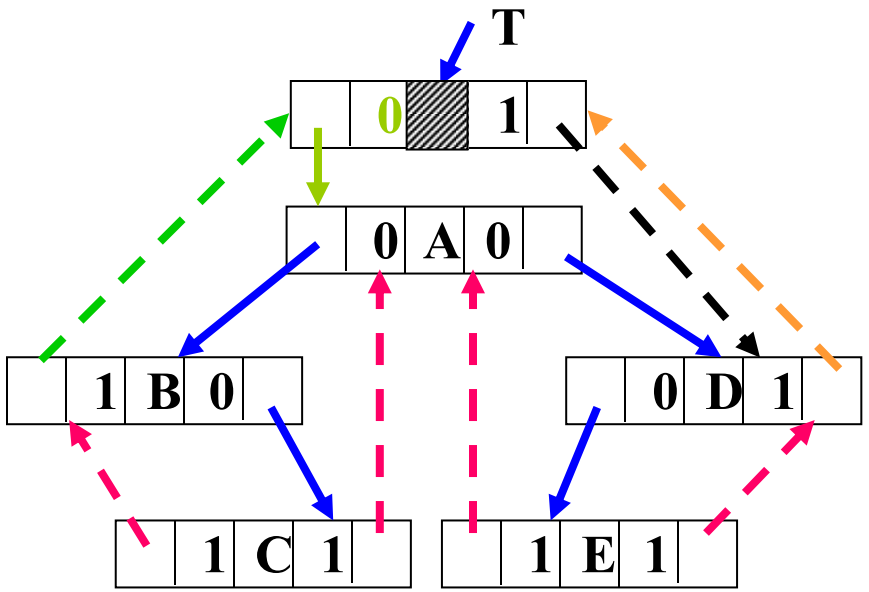




输出: B C A E D

- 算法
  - 按中序线索化二叉树 Ch5_21.txt
  - 遍历中序线索二叉树 Ch5_22.txt





中序序列：BCAED

在中序线索二叉树中找结点后继的方法： 带头结点的中序线索二叉树

（1）若rt=1, 则rc域直接指向其后继

（2）若rt=0, 则结点的后继应是其右子树的左链尾（lt=1)的结点

在中序线索二叉树中找结点前驱的方法：

（1）若lt=1, 则lc域直接指向其前驱

（2）若lt=0, 则结点的前驱应是其左子树的右链尾（rt=1)的结点

6.8.1字典的定义

- 当集合中的元素有一个线性序，即全集合是一个有序集时，往往涉及与这个线性序有关的一些集合运算。
- 例如对于集合S中的一个元素x，找它在集合S中按照线性序排列的前驱元素或后继元素的运算。
- 用符号表表示集合时，这类运算较难实现或实现的效率不高，为此引入另一个抽象数据类型——字典。

6.8.1 字典的定义

- 字典是以有序集为基础的抽象数据类型。
- 它支持以下运算：
  - (1)Member(x)，成员运算。
  - (2)Insert(x)，插入运算：将元素x插入集合。
  - (3)Delete(x)，删除运算：将元素x从当前集合中删去。
  - (4)Predecessor(x)，前驱运算：返回集合中小于x的最大元素。
  - (5)Successor(x)，后继运算：返回集合中大于x的最小元素。
  - (6)Range(x，y)，区间查询运算：返回集合中界于x和y之间，即x≤z≤y的所有元素z组成的集合。
  - (7)Min( )，最小元运算：返回当前集合中依线性序最小的元素。

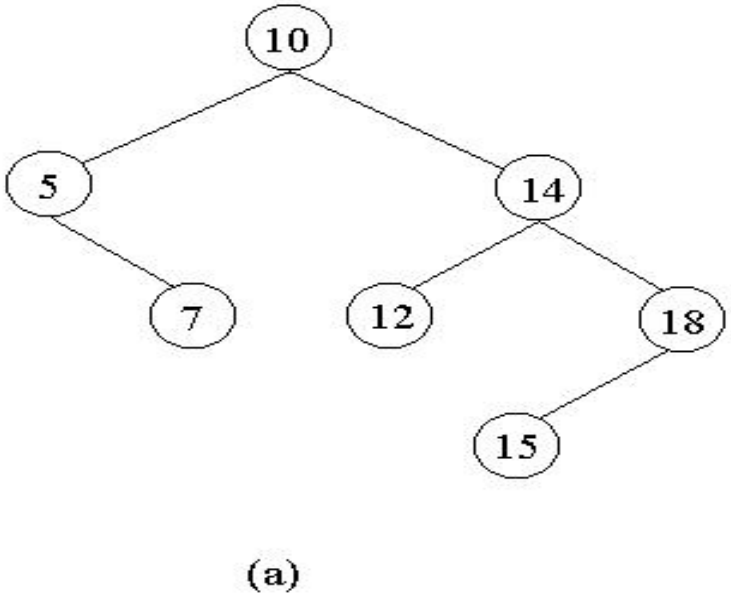
6.8.2 用数组实现字典

- 用数组实现字典与用数组实现符号表的不同之处：
  - 可以利用线性序将字典中的元素从小到大依序存储在数组中，通过数组下标来反映字典元素之间的序关系。
- 优点：可用二分法高效地实现与线性序有关的一些运算。
  - 如：**Member(x)**，**Predecessor(x)**和 **Successor(x)**可在时间***O(logn)***内实现。
- 缺点：插入和删除运算的效率较低。
  - 每执行一次**Insert**或**Delete**运算，需要移动部分数组元素，从而导致它们在最坏情况下的计算时间为***O(n)***。
- 考虑：能否用链表来实现字典？？？
  - **Member**运算需要***O(n)***时间，一旦找到元素在链表中插入或删除的位置后，只要用***O(1)***时间就可完成插入或删除操作。

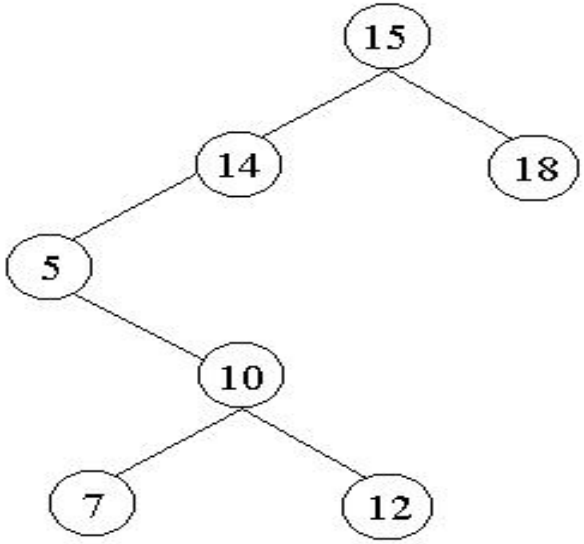
➔两种实现方式均不可取！

6.8.3 用二叉搜索树实现字典

- 基本思想：
  - 用二叉树来存储有序集，每一个结点存储一个元素。
  - 满足：存储于每个结点中的元素x大于其左子树中任一结点中所存储的元素，小于其右子树中任一结点中所存储的元素。



(a)



(b)

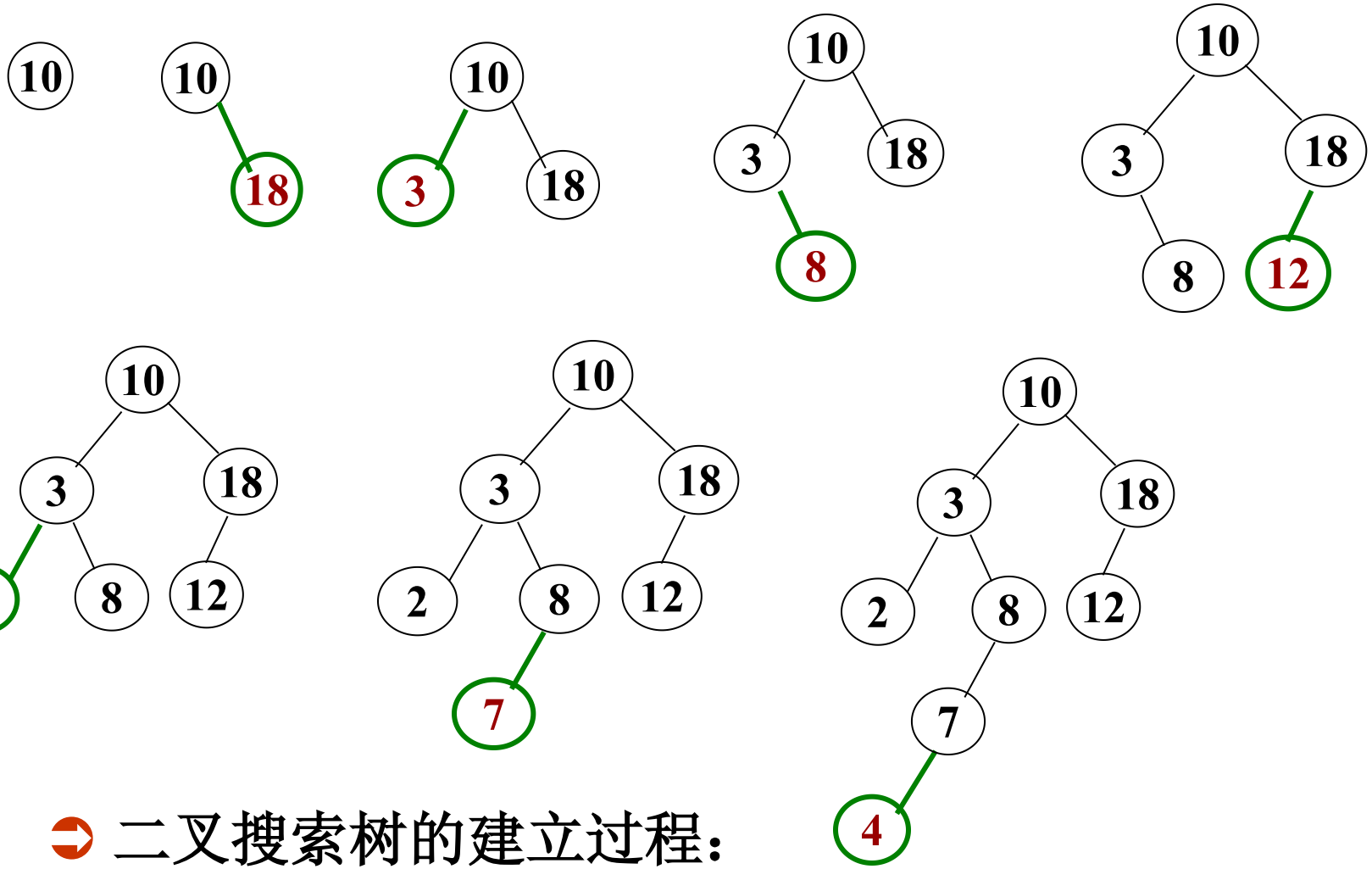
➔ 二叉搜索树结点的定义及程序实现

-
-

二叉搜索树的定义及运算的实现

-
-

例 {10， 18， 3， 8， 12， 2， 7， 4}



➲ 二叉搜索树的建立过程：

例

- 运算Delete(const T& x)的实现

运算Delete(const T& x)的实现

- 设要删除二叉搜索树中的结点p，分三种情况：
  - p为叶结点 ➔直接删除节点p
  - p只有左子树或右子树
    - p只有左子树 ➔用p的左儿子代替p
    - p只有右子树 ➔用p的右儿子代替p
  - p左、右子树均非空
    - 找p的左子树的最大元素结点（即p的前驱结点），用该结点代替结点p，然后删除该结点。

➔ 用二叉搜索树实现字典时间复杂性分析

- 最坏情况分析—member,insert,delete都需要*O(n)*
- 平均情况分析

  引入记号：记**p(n)**为含有**n**个结点的二叉搜索树的平均查找长度。显然**p(0)=0,p(1)=1;**

  若设某二叉搜索树的左子树有**i**个结点，则：

  p(i)+1为查找左子树中每个结点的平均查找长度;

  p(n-i-1)+1为查找右子树中每个结点的平均查找长度;

➲ 由此构造而得的二叉搜索树在**n**个结点的查找概率相等的情况下，其平均查找长度为：

$$q(n,i)=\frac{1}{n}\left(1+i\left(p(i)+1\right)+\left(n-i-1\right)\left(1+p(n-i-1)\right)\right)$$

又假设当前的二叉搜索树有n个结点，而它是从空树开始反复调用n次的Insert运算得到的，且被插入的n个元素的所有可能的顺序是等概率的。则：

$$p(n)=\frac{1}{n}(\sum_{i=0}^{n-1}q(n,i))$$

$$=\frac{1}{n}(\sum_{i=0}^{n-1}\frac{1}{n}(1+i(p(i)+1)+(n-i-1)(p(n-i-1)+1)))$$

$$=1+\frac{1}{n^2}\sum_{i=0}^{n-1}\left(ip(i)+(n-i-1)p(n-i-1)\right)$$

$$=1+\frac{2}{n^2}\sum_{i=0}^{n-1}ip(i)$$

对n用数学归纳法可以证明：$p(n)\le 1+4\log n$

当n=1时显然成立。若设i<n时有 $p(i)\le 1+4\log i$ ，则

$$p(n) \le 1 + \frac{2}{n^2}\sum_{i=1}^{n-1} i(1 + 4\log\ i)$$

$$\le 1 + \frac{2}{n^2}\sum_{i=1}^{n-1} 4i\log\ i + \frac{2}{n^2}\sum_{i=1}^{n-1} i$$

$$\le 2 + \frac{8}{n^2}\sum_{i=1}^{n-1} i\log i$$

略去 -1/n 项

$$\le 2 + \frac{8}{n^2}\left(\sum_{i=1}^{n/2-1} i\log(n/2) + \sum_{i=n/2}^{n-1} i\log n\right)$$

$$\le 2 + \frac{8}{n^2}\left(\frac{n^2}{8}\log(n/2) + \frac{3n^2}{8}\log n\right)$$

$$= 2 + \frac{8}{n^2}\left(\frac{n^2}{2}\log n - \frac{n^2}{8}\right)$$

$$= 1 + 4\log n$$

➲ 平均情况下的时间复杂度为： $O(\log\ n)$

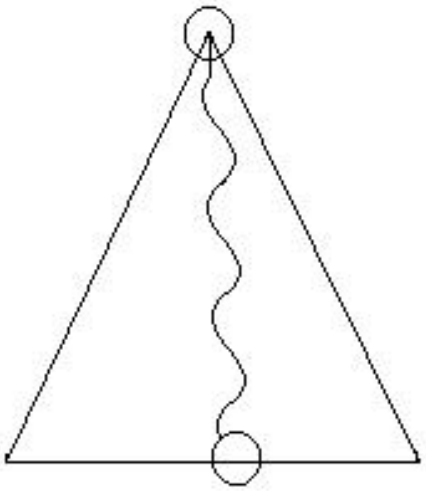
- 运算Predecessor(x)和Successor(x)的实现
  ——类似于Search(x)算法
- 运算Range(y, z)的实现：可借助于Search(y)和Successor(y)运算
  - 首先，用Search(y)检测y是否在二叉搜索树中，是则输出y，否则不输出y；
  - 然后，从y开始，不断地用Successor找当前元素在二叉搜索树中的后继元素。当找出的后继元素x满足x≤z时，就输出x，并将x作为当前元素。重复这个过程，直到找出的当前元素的后继元素大于z，或二叉搜索树中已没有后继元素为止。
  - 时间复杂度：若二叉树搜索树中有r个元素x满足y≤x≤z，则在最坏情况下用*O(rn)*时间，在平均情况下用*O(rlogn)*时间可实现Range运算。

运算Range(y,z)的改进

考虑半无限查询区域 $[y,+\infty)$,

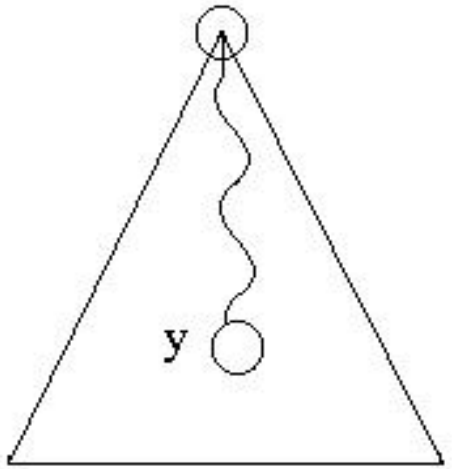
——即找出二叉搜索树中满足y ≤ x的所有元素x。

当y不在二叉搜索树中时，产生一条从根到叶的路径。如下图(a)



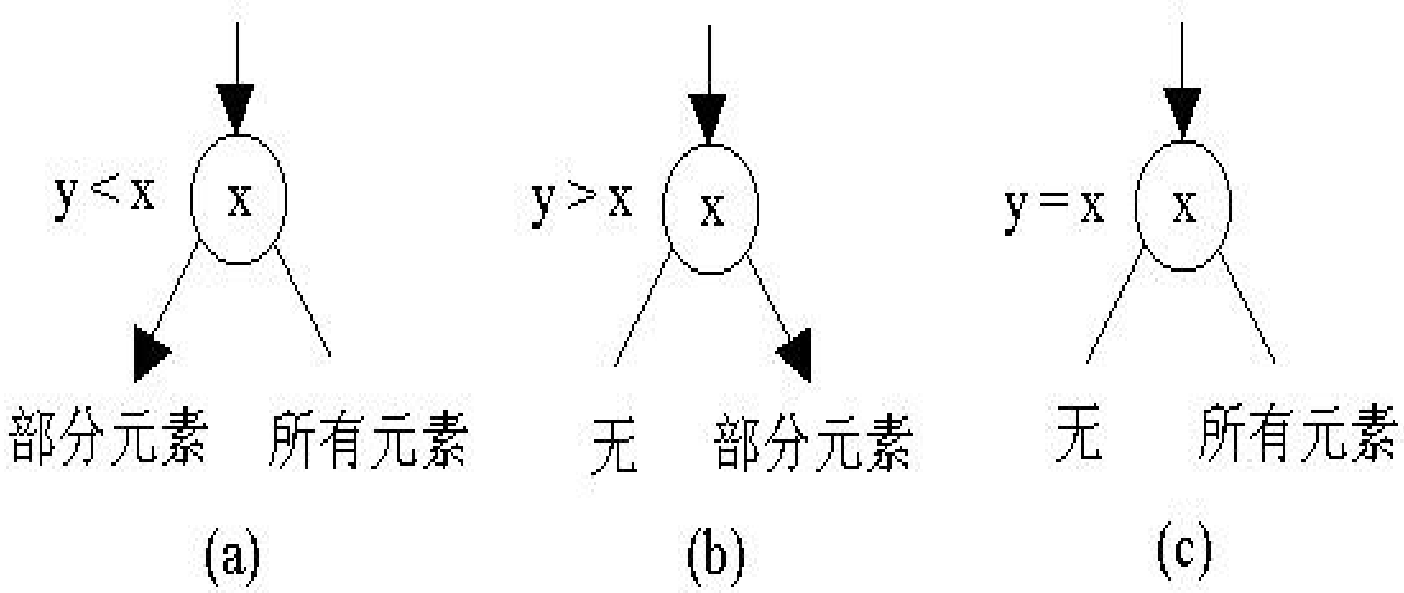
(a)

当y在二叉搜索树中时，产生一条从根到存储元素y的结点的路径。如下图(b)



(b)

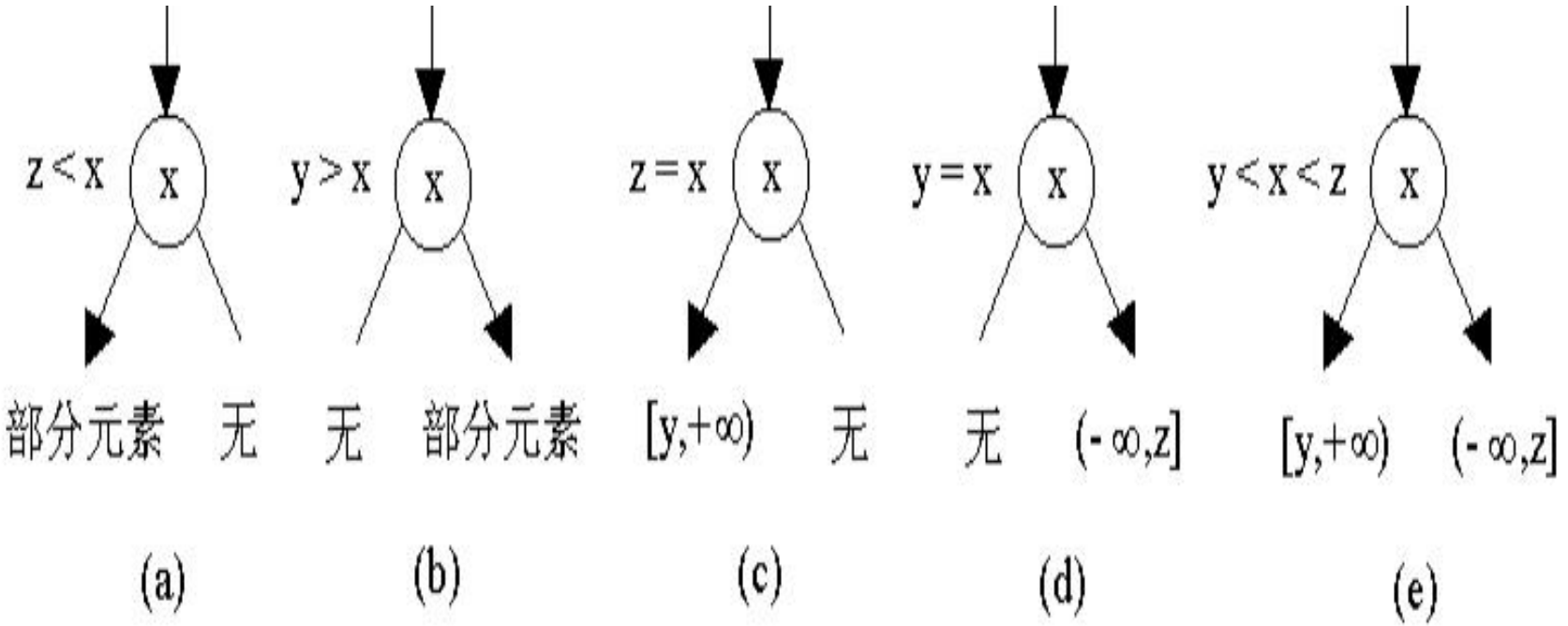
- 在找到的搜索路径上的所有结点可分为以下3种情况，如下图所示：



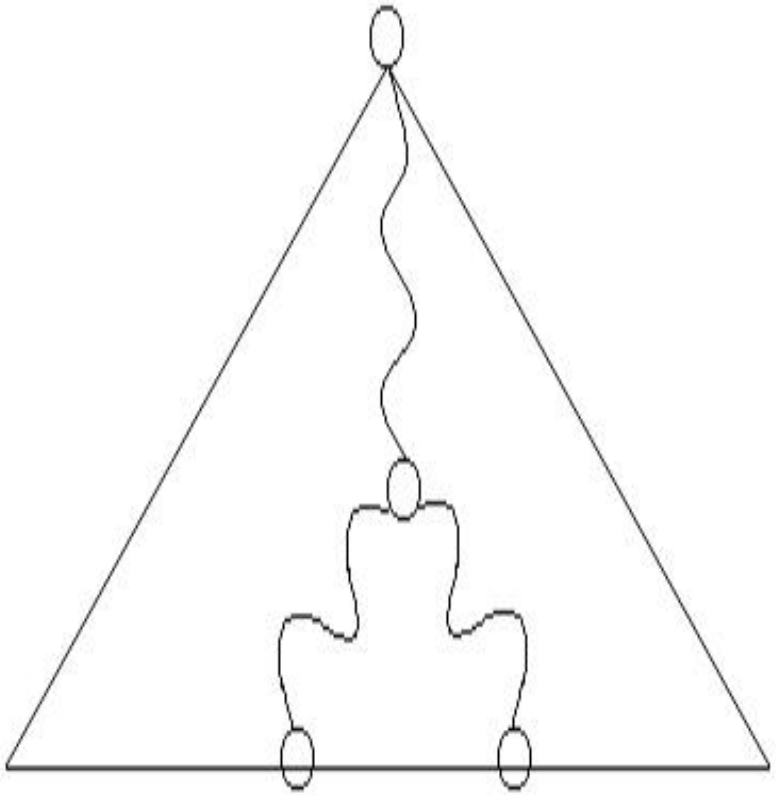
- 运算Range(y,z)的实现

  ——可用类似于Range(y,∞)算法

  - 从二叉搜索树的根结点开始，同时与y和z比较,此时,结点分类的情况可能有（见下图）：



- 运算Range(y,z)的搜索路径如下图：



平衡的二叉搜索树--AVL树

引言—AVL树产生的背景

- 问题的提出:用二叉搜索树实现有序字典在最坏情况下— member,insert,delete都需要O(n);平均情况下需要O(log n)。
  问在最坏情况下能降到O(log n)吗?

平衡的二叉搜索树AVL树

引言—AVL树产生的背景(续)

- 解决问题的设想:
  - n个结点的二叉树最矮是近似满二叉树,其高为[log n]。若放宽此限制为每一个结点的左子树与右子树高度差的绝对值不超过1，则二叉树当然就达不到最矮,却可望接近最矮,而不超过0(log n),目的就达到了。这正是AVL树。剩下的问题是设法找一种在insert和delete后只需0(log n)时间的维护算法。
- 设想的证实:
  - (1) n个结点的AVL树的高度为O(log n);
  - (2) insert和delete后的维护算法在最坏的情况下只需O(logn)的时间。

平衡的二叉搜索树AVL树

AVL树的定义和性质

- 递归定义:
  - 空的和单结点的二叉搜索树都是AVL树；结点数大于1的二叉搜索树，若满足左子树和右子树都是AVL树且左、右子树高度之差的绝对值不超过1,那么,它是AVL树。
- 性质:

(1)AVL树T的结点数n与高度h的关系。

设高度h的AVL树的最少结点数N(h)。N(h)一定出现在树的左、右子树中一棵高为h-1,而另一棵高为h-2时。则N(h)满足如下递归方程：

$$N(h)=\begin{cases}1 & h=0\\ 2 & h=1\\ N(h-1)+N(h-2)+1 & h>1\end{cases}$$

平衡的二叉搜索树AVL树

解上面的递归方程得：

$$N(h)=F(h+2)-1=\left(\left((1+\sqrt{5})/2\right)^{h+2}-\left((1-\sqrt{5})/2\right)^{h+2}\right)/\sqrt{5}-1$$

由于 $$\left(\left((1+\sqrt{5})/2\right)^{h+2}-\left((1-\sqrt{5})/2\right)^{h+2}\right)/\sqrt{5}>\left(1+\sqrt{5}\right)^{h+2}/\sqrt{5}-1$$

因此 $$n\geq N(h)>\left((1+\sqrt{5})/2\right)^{h+2}/\sqrt{5}-2$$

$$h+2<\log_{\frac{1+\sqrt{5}}{2}}(n+2)+\log_{\frac{1+\sqrt{5}}{2}}\sqrt{5}$$

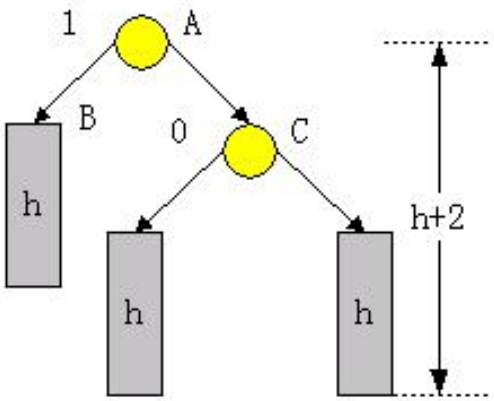
$$h<\log_{\frac{1+\sqrt{5}}{2}}(n+2)+\log_{\frac{1+\sqrt{5}}{2}}\sqrt{5}-2\leq 1.4404\log(n+2)-0.328$$

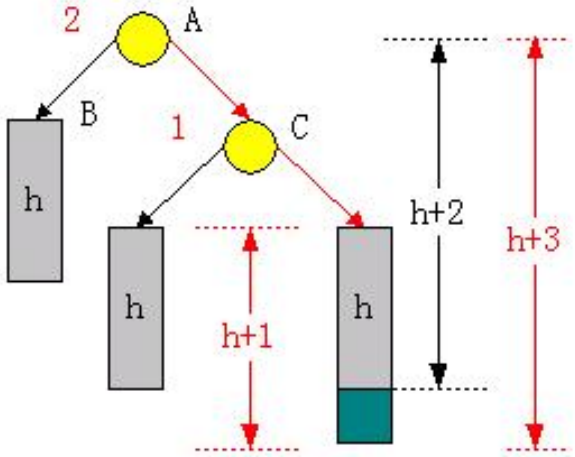
平衡的二叉搜索树AVL树

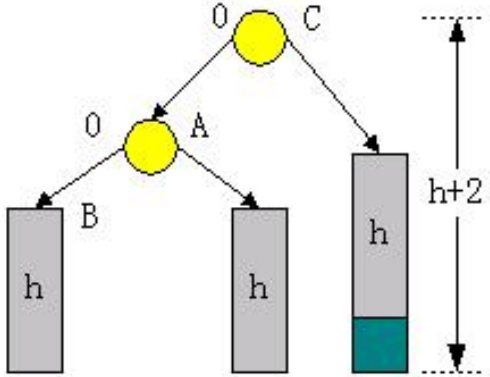
旋转变换

- 旋转变换的目的：是调整结点的子树高度，并维持二叉搜索树性质，即结点中元素的中序性质。
- 旋转变换分为单旋转变换和双旋转变换2种类型。
- 单旋转变换又分为右单旋转变换和左单旋转变换。
- 双旋转变换又分为先左后右双旋转变换和先右后左双旋转变换。

1、左单旋的情况

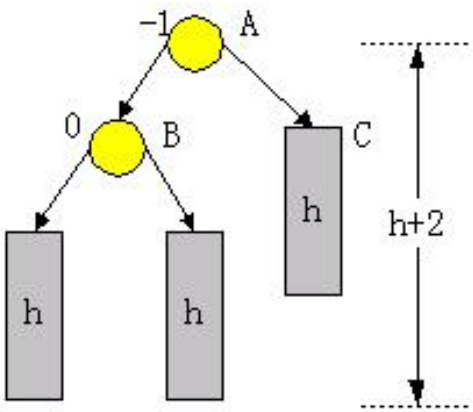




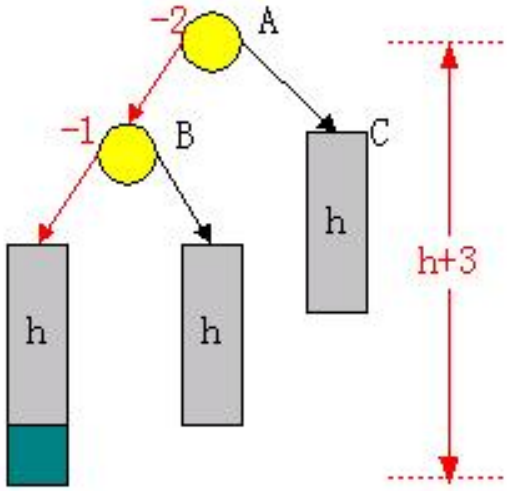


原来的AVL树　　插入一结点，A点不平衡　　左单旋的结果

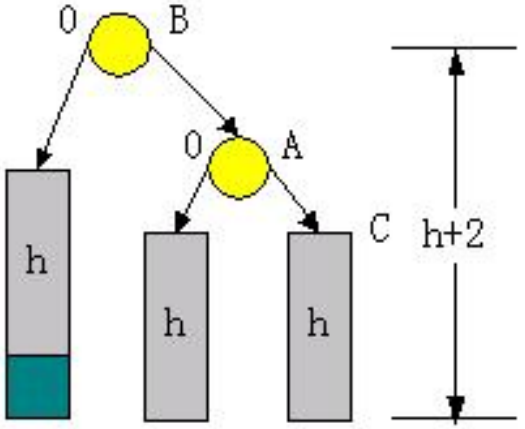
2.右单旋的情况



原来的AVL树

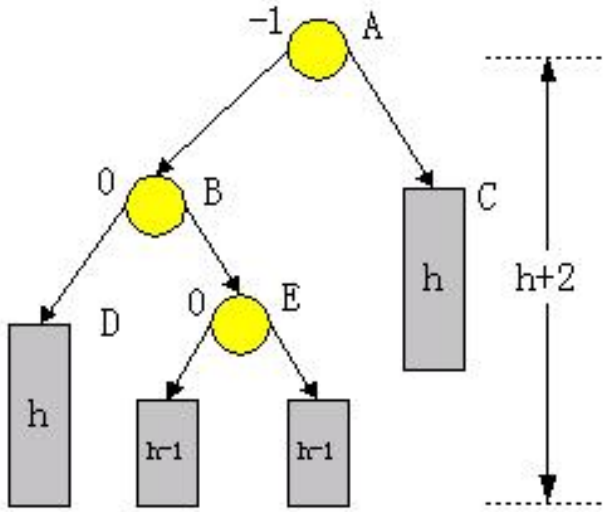


插入一结点，A点不平衡

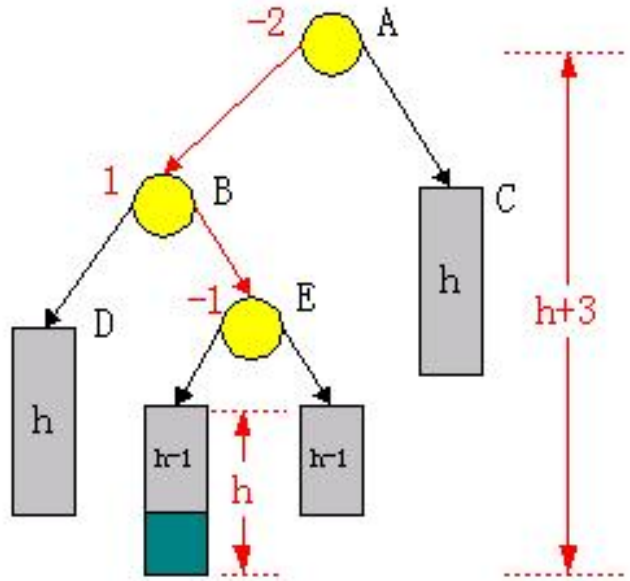


右单旋的结果

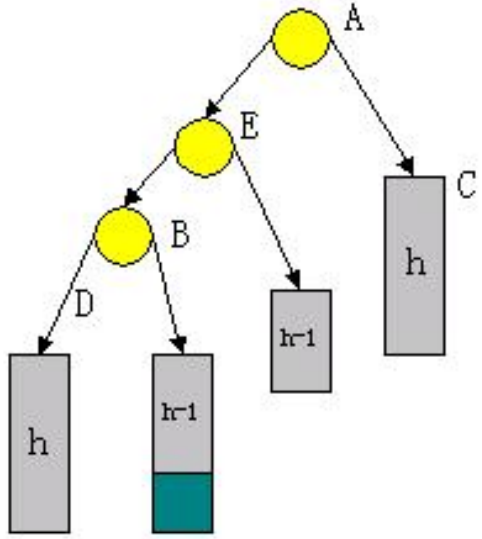
3.先左后右双旋的情况



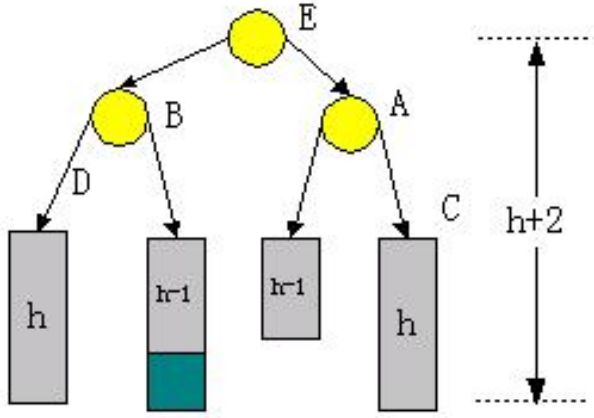
原来的AVL树



插入一结点，A点不平衡

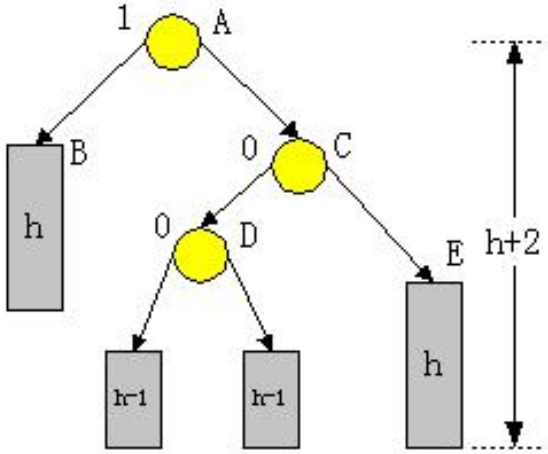


先左旋

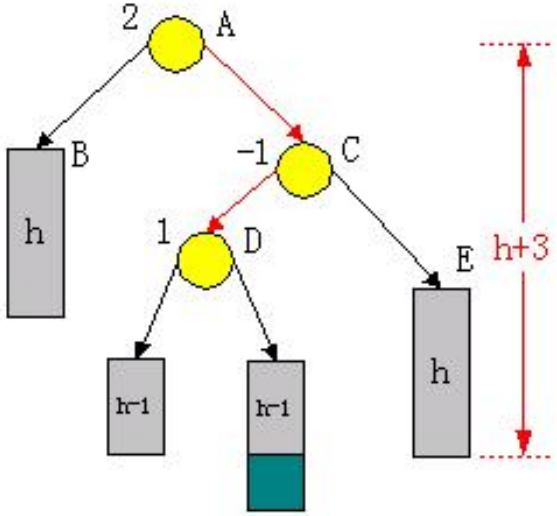


再右旋

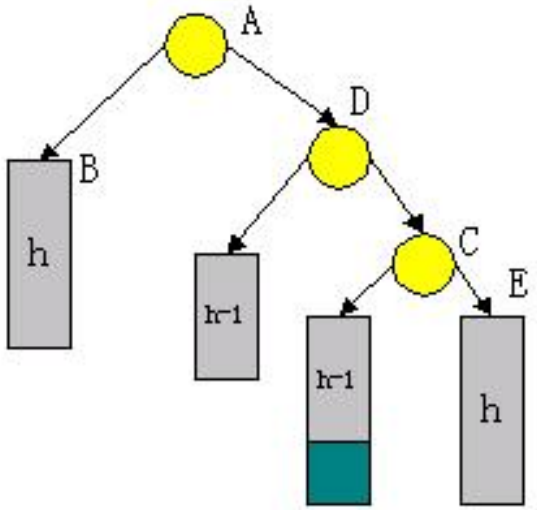
4.先右后左双旋的情况



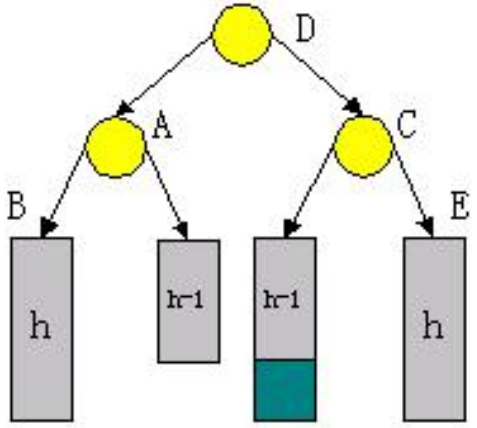
原来的AVL树



插入一结点，A点不平衡



先右旋



再左旋

平衡的二叉搜索树AVL树

AVL树的插入运算

- AVL树与二叉搜索树的插入运算是类似的。惟一的不同之处是，在AVL树中执行1次二叉搜索树的插入运算，可能会破坏AVL树的高度平衡性质，因此需要重新平衡。
- 设新插入的结点为v。从根结点到结点v的路径上，每个结点处插入运算所进入的子树高度可能增1。因此在执行1次二叉搜索树的插入运算后，需从新插入的结点v开始，沿此插入路径向根结点回溯，修正平衡因子，调整子树高度，恢复被破坏的平衡性质。

平衡的二叉搜索树AVL树

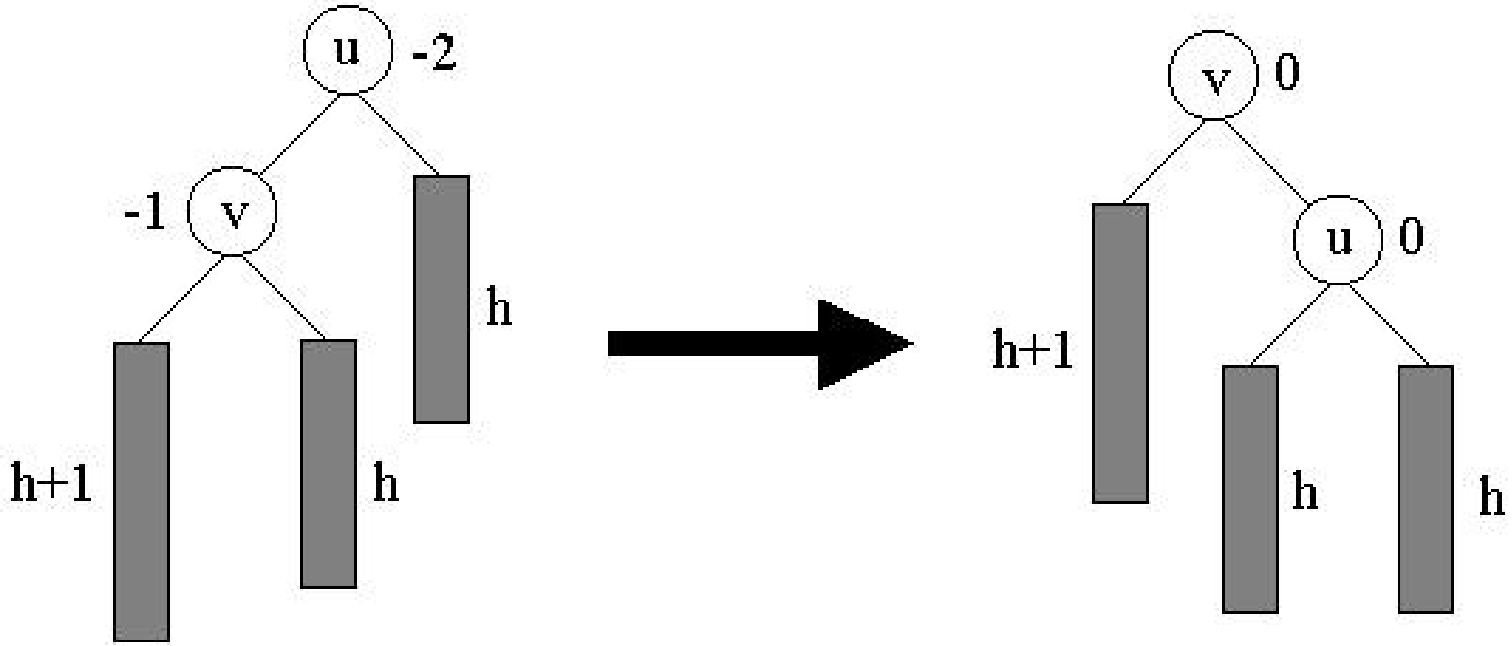
AVL树的插入运算

- 新结点v的平衡因子为0。现考察v的父结点u。若v是u的左儿子结点，则bal(u)应当减1，否则bal(u)应当增1。根据修正后的bal(u)的值分以下3种情形讨论。
- 情形1：bal(u)=0。此时以结点u为根的子树平衡，且其高度不变。因此从根结点到结点u的路径上各结点子树高度不变，从而各结点的平衡因子不变。此时可结束重新平衡过程。
- 情形2：| bal(u) | = 1。此时以结点u为根的子树满足平衡条件，但其高度增1。此时将当前结点向根结点方向上移，继续考察结点u的父结点的平衡状态。

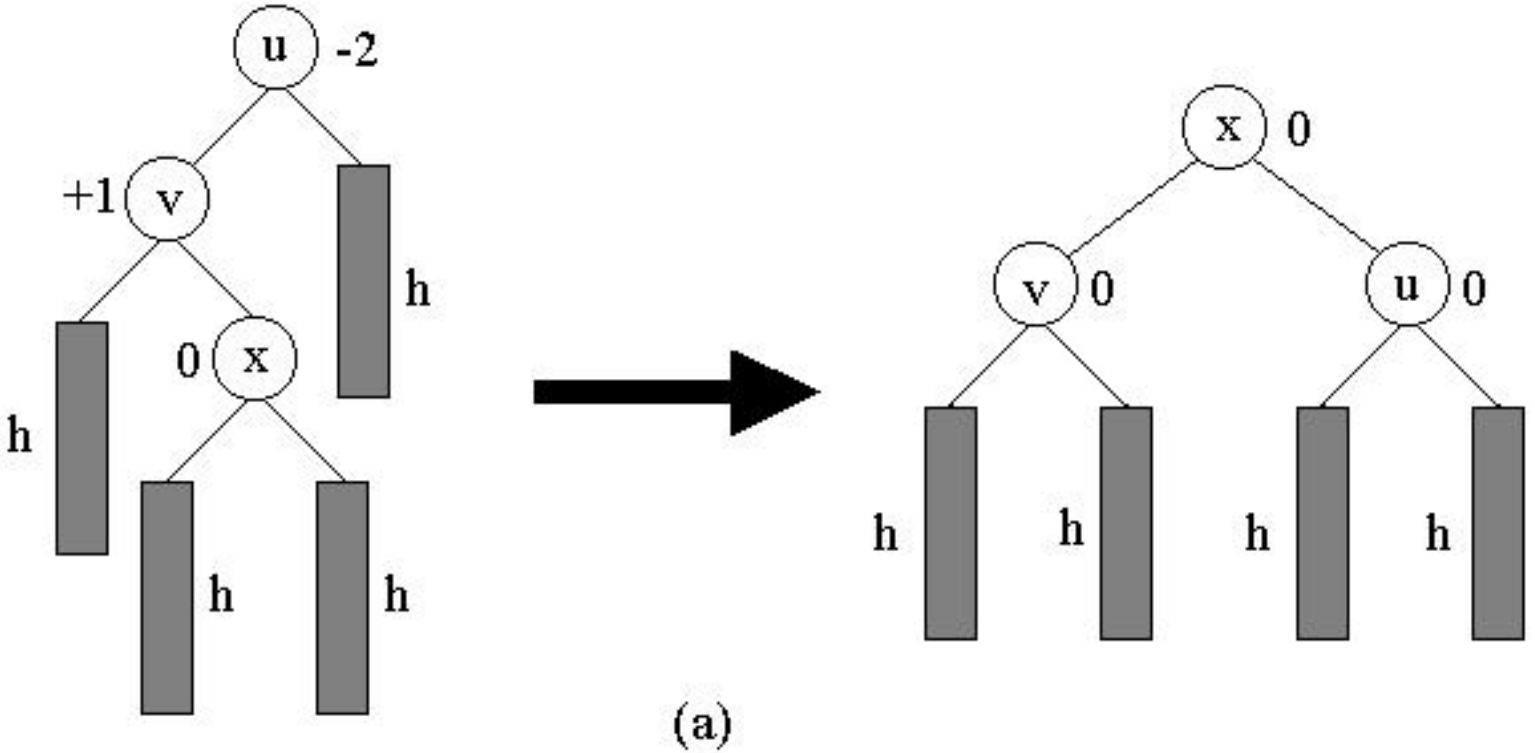
平衡的二叉搜索树AVL树

- 情形3：| bal(u) | = 2。 先讨论bal(u)=-2的情形。易知，此时结点v是结点u的左儿子结点，且bal(v)≠0。又可分为2种情形。
  - 情形3.1：bal(v)=-1。此时作1次右单旋转变换后，结束重新平衡过程。
  - 情形3.2：bal(v)=1。此时结点v的右儿子结点x非空。根据bal(x)的值，又分为bal(x)=0、bal(x)=-1和bal(x)=1的3种情形。在这3种情形下，分别作1次双旋转变换后，结束重新平衡过程。

情形3.1：

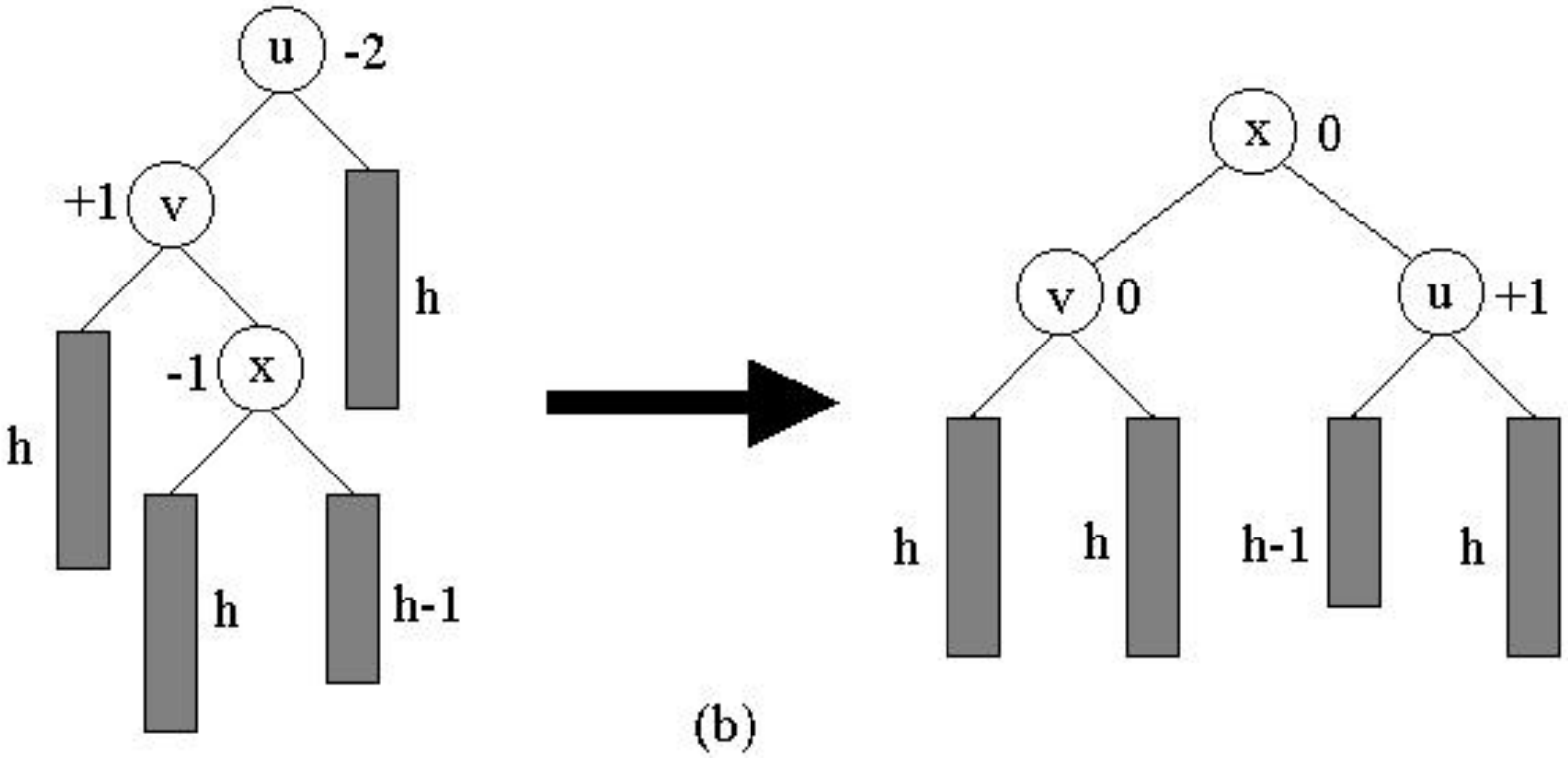


情形3.2： bal(x)=0

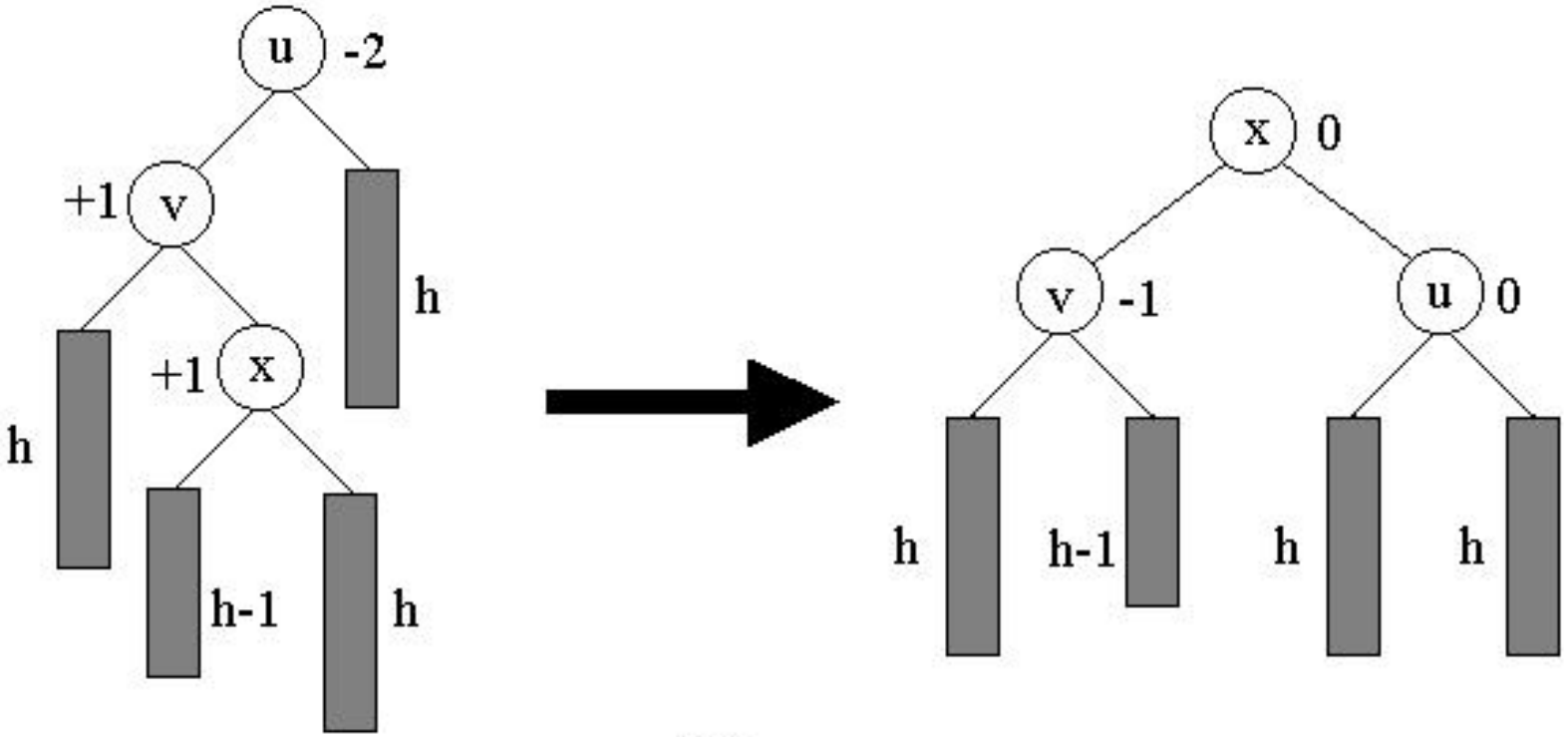


(a)

情形3.2： bal(x)=-1

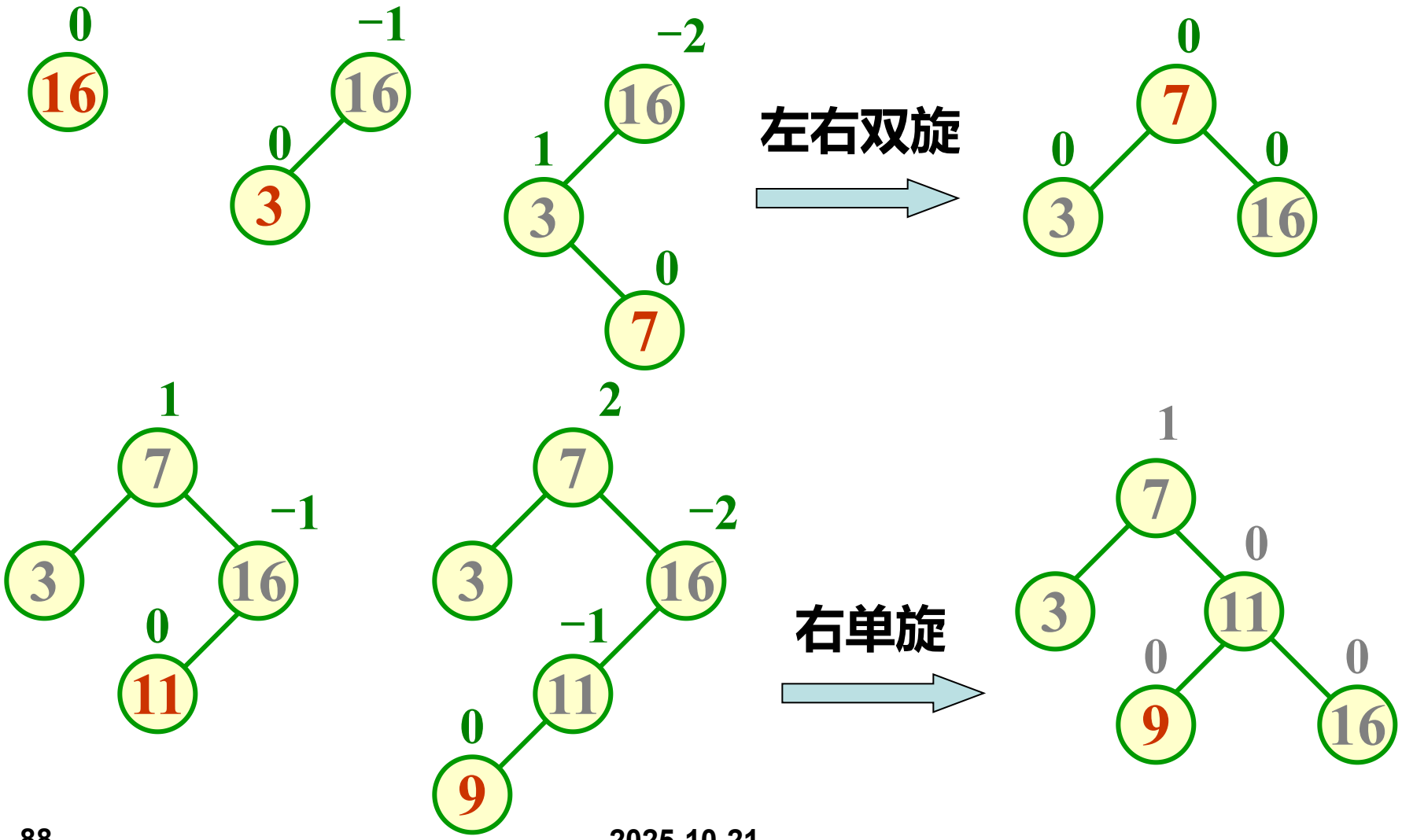


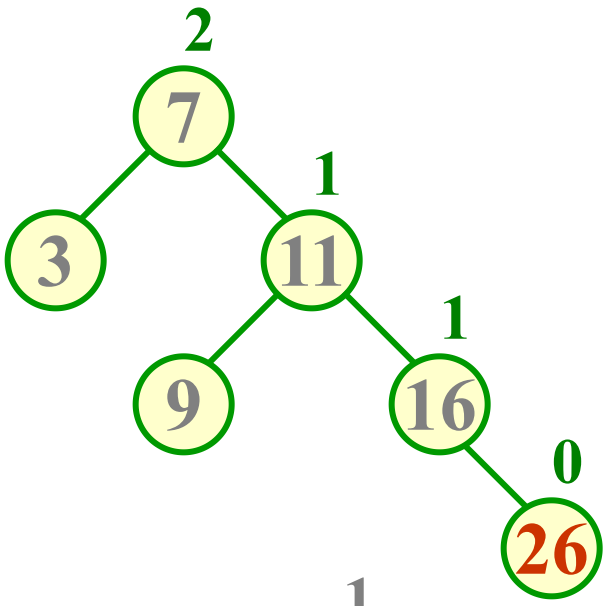
情形3.2： bal(x)=1



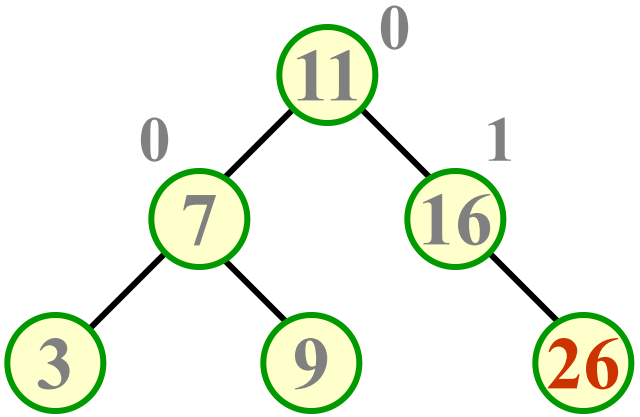
(c)

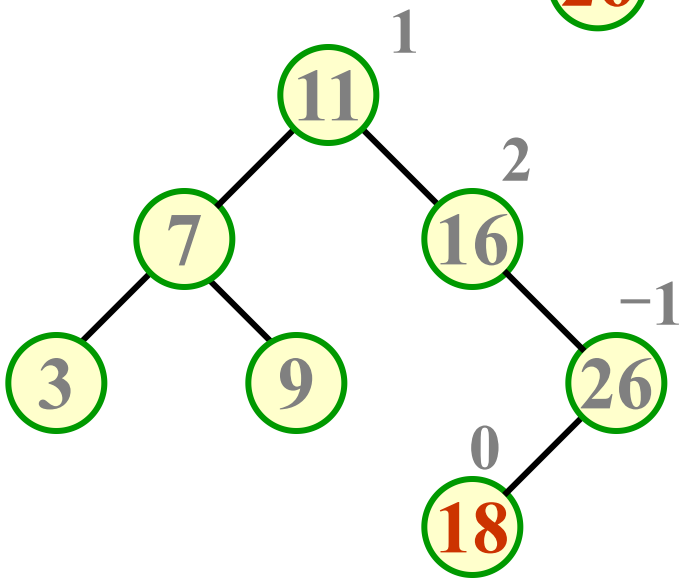
例:关键码序列为{16,3,7,11,9,26,18,14,15},插入和调整过程如下。



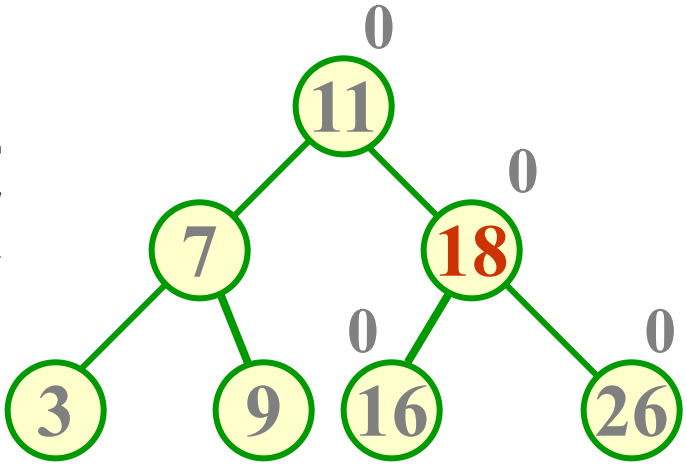

2
7
1
3
11
1
9
16
0
26

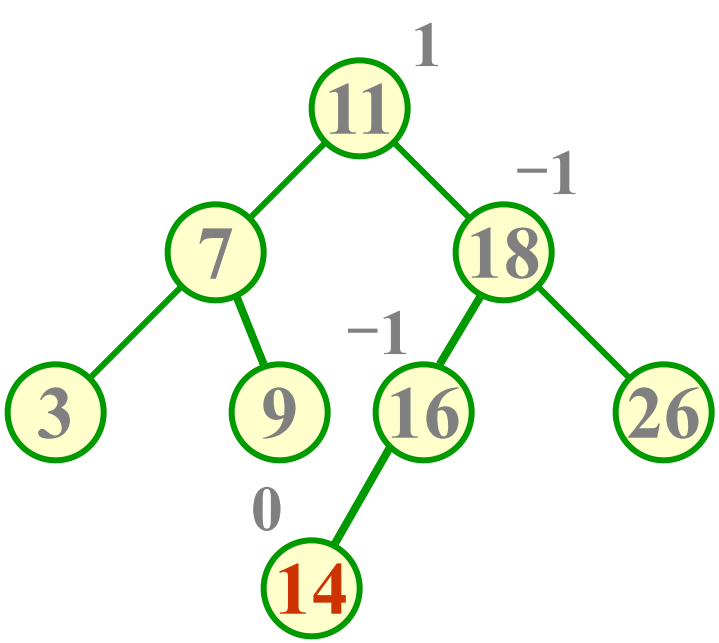
左单旋

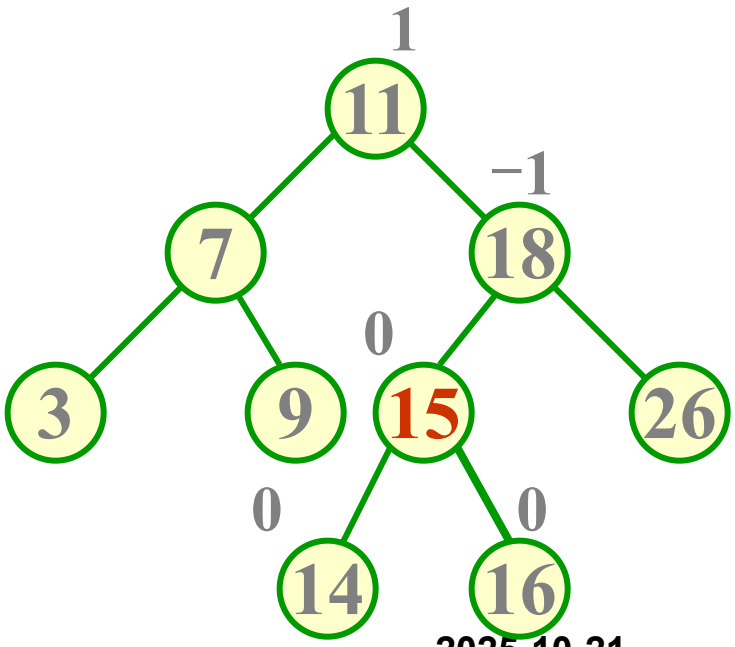

0
11
0
1
7
16
3
9
26


1
11
2
7
16
−1
3
9
26
0
18

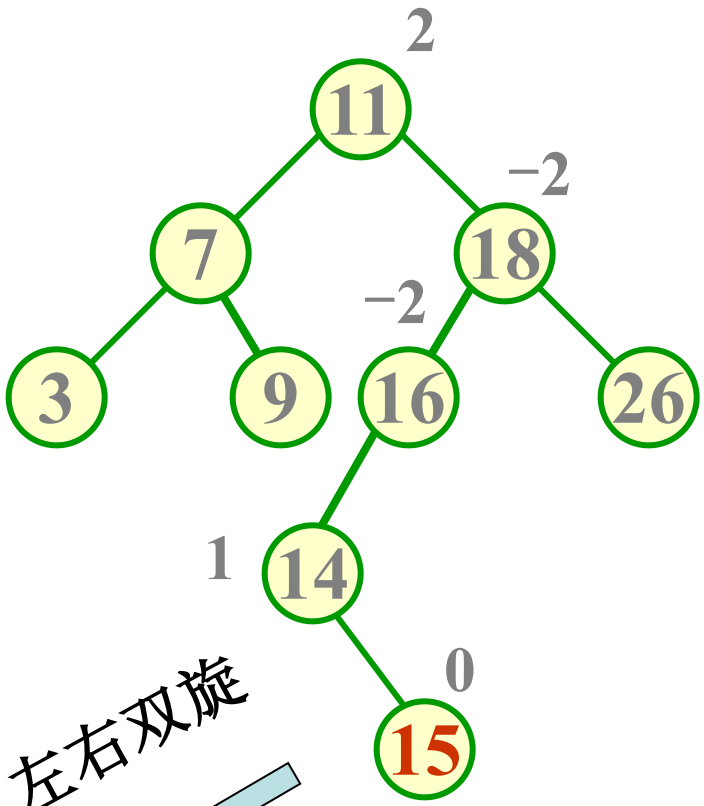
右左双旋


0
11
0
7
18
0
0
3
9
16
26


1
11
−1
7
18
−1
3
9
16
26
0
14


1
11
−1
7
18
0
3
9
15
26
0
0
14
16

左右双旋


2
11
−2
7
18
−2
3
9
16
26
1
14
0
15

平衡的二叉搜索树AVL树

AVL树的删除运算

- AVL树与二叉搜索树的删除运算是类似的。惟一的不同之处是，在AVL树中执行1次二叉搜索树的删除运算，可能会破坏AVL树的高度平衡性质，因此需要重新平衡。
- 设被删除结点为p，其惟一的儿子结点为v。结点p被删除后，结点v取代了它的位置。从根结点到结点v的路径上，每个结点处删除运算所进入的子树高度可能减1。因此在执行1次二叉搜索树的删除运算后，需从结点v开始，沿此删除路径向根结点回溯，修正平衡因子，调整子树高度，恢复被破坏的平衡性质。

平衡的二叉搜索树AVL树

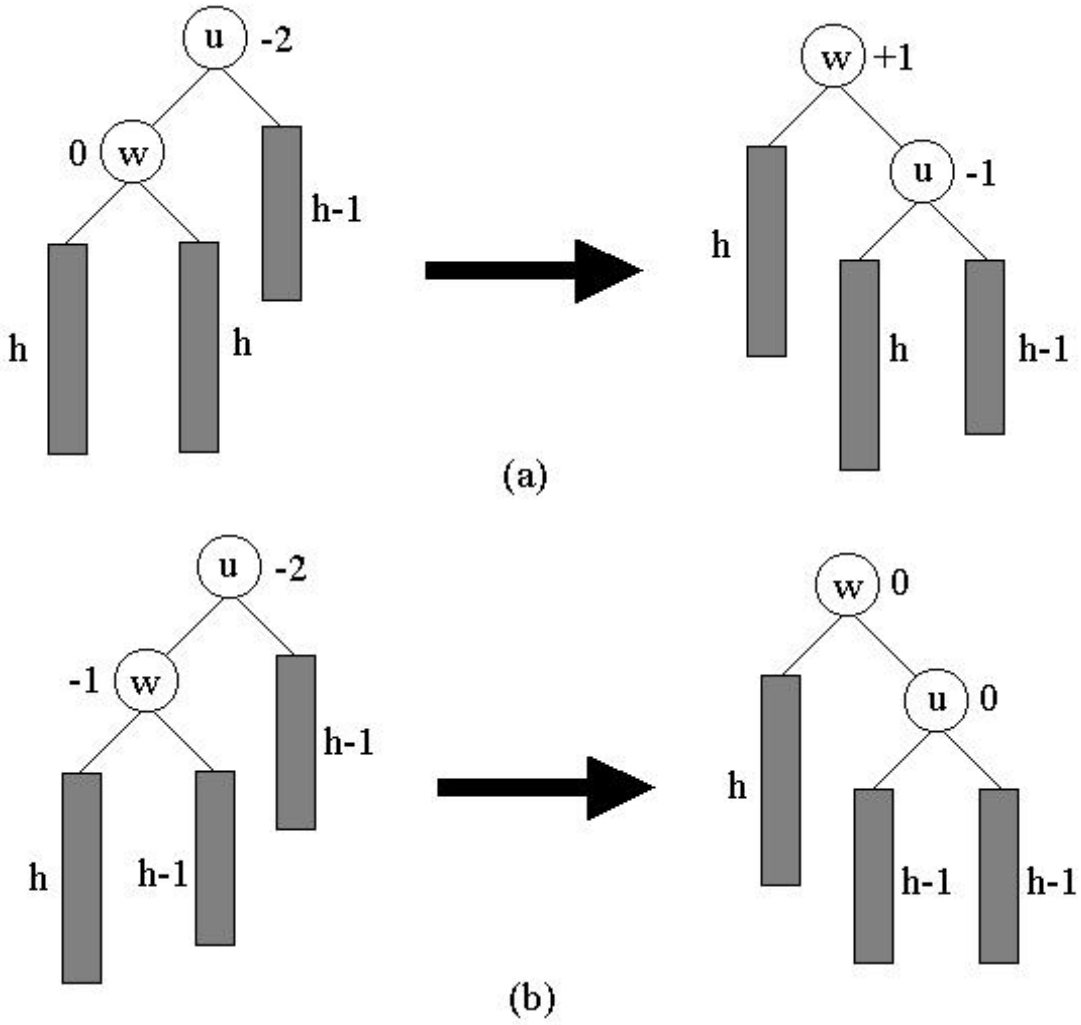
AVL树的删除运算

- 考察v的父结点u。若v是u的左儿子结点，则bal(u)应当增1，否则bal(u)应当减1。根据修正后的bal(u)的值分以下3种情形讨论。
- 情形1：| bal(u) | = 1。此时以结点u为根的子树满足平衡条件，且其高度不变。因此从根结点到结点u的路径上各结点子树高度不变，从而各结点的平衡因子不变。此时可结束重新平衡过程。
- 情形2：bal(u)=0。此时以结点u为根的子树平衡，但其高度减1。此时将当前结点向根结点方向上移，继续考察结点u的父结点的平衡状态。

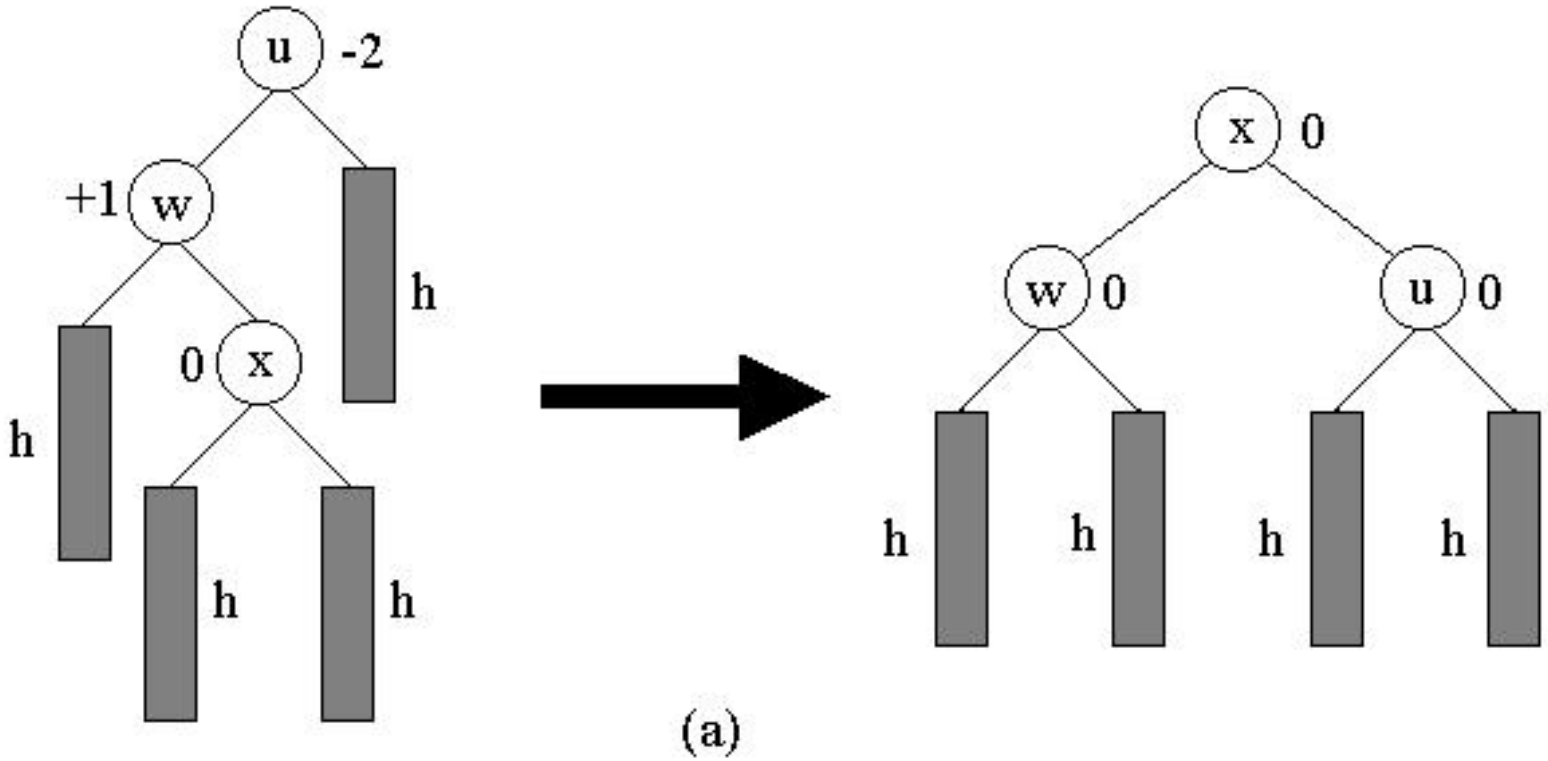
平衡的二叉搜索树AVL树

AVL树的删除运算

- 情形3：| bal(u) | = 2。 先讨论bal(u)=-2的情形。易知，此时结点v是结点u的右儿子结点。考察结点u的左儿子结点w，根据bal(w)的值又可分为2种情形。
  - 情形3.1：bal(w)≠1。此时作1次右单旋转变换后，使结点u恢复平衡。bal(w)=0和bal(w)=-1的情形分别如图 (a)和(b)所示。
  - 情形3.2：bal(w)=1。此时结点w的右儿子结点x非空。根据bal(x)的值，又分为bal(x)=0、bal(x)=-1和bal(x)=1的3种情形。在这3种情形下，分别作1次双旋转变换后，使结点u恢复平衡。

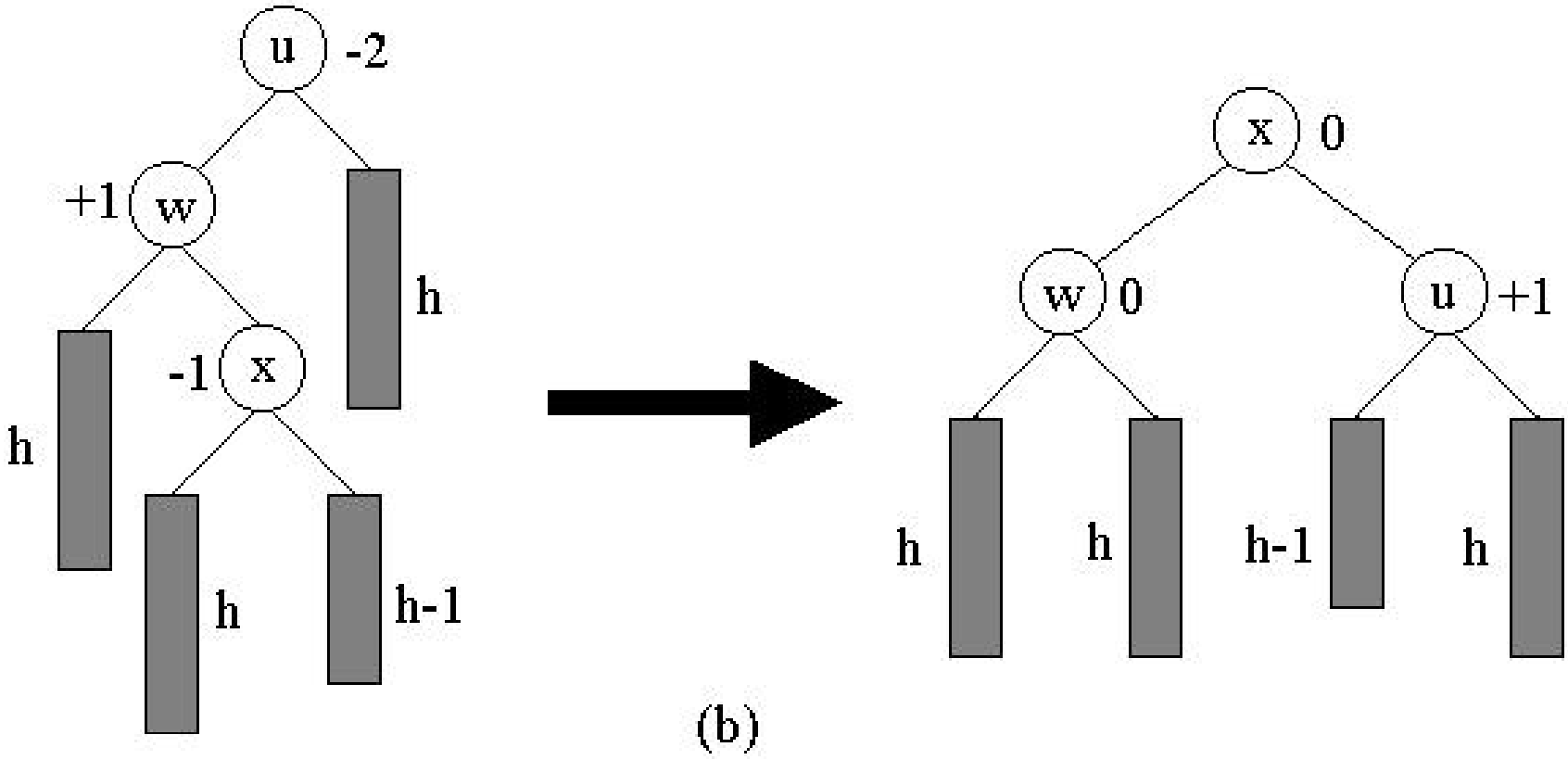


删除重新平衡的情形3.1



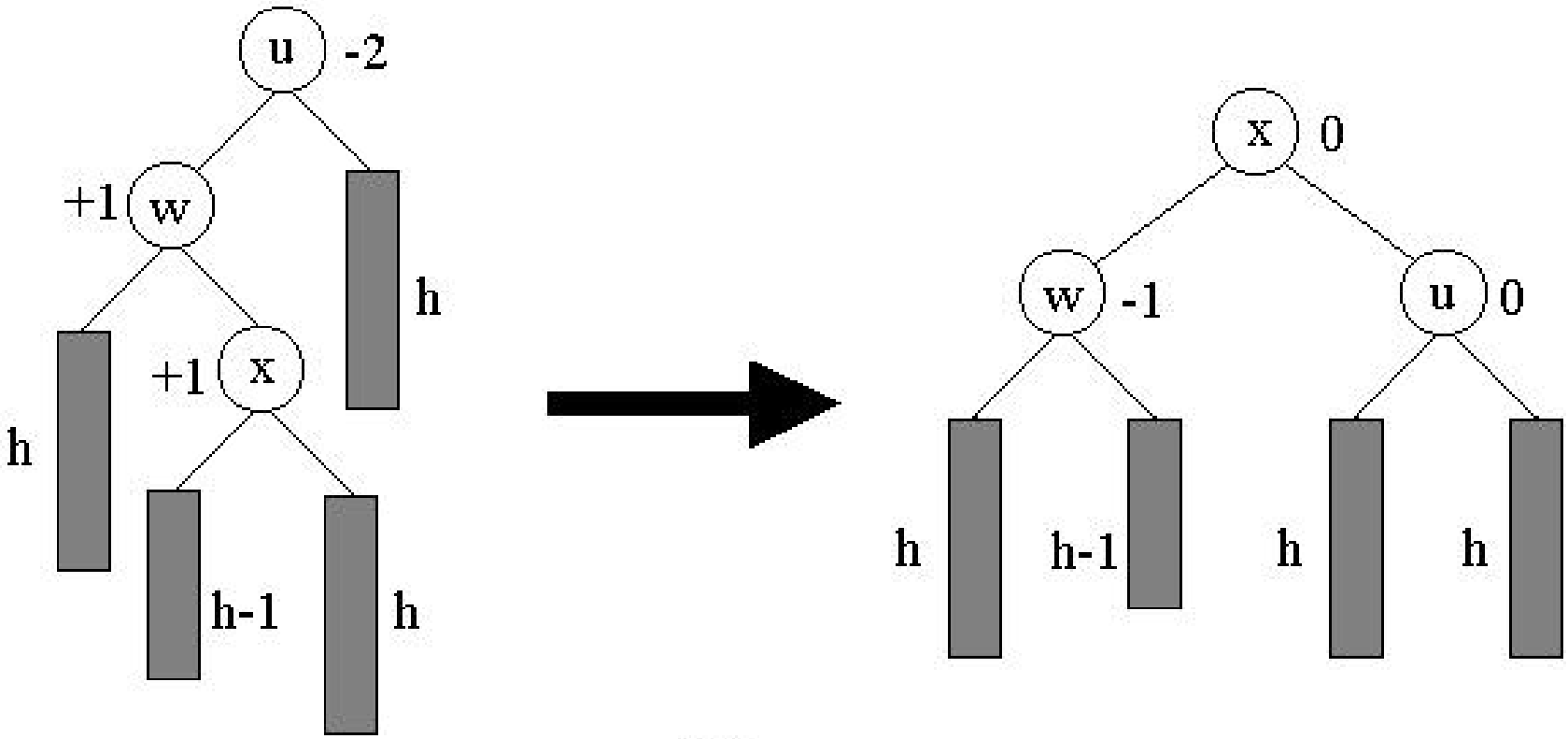
(a)

删除重新平衡的情形3.2



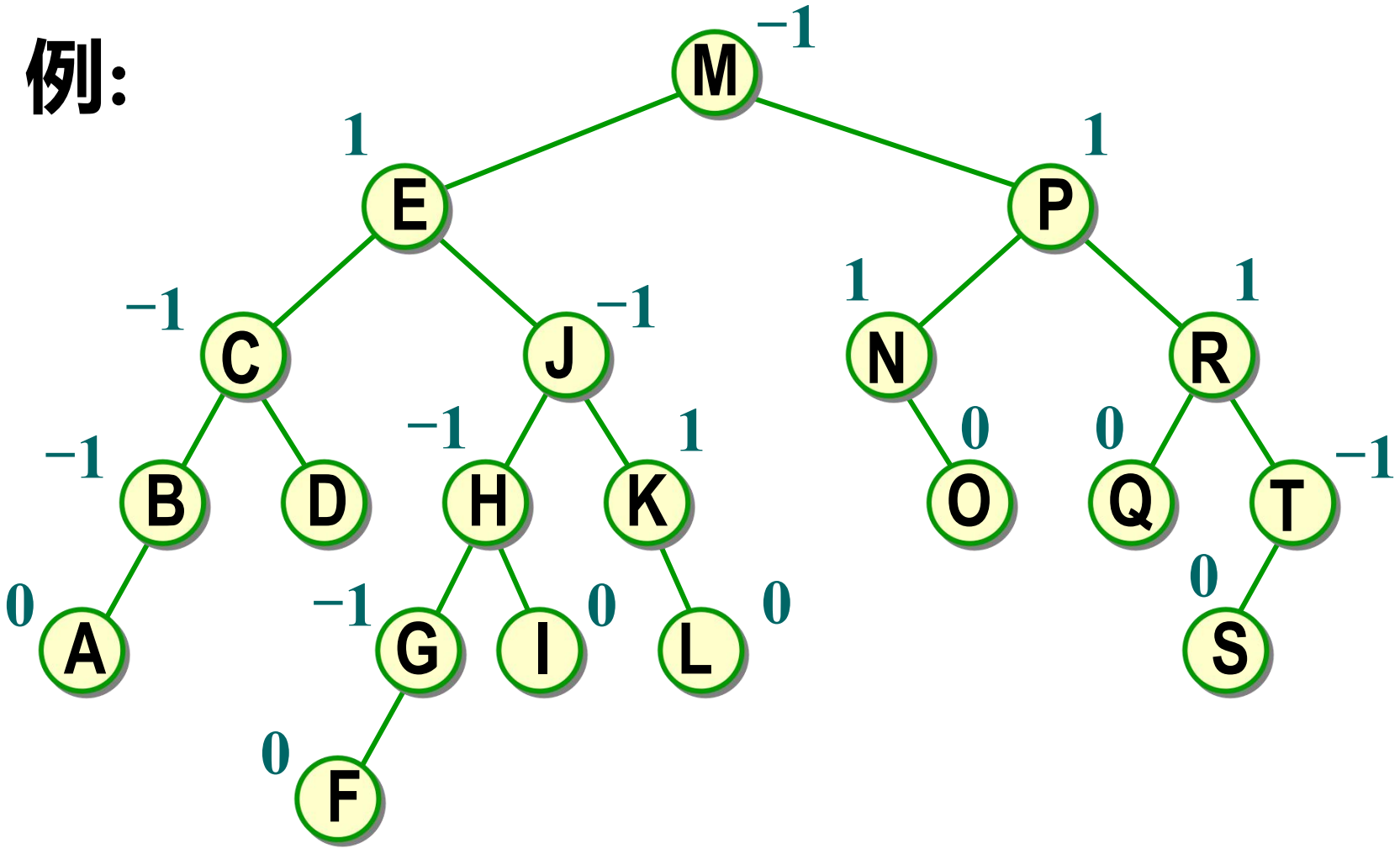
(b)

删除重新平衡的情形3.2



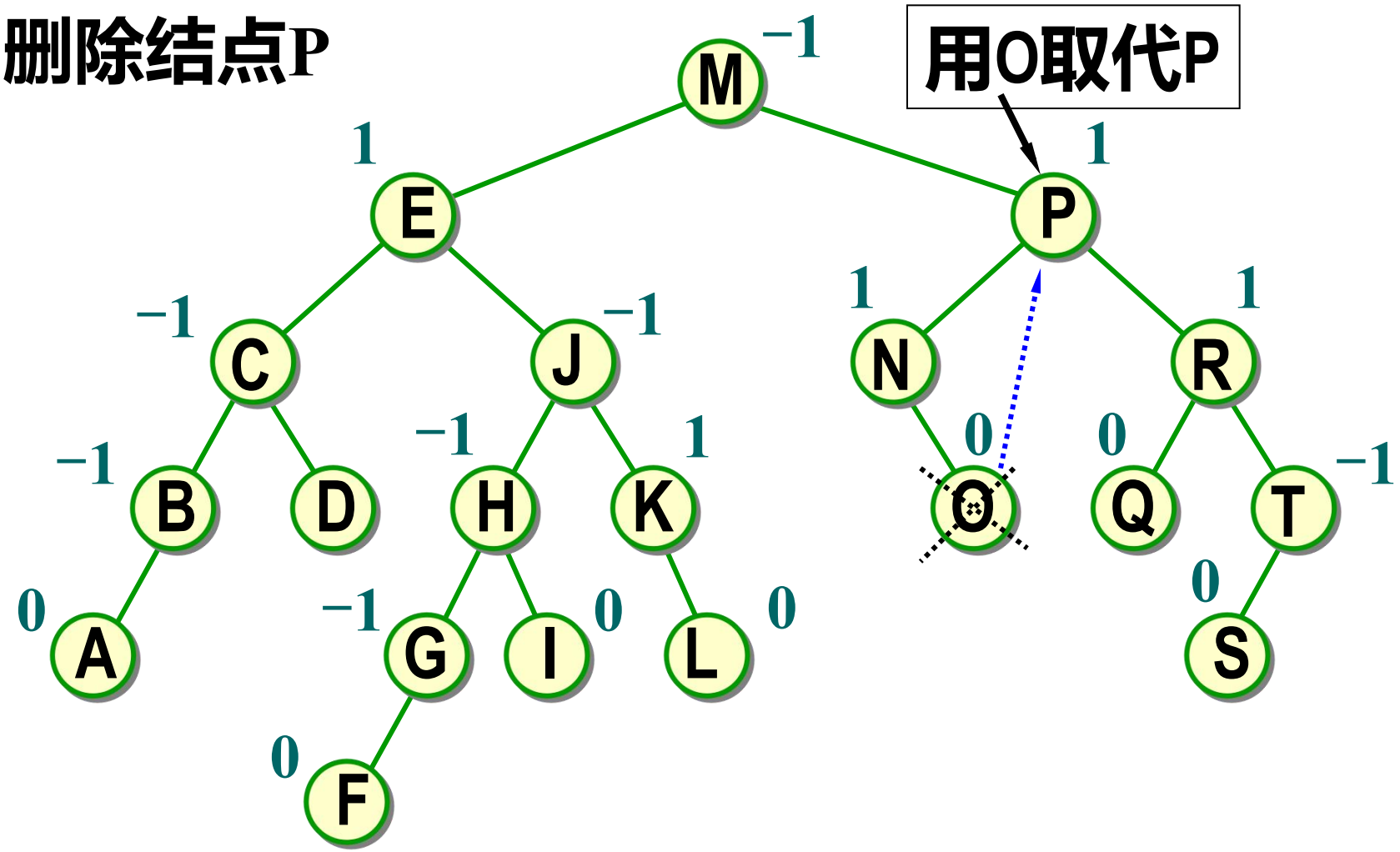
删除重新平衡的情形3.2

例:



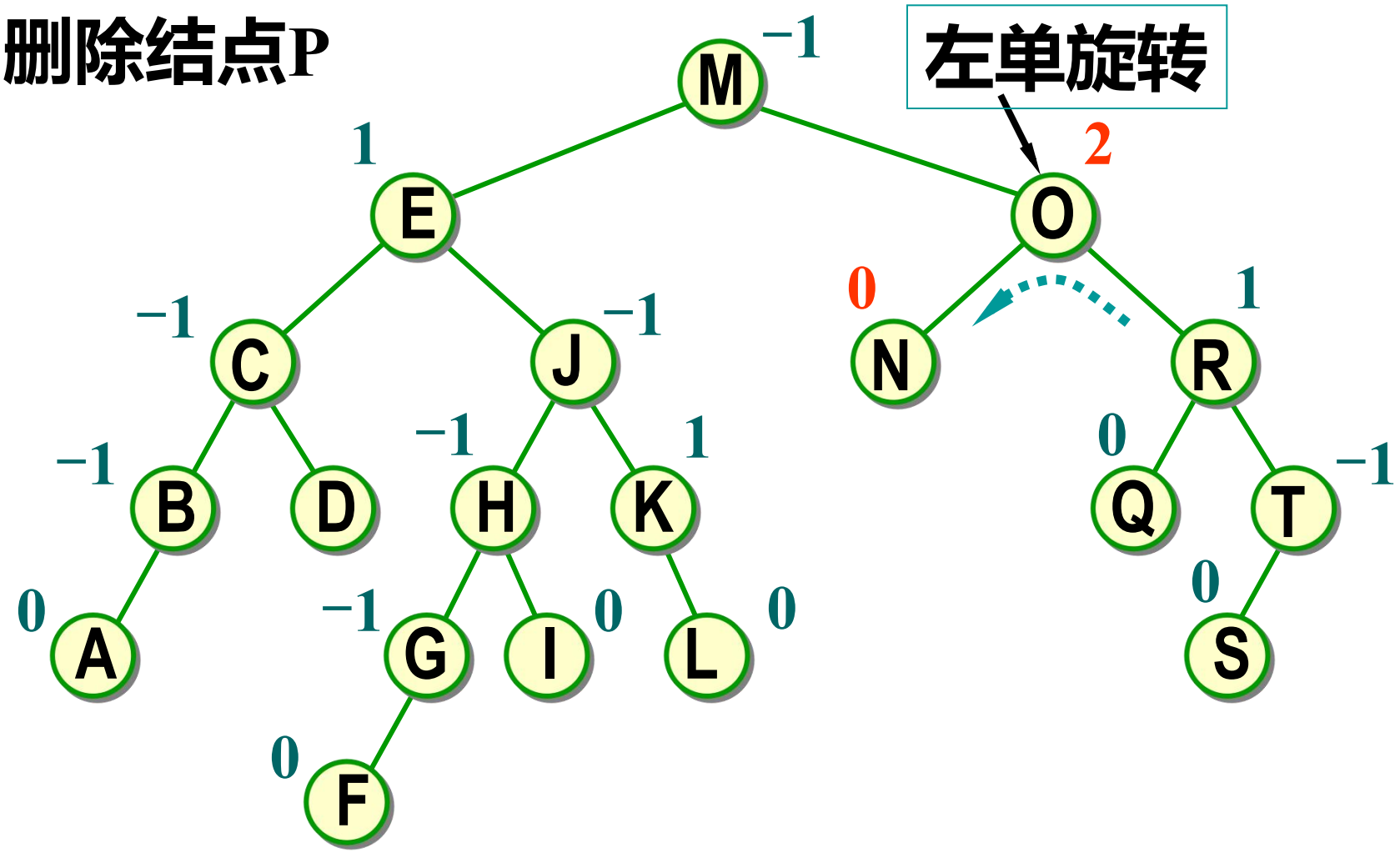
树的初始状态

删除结点P



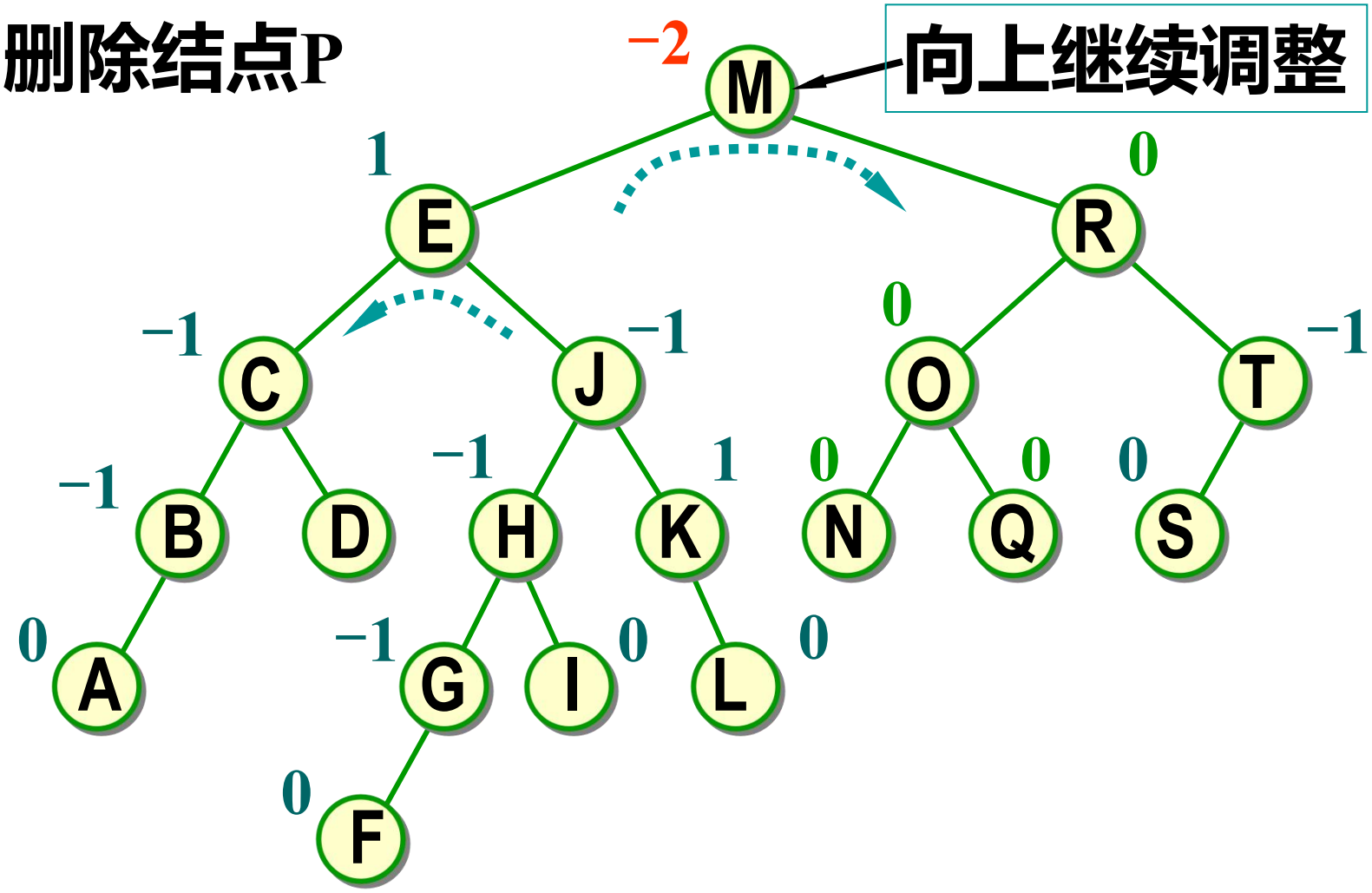
寻找结点P在中序下的直接前驱O，用O顶替P，删除O。

删除结点P



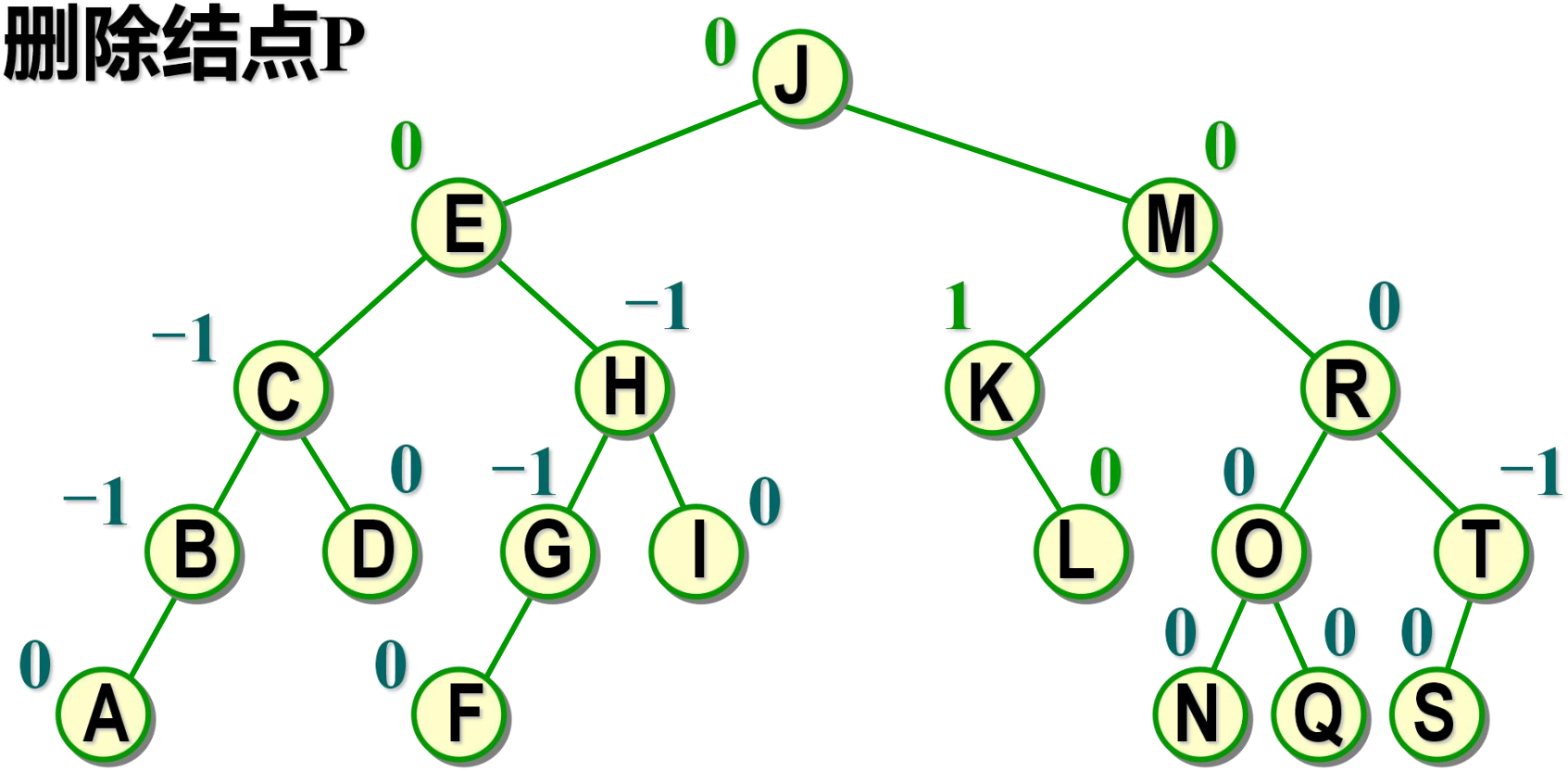
O与R的平衡因子同号，以R为旋转轴做左单旋转，M的子树高度减 1。

删除结点P



M的子树高度减 1, M发生不平衡。 M与E的平衡因子反号, 做左右双旋转。

删除结点P



6.9线段树

线段树结构是表示线段的一个几何数据结构。
高效支持关于线段集的几何运算：

```
#include <memory.h>

const int maxn=28010;

typedef struct intv
{/* 线段结点类型 */
        int low,/* 线段左端点下标 */
      high;/* 线段右端点下标 */
}Intv;
```

6.9线段树

```
typedef struct stnode *link;/* 线段树结点指针类型 */
typedef struct stnode {/* 线段树结点类型 */
    int left,/* 标准线段左端 */
        right;/* 标准线段右端 */
    int count;/* 正则覆盖计数 */
    int clq;/* 线段集最大团 */
    float uni;/* 线段集并的长度 */
}Stnode;
```

6.9线段树

```
Stnode *tree=(Stnode
*)malloc((2*maxn)*sizeof(Stnode));/* 线段树结点数组 */

int nn,/* 线段结点总数 */
    mm;/* 线段总数 */
float *xx;/* 线段结点数组 */
intv *iset;/* 线段数组 */
```

6.9线段树

```
void build(int l,int r,int pos)
{/* 建立线段树结构 */
        tree[pos].left=l;
        tree[pos].right=r;
        if(l+1==r)return;
        int mid=(l+r)/2;
        build(l,mid,pos*2);
        build(mid,r,pos*2+1);
}
```

6.9线段树

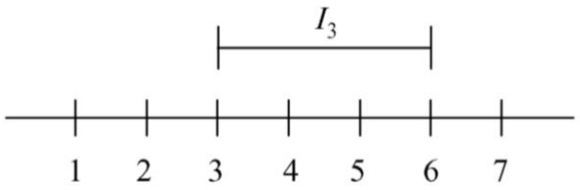
```
void outtree(int n)
{/* 输出线段树结点数组 */
   for(int i=1;i<=2*n+1;i++)
       printf("pos=%d l=%d r=%d
count=%d\n",i,tree[i].left,tree[i].right,tree[i].count);
}
```

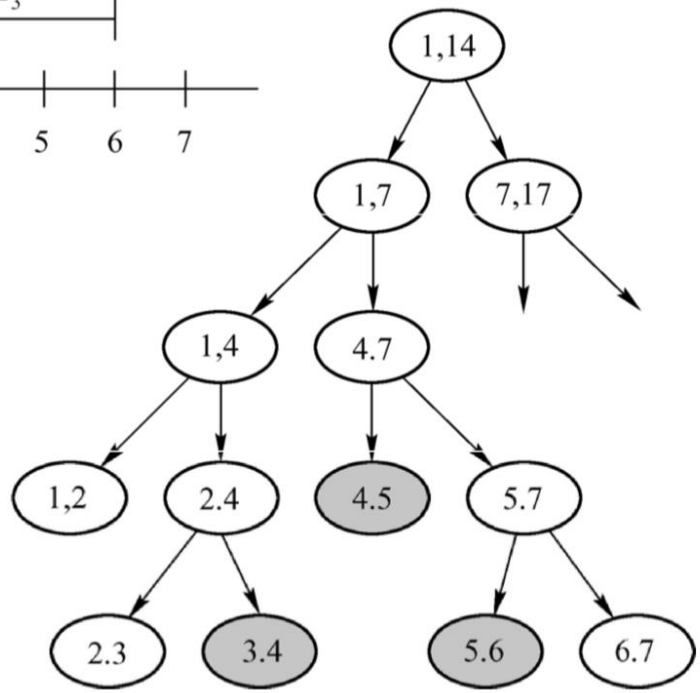
6.9线段树

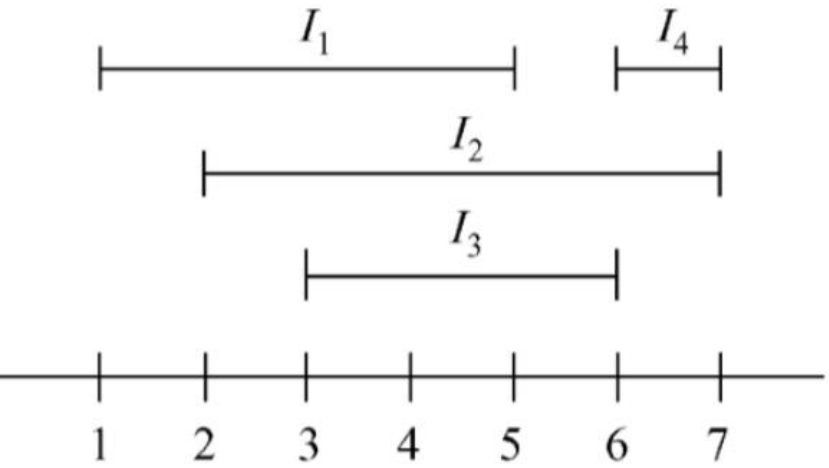
```
void change(int pos,int k)
{/* 更新结点信息 */
   /* k=1为插入，k=-1为删除 */
   tree[pos].count+=k;
}
```

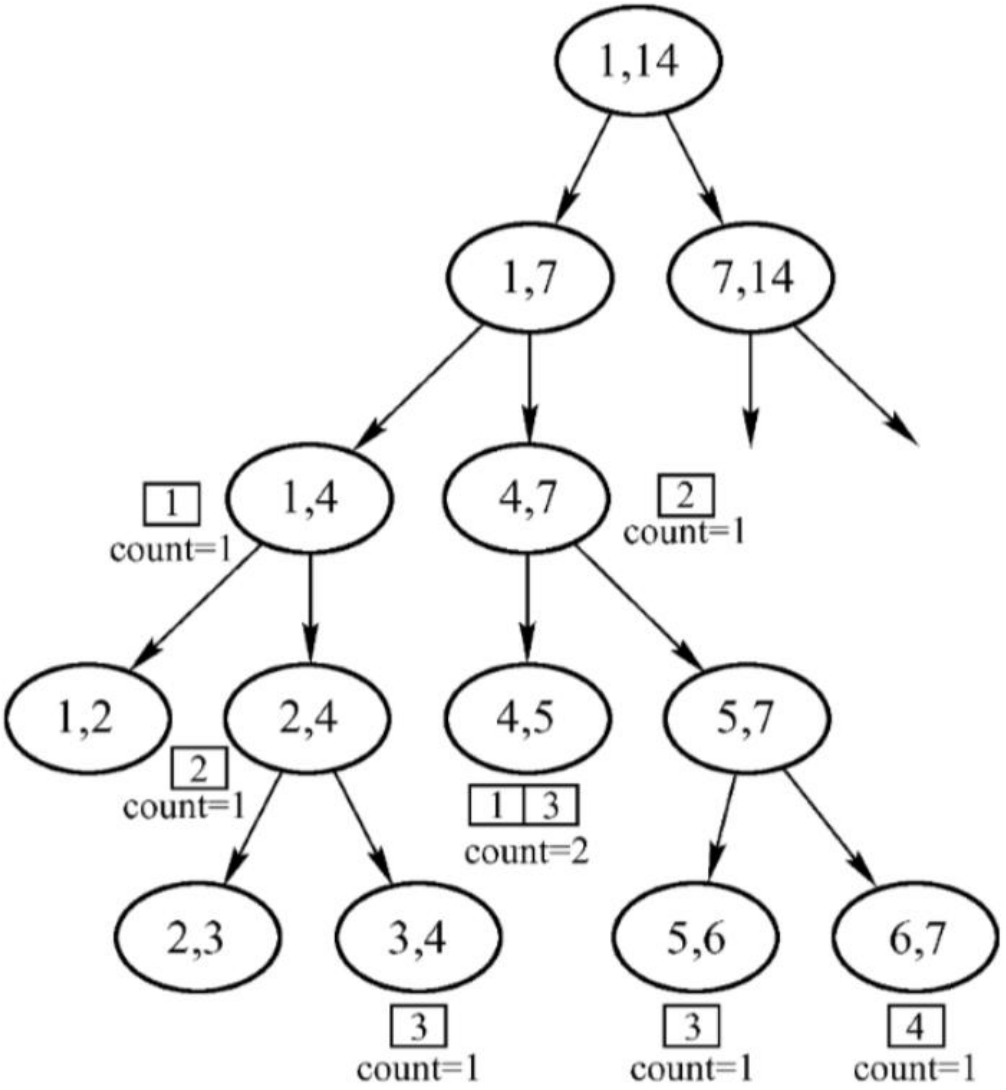
6.9线段树

```
void inst(intv r,int pos)
{/* 插入单个线段 */
   if(r.low<=tree[pos].left && tree[pos].right<=r.high)change(pos,1);
   else{
      int mid=(tree[pos].left+tree[pos].right)>>1;
      if(r.low<mid)inst(r,pos*2);
      if(r.high>mid)inst(r,pos*2+1);
   }
   update(pos);
}
```






I_1
I_4
I_2
I_3
1
2
3
4
5
6
7


1,14
1,7
7,14
1,4
4,7
1
count=1
2
count=1
1,2
2,4
4,5
5,7
2
count=1
1 3
count=2
2,3
3,4
5,6
6,7
3
count=1
3
count=1
4
count=1

6.9线段树

```
void update(int pos)
{/* 更新用户信息 */
   int l=tree[pos].left,r=tree[pos].right;
   int cnt=tree[pos].count;
   float ret=(tree[pos].count)?xx[r]-xx[l]:0;
   if(r-l<=1){
      tree[pos].clq=cnt;
      tree[pos].uni=ret;
   }
   else{
      float unil=tree[pos*2].uni,unir=tree[pos*2+1].uni;
      int clql=tree[pos*2].clq,clqr=tree[pos*2+1].clq;
      tree[pos].clq=cnt+max(clql,clqr);
      if(cnt)tree[pos].uni=ret;
      else tree[pos].uni=unil+unir;
   }
}
```

6.9线段树

```
void erase(intv r,int pos)
{/* 删除单个线段 */
   if(r.low<=tree[pos].left && tree[pos].right<=r.high)change(pos,-
1);
   else{
      int mid=(tree[pos].left+tree[pos].right)>>1;
      if(r.low<mid)erase(r,pos*2);
      if(r.high>mid)erase(r,pos*2+1);
   }
   update(pos);
}
```

6.9线段树

```
void insert()
{/* 插入线段集iset中所有线段 */
   for(int i=0;i<mm;i++)inst(iset[i],1);
}
```

6.9线段树

```
void buildst(int n,int m,float *x,intv *s)
{/* 建立线段集s的线段树 */
    nn=n,mm=m;
    xx=x;
    iset=s;
    memset(tree,0,(2*maxn)*sizeof(Stnode));
    //outtree(nn);
    build(0,n-1,1);
    insert();
}
```

6.9线段树

```
int stab(float x, int pos)
{/* 线段树穿刺计数 */
    int l=tree[pos].left,r=tree[pos].right,c=0;
    if(x>xx[l] && x<=xx[r])c+=tree[pos].count;
    if(r-l>1){
        int mid=(l+r)>>1;
        if(x<=xx[mid])c+=stab(x,pos*2);
        else c+=stab(x,pos*2+1);
    }
    return c;
}
```

6.9线段树

```
float uni1(int pos)
{/* 线段并 */
    return tree[pos].uni;
}
```

6.9线段树

```
float uni(int pos)
{/* 线段并 */
    int l=tree[pos].left,r=tree[pos].right;
    float ret=(tree[pos].count)?xx[r]-xx[l]:0;
    if(r-l<=1 || (tree[pos].count))return ret;
    else return uni(pos*2)+uni(pos*2+1);
}
```

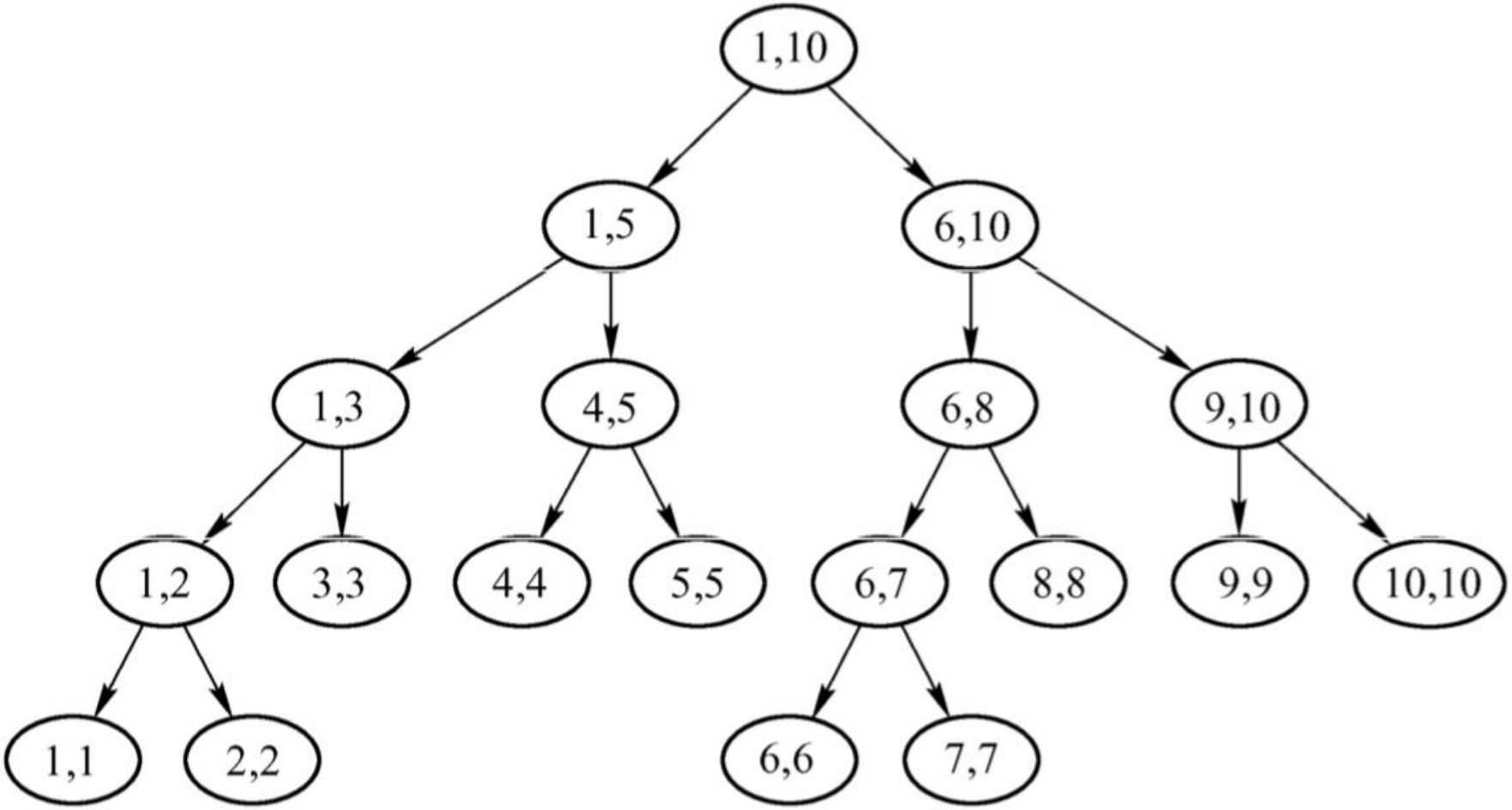
6.9线段树

```
int maxclq(int pos)
{/* 线段集最大团 */
   int l=tree[pos].left,r=tree[pos].right,cnt=tree[pos].count;
   if(r-l<=1)return cnt;
   else return cnt+max(maxclq(pos*2),maxclq(pos*2+1));
}
```

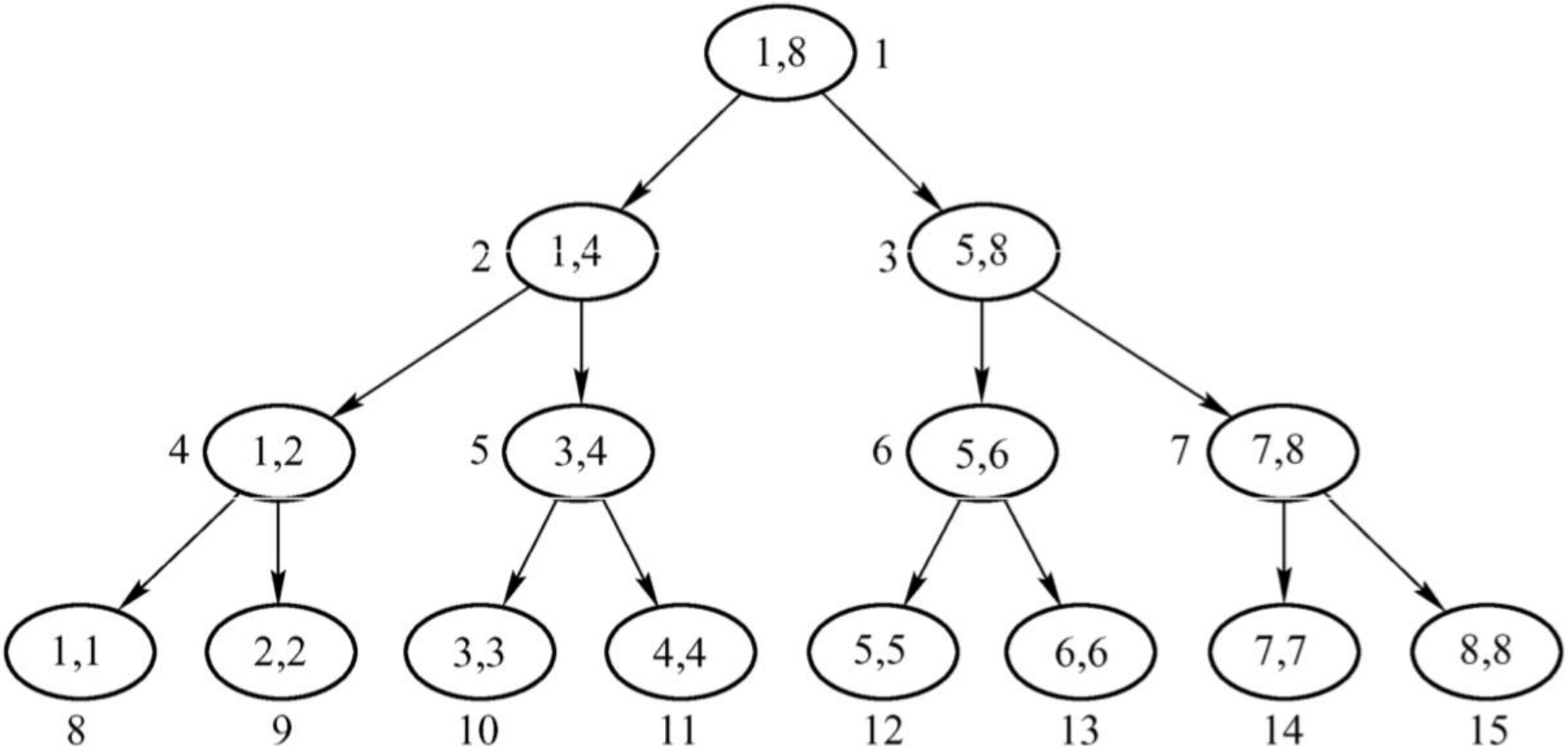
6.9线段树

```
int maxclq1(int pos)
{/* 线段集最大团 */
    return tree[pos].clq;
}
```

6.10序列树



6.10序列树


1,8
1
2
1,4
3
5,8
4
1,2
5
3,4
6
5,6
7
7,8
1,1
2,2
3,3
4,4
5,5
6,6
7,7
8,8
8
9
10
11
12
13
14
15

THE END