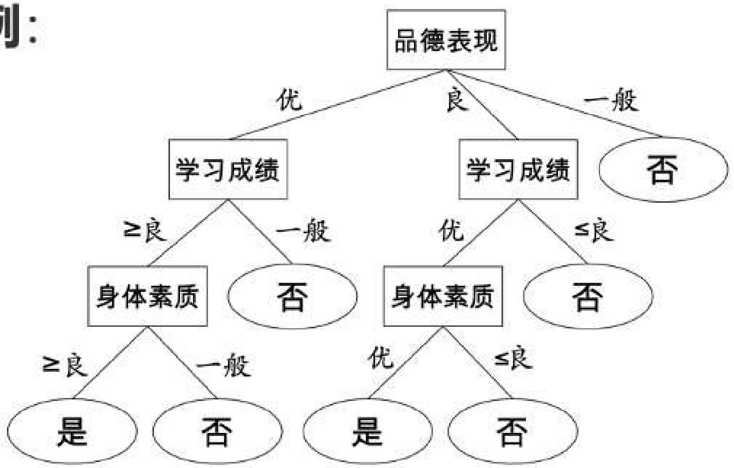
5.8 应用场景：决策树

决策树：一种解决分类问题的算法

实例：



决策树

决策树

- 每个中间结点代表一个特征属性，每个分支代表一个属性值
- 中间结点对属性值进行测试，根据判断结果决定进入下面哪个子结点
- 叶结点代表最终的决策

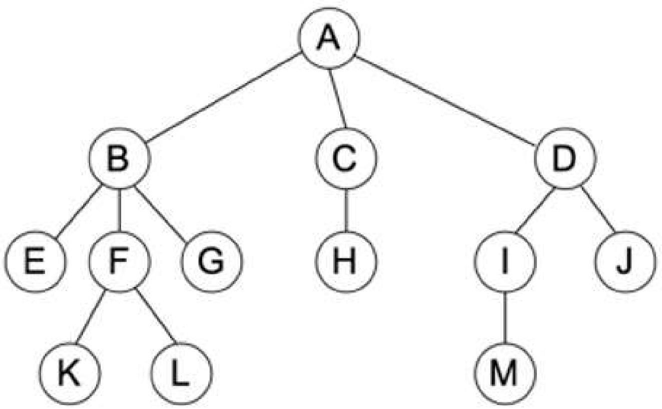
应用　对学生进行分类

- 根结点包含了所有学生，而每个中间结点包含一个学生集合（子集）
- 根据特征属性的测试结果将集合划分给各个子结点
- 每个叶结点存放一个类别，表示最终的分类结果

5.6.1 树的存储结构

与二叉树相似，树也有**顺序存储**与**链接存储**两种方式，而选择何种方式与在树结点中记录哪些表示树逻辑结构的信息相关

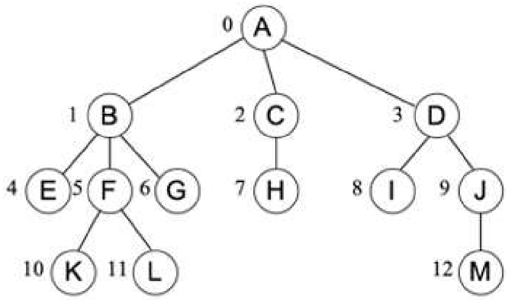
常用的树逻辑结构表示法：**父亲表示法**、**孩子表示法**以及**孩子兄弟表示法**



父亲表示法

父亲表示法要求每个结点保存父结点的位置信息，也可称为**父结点表示法**

适合用**顺序表**来存储树的所有结点



对所有结点从0开始连续编号

索引	数据	父结点
0	A	-1
1	B	0
2	C	0
3	D	0
4	E	1
5	F	1
6	G	1
7	H	2
8	I	3
9	J	3
10	K	5
11	L	5
12	M	9

可用顺序表存储树

结点数据包含两个域，一个是数据域tree[i].data记录树结点的数据元素，另一个域tree[i].parent存放父结点位置

根结点的父结点位置域是-1

高等教育出版社

算法5-17：查找父亲表示法的树的根结点FindRoot(*tree*, *x*)

输入：父亲表示法的树（顺序表）*tree*，结点（索引）*x*
输出：树*tree*的根结点索引

while $tree[x].parent \neq -1$ **do** //结点x有父结点，非根
| $x \leftarrow tree[x].parent$ // x移动至父结点
end
return x

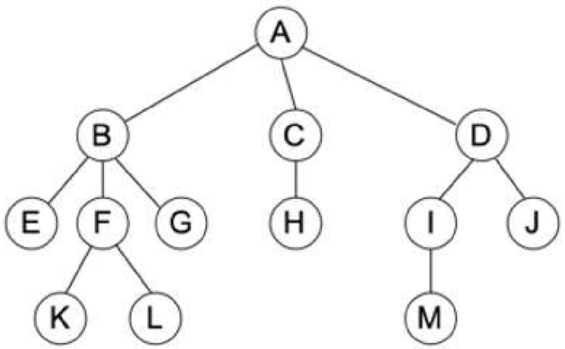
时间复杂度O(H), H表示树的高度

- 父亲表示法方便每个结点查找其祖先结点，而且每个结点只需存放父结点位置，可节省存储空间
- 父亲表示法可用于实现并查集（不相交集）
- 如果是查找结点的所有子结点或兄弟结点，父亲表示法需要对整个树进行遍历，时间效率低

父亲表示法的应用：最近公共祖先问题

问题描述：在基于父亲表示法的树中，查找两个结点的最近公共祖先LCA。结点x和结点y的最近公共祖先的定义如下：

- 如果x是y的祖先结点，则x就是LCA；同样，如果y是x的祖先结点，则y是LCA
- 如果x和y没有祖孙关系，则x和y一定有相同的祖先结点(?)，在所有共同祖先中，层数最大的结点是x和y的LCA



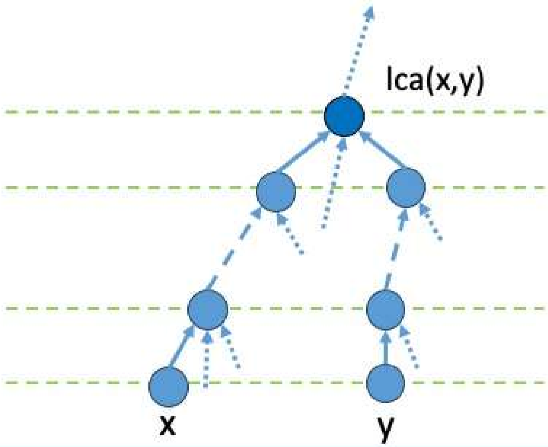
- LCA(D, M) = D
- LCA(F, H) = A
- LCA(E, K) = B

父亲表示法的应用：最近公共祖先问题

问题描述：在基于父亲表示法的树中，查找两个结点的最近公共祖先LCA。结点x和结点y的最近公共祖先的定义如下：

- 如果x是y的祖先结点，则x就是LCA；同样，如果y是x的祖先结点，则y是LCA
- 如果x和y没有祖孙关系，则x和y一定有相同的祖先结点(?)，在所有共同祖先中，层数最大的结点是x和y的LCA

情形一: 结点x和结点y在树的同一层上, 即 level(x) = level(y)



- x和y到 lca(x,y)的路径长度（分支数）相同

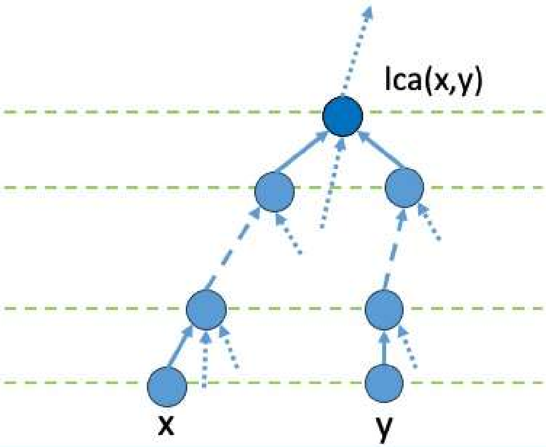
- x和y同步向上移动，直到x与y相等为止，即LCA

父亲表示法的应用：最近公共祖先问题

问题描述：在基于父亲表示法的树中，查找两个结点的最近公共祖先LCA。结点x和结点y的最近公共祖先的定义如下：

- 如果x是y的祖先结点，则x就是LCA；同样，如果y是x的祖先结点，则y是LCA
- 如果x和y没有祖孙关系，则x和y一定有相同的祖先结点(?)，在所有共同祖先中，层数最大的结点是x和y的LCA

情形一：结点x和结点y在树的同一层上，即 level(x) = level(y)




```
1. while x ≠ y do
2. |   x ← tree[x].parent  //x和y同步向上移动
3. |   y ← tree[y].parent
4. end
5. return x  //返回结点x和结点y的最近公共祖先
```

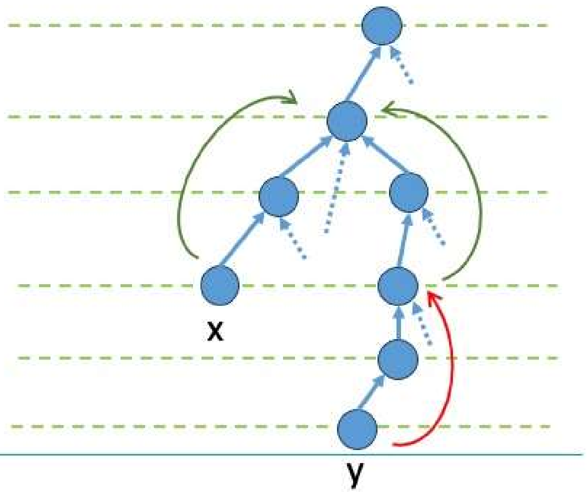
高等教育出版社

父亲表示法的应用：最近公共祖先问题

问题描述：在基于父亲表示法的树中，查找两个结点的最近公共祖先LCA。结点x和结点y的最近公共祖先的定义如下：

- 如果x是y的祖先结点，则x就是LCA；同样，如果y是x的祖先结点，则y是LCA
- 如果x和y没有祖孙关系，则x和y一定有相同的祖先结点(?)，在所有共同祖先中，层数最大的结点是x和y的LCA

情形二: 结点x和结点y不在树的同一层上, 假设 level(x) < level(y)



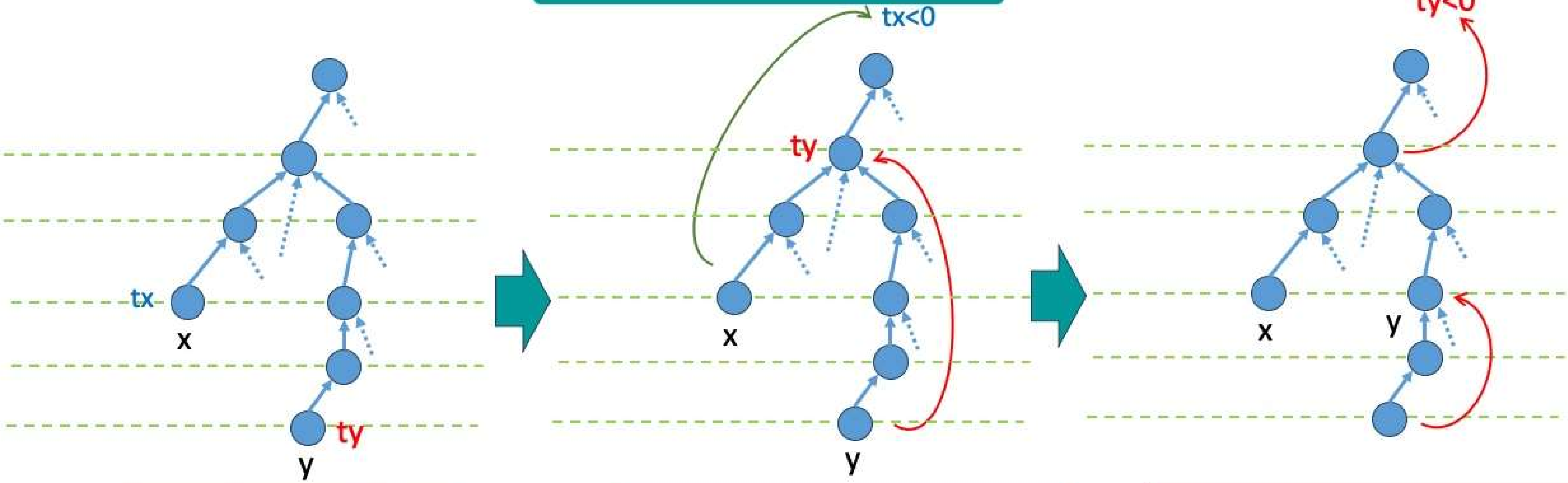
➢ 先把y向上移到与x同一层

➢ 再让x和y同步向上移动

父亲表示法的应用：最近公共祖先问题

情形二: 结点x和结点y不在树的同一层上, 假设 level(x) < level(y)

如何把y向上移到与x同一层?



$tx \leftarrow x$; $ty \leftarrow y$

tx和ty同步上移，直到tx<0

y和ty同步上移，直到ty<0

父亲表示法的应用：最近公共祖先问题

问题描述：在基于父亲表示法的树中，查找两个结点的最近公共祖先LCA。结点x和结点y的最近公共祖先的定义如下：

- 如果x是y的祖先结点，则x就是LCA；同样，如果y是x的祖先结点，则y是LCA
- 如果x和y没有祖孙关系，则x和y一定有相同的祖先结点(?)，在所有共同祖先中，层数最大的结点是x和y的LCA

```
tx ← x
ty ← y
while tx ≥ 0 且 ty ≥ 0 do
|  tx ← tree[tx].parent      //tx和ty同步向上移动
|  ty ← tree[ty].parent
end
if ty < 0 then      //level(x) ≥ level(y) 交换x和y
|  x ↔ y
|  tx ↔ ty
end
while ty ≥ 0 do
|  ty ← tree[ty].parent   //y和ty同步向上移动
|  y ← tree[y].parent
end
while x ≠ y do         //x和y在同一层
|  x ← tree[x].parent  //x和y同步向上移动
|  y ← tree[y].parent
end
return x  //返回结点x和结点y的最近公共祖先
```

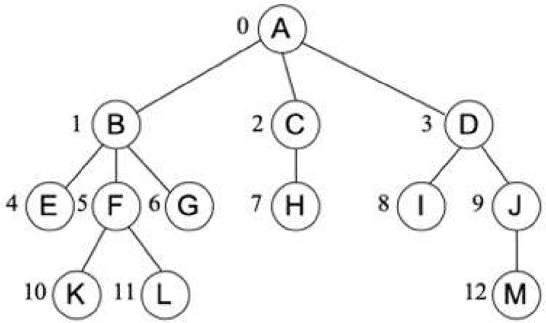
思考题1：如果x是y的祖先结点，会发生什么情况?

思考题2：如何在链接存储结构的二叉树中求最近公共祖先?

孩子表示法

与父亲表示法一样，用**顺序表**存储树，每个结点包含数据域，父结点位置域，以及**子结点链表域**，用来存放指向单链表的指针

链表中各子结点按从左向右的顺序排列



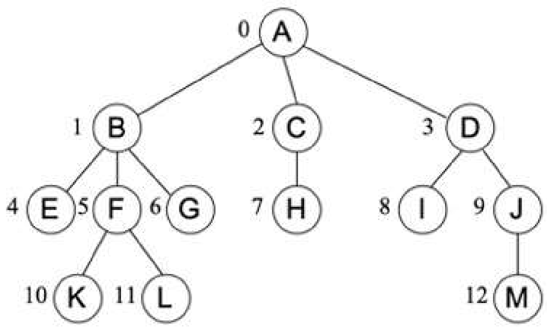
索引	数据	父结点	子结点链表
0	A	-1	1 → 2 → 3
1	B	0	4 → 5 → 6
2	C	0	7
3	D	0	8 → 9
4	E	1	
5	F	1	10 → 11
6	G	1	
7	H	2	
8	I	3	
9	J	3	12
10	K	5	
11	L	5	
12	M	9	

孩子表示法

与父亲表示法一样，用**顺序表**存储树，每个结点包含数据域，父结点位置域，以及**子结点链表域**，用来存放指向单链表的指针

第一个孩子：各结点在树中最左边的子结点

下一个兄弟：各结点右侧并且相邻的兄弟结点



索引	数据	父结点	子结点链表
0	A	-1	→ 1 → 2 → 3
1	B	0	→ 4 → 5 → 6
2	C	0	→ 7
3	D	0	→ 8 → 9
4	E	1	
5	F	1	→ 10 → 11
6	G	1	
7	H	2	
8	I	3	
9	J	3	→ 12
10	K	5	
11	L	5	
12	M	9	

各结点的第一个孩子就是其子结点链表的第一个结点，查找时间O(1)

各结点的下一个兄弟需要遍历其父结点的子结点链表，时间复杂度O(d), d是树的度

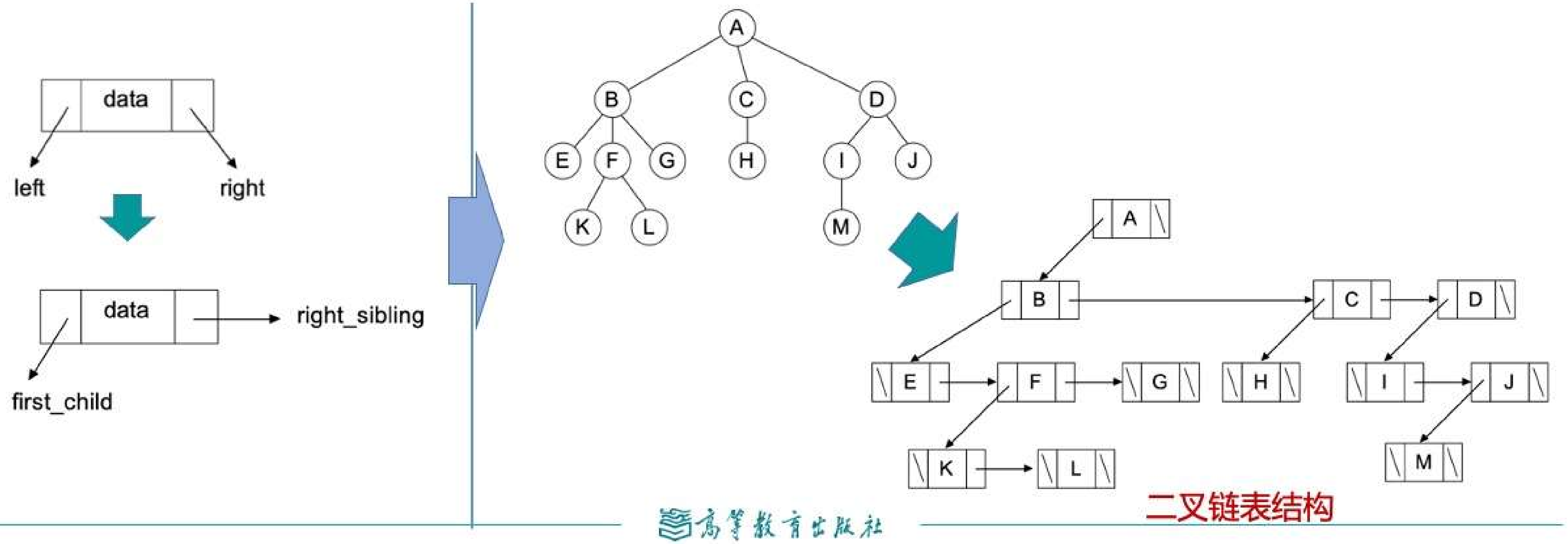
结点B的第一个孩子: E，下一个兄弟：C

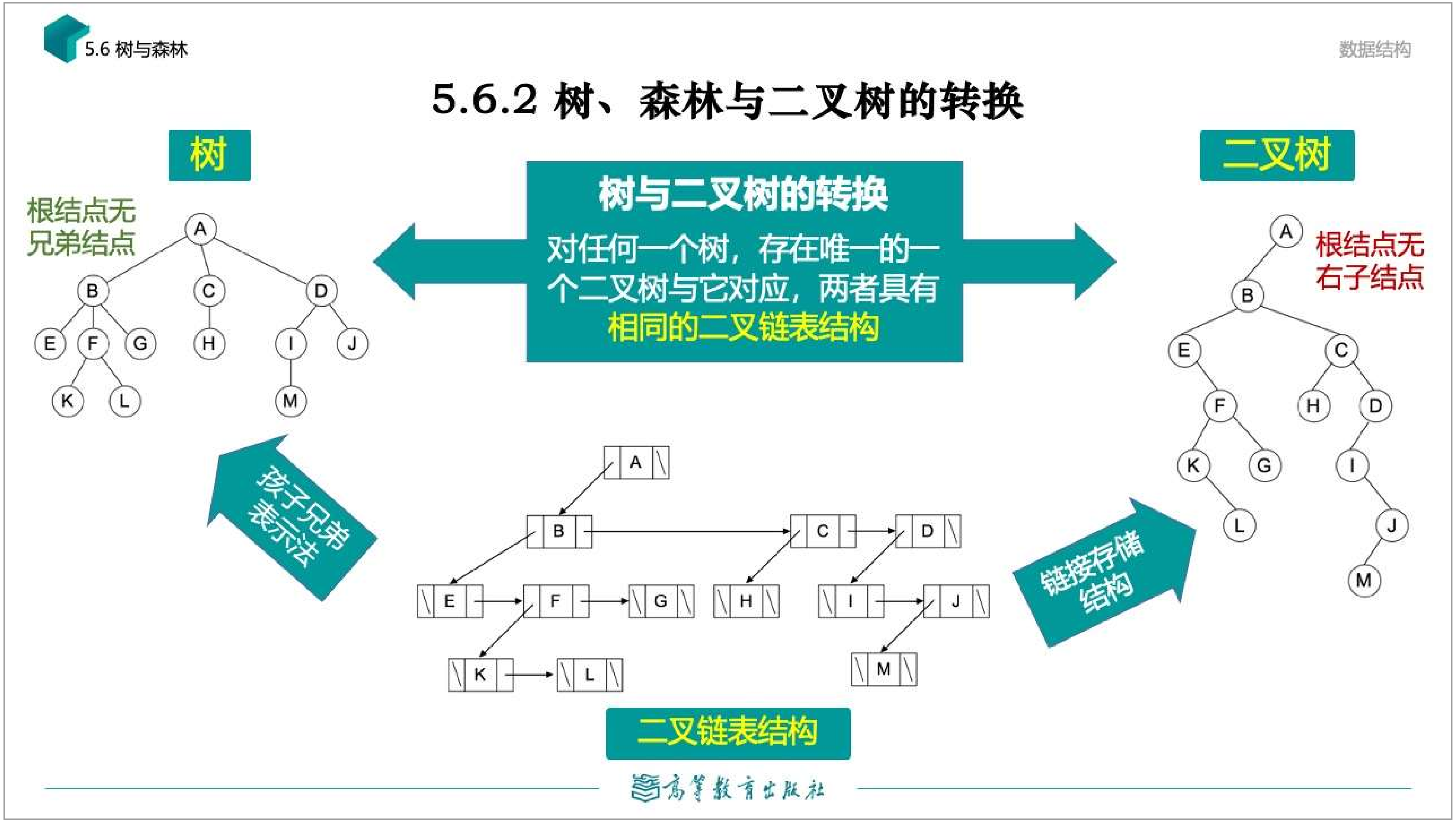
高等教育出版社

孩子兄弟表示法

树应用最广泛的存储结构。每个结点存放其**第一个孩子**和**下一个兄弟**的信息，可以直接使用二叉链表实现，因此这种表示法也称作**二叉链表表示法**

二叉链表中的指针域left 改称 first_child，指向结点的第一个子结点；同时，指针域right 改称next_sibling，指向右侧的兄弟结点




5.6 树与森林
数据结构
5.6.2 树、森林与二叉树的转换
树
根结点无兄弟结点
树与二叉树的转换
对任何一个树，存在唯一的一个二叉树与它对应，两者具有相同的二叉链表结构
二叉树
根结点无右子结点
孩子兄弟表示法
链接存储结构
二叉链表结构
高等教育出版社

5-6 树与森林

算法5-18：查找树中带有指定数据的结点 Search(*tree*, *x*)

输入：基于孩子兄弟表示法的*tree*，结点元素 *x*

输出：如果树中有数据域等于*x*的结点，返回该结点；否则，返回NIL

```
node_ptr ← tree
if node_ptr ≠ NIL then
| if node_ptr.data ≠ x then
| | node_ptr ← Search(tree.first_child, x)      //在子孙结点中查找
| | if node_ptr = NIL then                       //不在子孙结点中
| | | node_ptr ← Search(tree.next_sibling, x)  //在兄弟结点及其子孙中找
| | end
| end
end
return  node_ptr
```

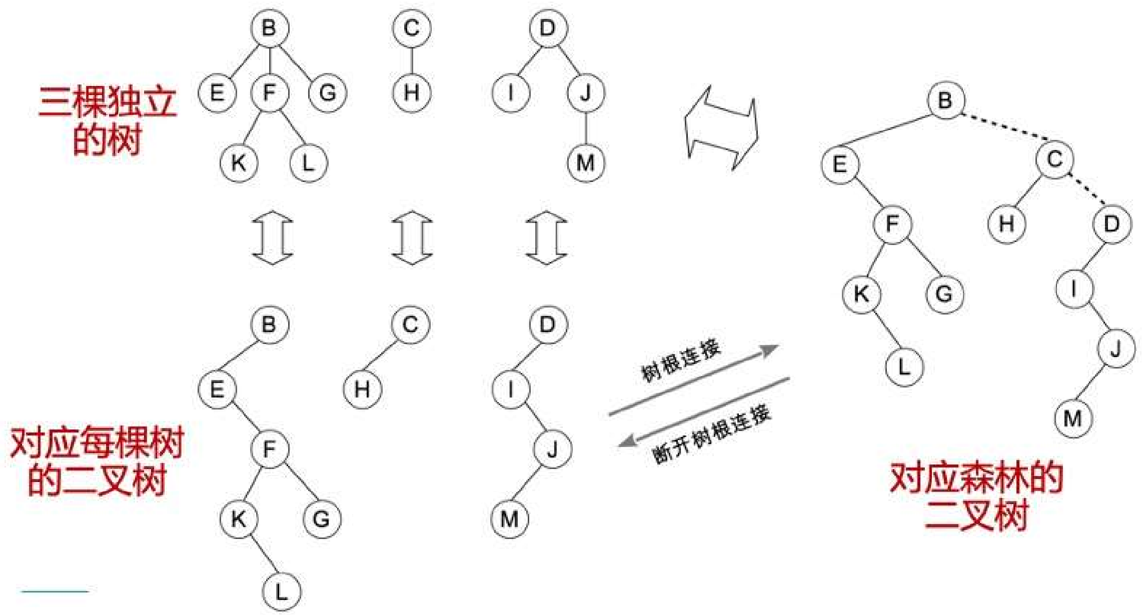
采用二叉树的前序遍历算法

时间复杂度O(n), n是树中结点数目

森林与二叉树的转换

对于每一棵独立的树，由于根结点没有兄弟，它对应的二叉树的根没有右子结点，即**右子树为空**

利用右子树的链将树串联起来，建立**森林与二叉树的对应关系**



森林转换成二叉树

(1) 把森林中的每个树转换为二叉树

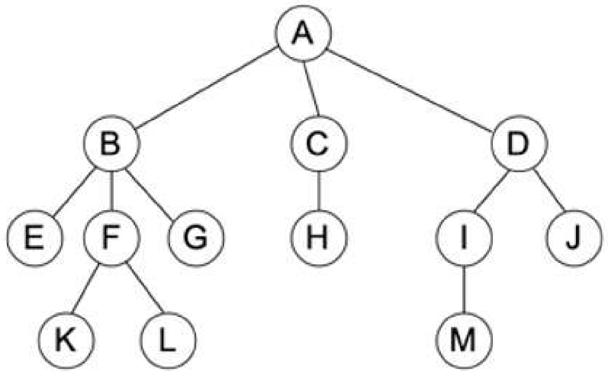
(2) 把森林中第一个二叉树的根结点作为转换后的二叉树的根，从第二个二叉树开始，把每个二叉树的根作为前一个二叉树的根的右子结点

5.6.3 树与森林的遍历

树的遍历方案：前序遍历、后序遍历（无中序遍历！）

前序遍历：先访问树根，然后对根的各子树从左向右依次进行前序遍历

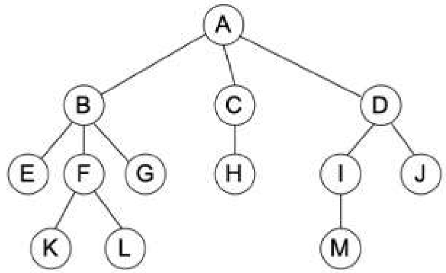
后序遍历：遍历根的各子树，最后访问根结点



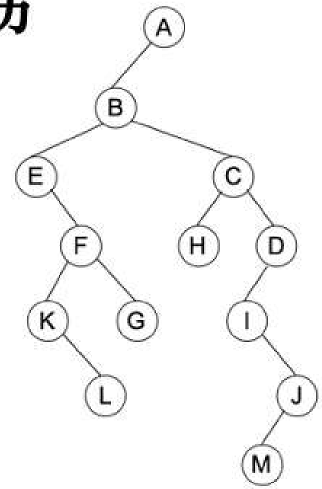
前序遍历：<A, B, E, F, K, L, G, C, H, D, I, J, M>

后序遍历：<E, K, L, F, G, B, H, C, I, M, J, D, A>

树与对应的二叉树的遍历



对应



前序 <A, B, E, F, K, L, G, C, H, D, I, J, M> = <A, B, E, F, K, L, G, C, H, D, I, J, M> 前序

树的前序遍历与对应的二叉树的前序遍历结果相同

后序 <E, K, L, F, G, B, H, C, I, M, J, D, A> = <E, K, L, F, G, B, H, C, I, M, J, D, A> 中序

树的后序遍历与对应的二叉树的中序遍历结果相同

基于二叉树遍历方案的树遍历算法

二叉树的前序遍历算法

算法5-19. 前序遍历树 PreOrder(*tree*)
输入：基于孩子兄弟表示法存储的树*tree*（二叉链表结构） 输出：按前序遍历的顺序依次访问各结点

if *tree* ≠ NIL **then** //空树不做处理，直接返回
| Visit(*tree*) //先访问根结点
| PreOrder(*tree.first_child*) //接下来访问*tree*所有子孙结点
| PreOrder(*tree.next_sibling*) //最后访问*tree*后序的兄弟结点及其子孙
end

二叉树的中序遍历算法

算法5-20. 后序遍历树 PostOrder(*tree*)
输入：基于孩子兄弟表示法存储的树*tree*（二叉链表结构） 输出：按后序遍历的顺序依次访问各结点

if *tree* ≠ NIL **then** //空树不做处理，直接返回
| PostOrder(*tree.first_child*) //先访问*tree*所有子孙结点
| Visit(*tree*) //接下来访问根结点
| PostOrder(*tree.next_sibling*) //最后访问*tree*后序的兄弟结点及其子孙
end

单选题 1分

层序遍历基于孩子兄弟表示法的树，设队列先头取出的结点node非空，下面的操作中正确的是

A

```
node ← node.first_child
while node ≠ NIL do
|   EnQueue(queue, node)
|   node ← node.next_sibling
end
```

B

```
EnQueue(queue, node.next_sibling)
EnQueue(queue, node.first_child)
```

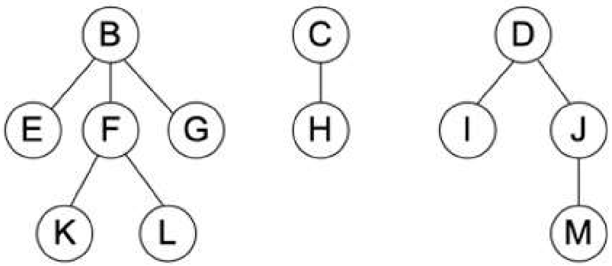
C

```
t ← node.next_sibling
while t ≠ NIL do
|   EnQueue(queue, t)
|   t ← t.next_sibling
end
EnQueue(queue, node.first_child)
```

森林的遍历

前序遍历：从其中的第一个树开始，按序对每个树进行前序遍历

后序遍历：从其中的第一个树开始，按序对每个树进行后序遍历

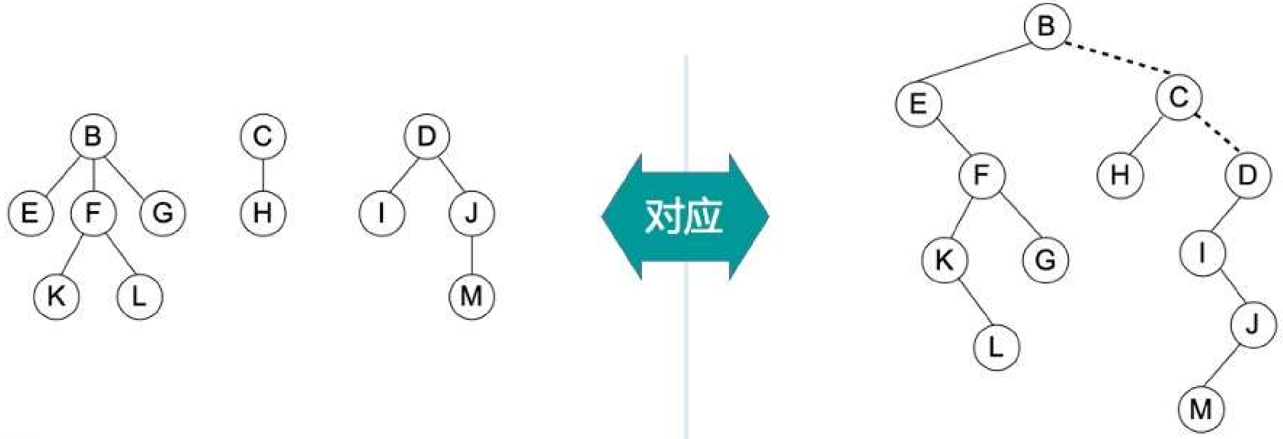


前序遍历：< B, E, F, K, L, G, C, H, D, I, J, M >

后序遍历：< E, K, L, F, G, B, H, C, I, M, J, D >

从左向右依次遍历每棵树！

森林与对应的二叉树的遍历



前序 <B, E, F, K, L, G, C, H, D, I, J, M> = <B, E, F, K, L, G, C, H, D, I, J, M> 前序

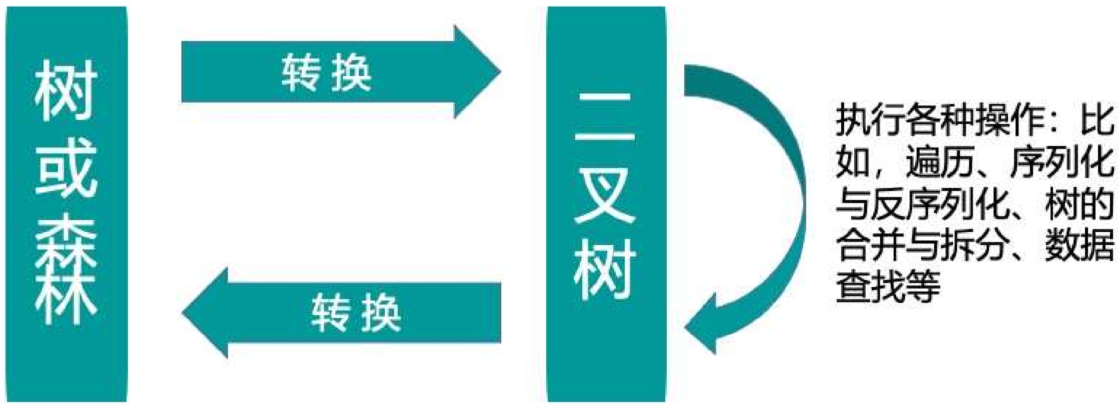
森林的前序遍历与对应的二叉树的前序遍历结果相同　算法5.19可用

后序 <E, K, L, F, G, B, H, C, I, M, J, D> = <E, K, L, F, G, B, H, C, I, M, J, D> 中序

森林的后序遍历与对应的二叉树的中序遍历结果相同　算法5.20可用

5.6.2 树、森林与二叉树的转换

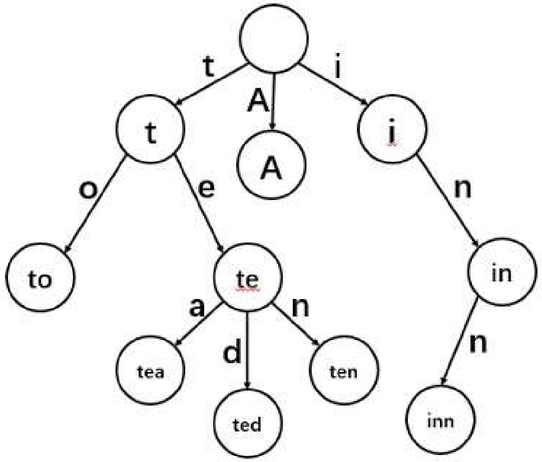
树、森林与二叉树的对应关系表明可以把树或森林先转换为二叉树，使用二叉树的各种操作进行处理，处理结束后还可以再转换回原来的树或森林



5.7.1 拓展延伸：前缀树

前缀树：又名Trie树、字典树、单词查找树，是专门处理**字符串匹配**的树形结构

典型应用：可以存储大量的字符串并从中快速查找指定的字符串，所以经常被搜索引擎系统用于**文本词频统计**



前缀树

- 逻辑结构上，前缀树是一棵k叉树，k通常等于构成字符串的字符集规模
- 结点的每个分支对应字符集中唯一的一个字符
- 从根到各结点的路径，路径经过的分支序代表结点对应的字符串
- 每个结点对应的字符串不同，因此前缀树把所有字符串的共同前缀合并在一条路径上表示，从而最大限度地减少多余的字符串比较

5.8 应用场景：决策树

决策树：一种解决分类问题的算法

实例：

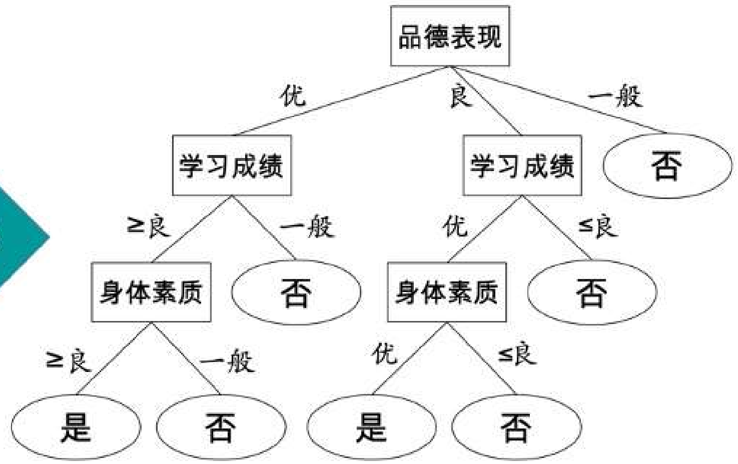
某高校的优秀学生选拔标准

1）学生在德、智、体三个方面都取得良以上的成绩

2）按照“为学须先立志”的原则，品行优异的学生可评为优秀

3）对于品德表现良好的学生，必须“文武全才”，即学习成绩和身体素质皆优

转换



决策树

5.8 应用场景：决策树

决策树：一种解决分类问题的算法

决策树的构建：给定一组训练数据，每个数据包含多个特征属性并且带有表示类别的标记。从训练数据集中归纳出一组分类规则，并由此构造一个决策树，使它能够对训练数据执行正确的分类, 也可用于预测新数据的类别。

决策树构建算法：经典的决策树生成算法有ID3、C4.5与CART，其中ID3是最早提出的机器学习算法

ID3算法是一种贪心法，其核心是“信息熵”。假设样本数据集C包含k个独立的类别，每个类别构成C的一个子集C_j（$1 \le j \le k$），则数据集C的总信息熵为：

$$H(C) = -\sum_{j=1}^{k} \frac{|C_j|}{|C|} \log \frac{|C_j|}{|C|}$$

ID3算法

ID3算法流程:

(1) 依次计算各特征属性的信息增益度，如果没有特征可选或信息增益量小于阈值，结束计算；否则执行(2)的操作

(2) 选择信息增益度最大的特征属性作为决策树结点，对特征值区间进行划分并建立子结点，每个子结点对应不同的特征值

(3) 把数据集按特征值划分给每个子结点

(4) 对各子结点重复执行(1)的操作。

信息增益

设特征属性A的取值为$\{a_1, a_2, \dots, a_m\}$ $(m \geq 1)$。用D_{a_i}表示数据集C中属性A取值a_i $(1 \leq i \leq m)$的所有数据集合，对D_{a_i}按类别标记进行分类，可得信息熵：

$$H(D_{a_i}) = -\sum_{j=1}^{k} \frac{|D_{a_i} \cap C_j|}{|D_{a_i}|} \log \frac{|D_{a_i} \cap C_j|}{|D_{a_i}|}$$

由此计算属性A对数据集C的条件熵：

$$H(C,A) = \sum_{i=1}^{m} \frac{|D_{a_i}|}{|C|} H(D_{a_i})$$

信息增益定义为$H(C)$与$H(C,A)$的差值，即

$$Gain(A) = H(C) - H(C, A)$$

$Gain(A)$表示对数据集先按特征属性A进行划分后，对判断任意数据属于哪个类别所需信息量的减少程度。

高等教育出版社

ID3算法

ID3算法实例

学生的成绩单及分类

学生ID	品德表现	学习成绩	身体素质	是否优秀
1	优	一般	良	否
2	优	良	良	是
3	良	优	一般	否
4	良	优	优	是
5	良	良	优	否
6	优	优	良	是
7	一般	一般	一般	否
8	一般	优	优	否
9	一般	良	一般	否
10	优	优	优	是

全体学生S中有4名优异生、6名非优异生，总信息熵

$$H(S)=-\frac{4}{10}\times\log(\frac{4}{10})-\frac{6}{10}\times\log(\frac{6}{10})=0.971$$

按品德表现划分学生，把学生分成三组，即品行优异组、品行良好组以及品行一般组，各组的信息熵为：

$$H(T_{优})=-\frac{3}{4}\times\log(\frac{3}{4})-\frac{1}{4}\times\log(\frac{1}{4})=0.811$$

$$H(T_{良})=-\frac{1}{3}\times\log(\frac{1}{3})-\frac{2}{3}\times\log(\frac{2}{3})=0.918$$

$$H(T_{一般})=-\frac{0}{3}\times\log(\frac{0}{3})-\frac{3}{3}\times\log(\frac{3}{3})=0.0$$

根据上面三个信息熵，可以算出按照品德表现进行划分后，S的条件熵

$$H(S,T)=\frac{4}{10}\times H(T_{优})+\frac{3}{10}\times H(T_{良})+\frac{3}{10}\times H(T_{一般})=0.6$$

同理，按学习成绩和身体素质划分后，S的条件熵分别是0.761和 0.675。由此可见，**品德表现的信息增益度最大**

因此，ID3算法选择品德表现这一特征属性作为决策树的根结点，其三个子结点分别对应$T_{优}$、$T_{良}$以及$T_{一般}$这三个子集；然后对各子结点继续进行拆分。

5.8 应用场景：决策树

决策树：一种解决分类问题的算法

决策树构建算法：经典的决策树生成算法有ID3、C4.5与CART，其中ID3是最早提出的机器学习算法

- ID3算法只能处理离散型特征，同时信息增益倾向于选择取值较多的属性。
- 针对ID3算法的缺陷，C4.5算法引入信息增益率来作为分类标准，能够处理连续数值型特征属性，同时在决策树构造过程中进行剪枝
- 与C4.5算法相比，CART算法采用了简化的二叉树模型，同时特征选择采用了近似的基尼系数来简化计算，该算法还可用于回归

总结：决策树算法是机器学习中常用的模型，易于理解，可解释性强，可以同时处理数值型和非数值型数据，能够处理关联度低的特征属性，符合人类的直观思维。但容易发生过拟合的现象，容易忽略特征属性之间的关联，并且预测精度易受异常数据的影响。

5.9 小结

本章是数据结构的重点之一，也是本书许多后续章节的基础：

- 树与二叉树的基本概念
- 二叉树的存储方式和运算实现
- 二叉树的两种应用：表达式树和哈夫曼树
- 普通的树形结构以及森林的表示法、存储结构
- 普通的树形结构、森林与二叉树的转换，以及基于二叉树的遍历方法
- 拓展延伸部分包含前缀树和后缀树
- 以决策树作为应用场景，包括基于信息熵构建决策树的ID3算法


谢谢观看