第5章 树与二叉树（下）

郑新、徐鹏飞、李健
北京师范大学人工智能学院
2025-2026学年 第一学期

考核要点

- **考核大纲**
  - Huffman树与编码：前缀编码、Huffman算法及优化
  - 树与森林：树的存储实现，树的遍历，树、森林与二叉树的转换
  - 二叉树拓展：二叉树的中序线索化和遍历

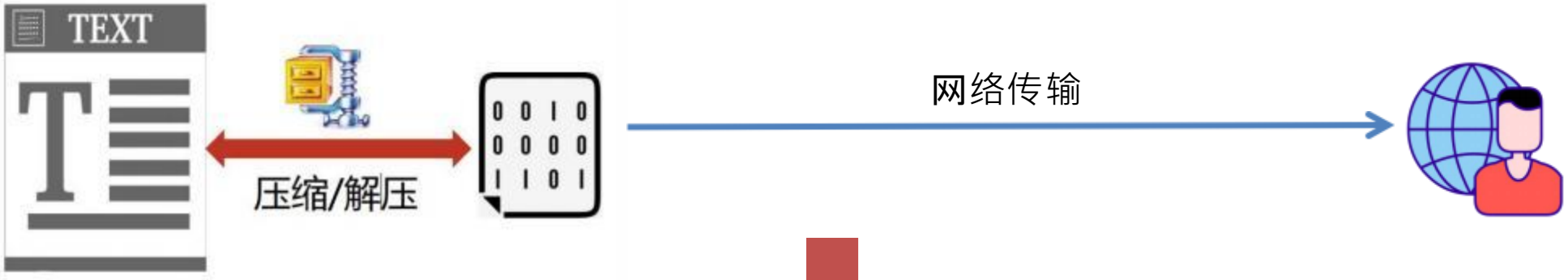
- **复习要点**
  - 灵活运用二叉树解决数据压缩问题（B）
  - 理解树与森林（A）；熟练掌握树、森林与二叉树的转换（A）
  - 了解线索化和线索树的遍历（B）

提纲

- 5.5 Huffman树与Huffman编码
- 5.6 树与森林
- 5.7 扩展延伸
- 5.8 应用场景：决策树

问题引入：文件传输

- 如何使用更少的编码表示信息？

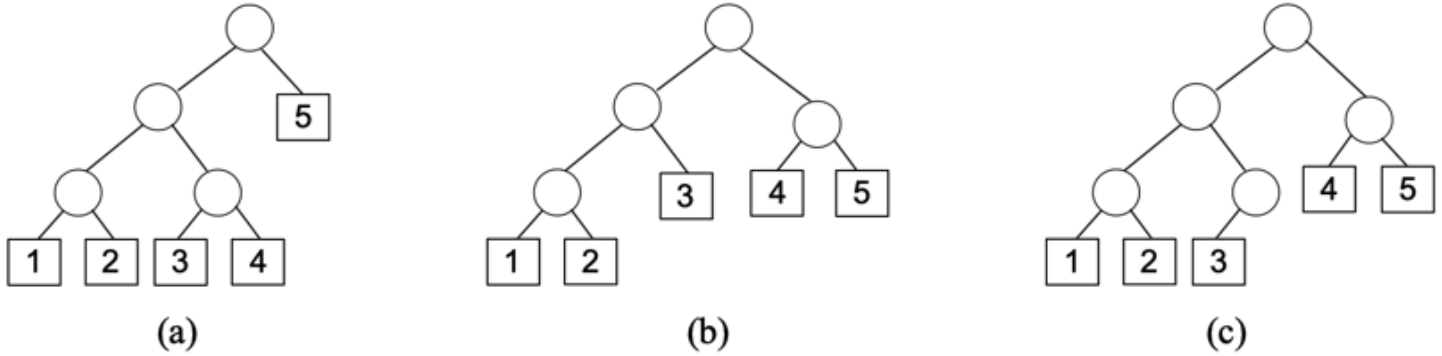


如何有效地实现对文本/数据的压缩和编码？

使用尽量少的编码完成字符的转换（提高存储和传输效率）
对已经编码的数据能够实现无损解码

5.5.1 Huffman树

- **带权二叉树：每个叶结点都带有一个正数权重二叉树**



带有相同权重集 {1, 2, 3, 4 ,5} 的三棵带权二叉树

- **结点的路径长度：结点的路径长度是指从根结点到该结点的路径上所包含的边的数目。**
  - 图(a) 中叶结点路径长度：3、3、3、3、1
  - 图(b) 中叶结点路径长度：3、3、2、2、2
  - 图(c) 中叶结点路径长度：3、3、3、2、2

5.5.1 Huffman树

- **叶结点的带权路径长度**：叶结点权重*叶结点路径长度

$$w_i l_i$$

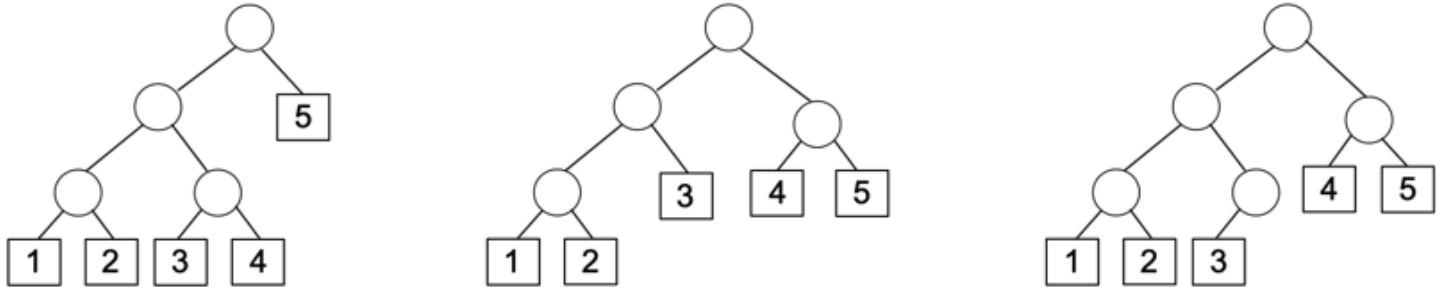
  - w_i是叶结点的权重
  - l_i是从根到叶结点的路径长度，即从根到该结点经过的边数
- **树的带权路径长度：**树中所有叶结点的带权路径长度之和

$$\mathrm{WPL} = \sum_{i=1}^{n} w_i l_i$$

  - n为叶结点个数，$i \in \{1, \cdots, n\}$
  - w_i是第i个叶结点的权重
  - l_i是第i个叶结点的路径长度

5.5.1 Huffman树

- **带权二叉树**：每个叶结点都带有一个正数权重二叉树



带有相同权重集 {1, 2, 3, 4 ,5} 的三棵带权二叉树

$$WPL(a) = 1 \times 3 + 2 \times 3 + 3 \times 3 + 4 \times 3 + 5 \times 3 = 36$$

$$WPL(b) = 1 \times 3 + 2 \times 3 + 3 \times 2 + 4 \times 2 + 5 \times 2 = 33$$

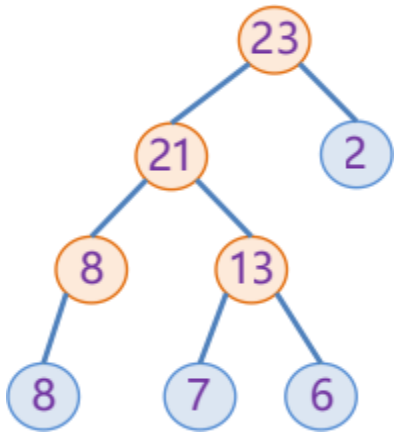
$$WPL(c) = 1 \times 3 + 2 \times 3 + 3 \times 3 + 4 \times 2 + 5 \times 2 = 35$$

最小值

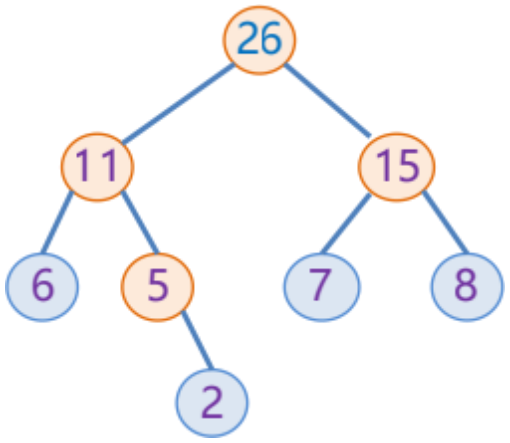
- **Huffman树**：给定一组叶结点权重，由此构建的所有带权二叉树中，带权路径长度最小的二叉树，亦称**最优二叉树**

5.5.1 Huffman树

- 例： 试比较以下两棵树的加权路径长度。



T1



T2

$WPL_1 = 8\times3+7\times3+6\times3+2\times1 = 65$

$WPL_2 = 6\times2+2\times3+7\times2+8\times2 = 48$

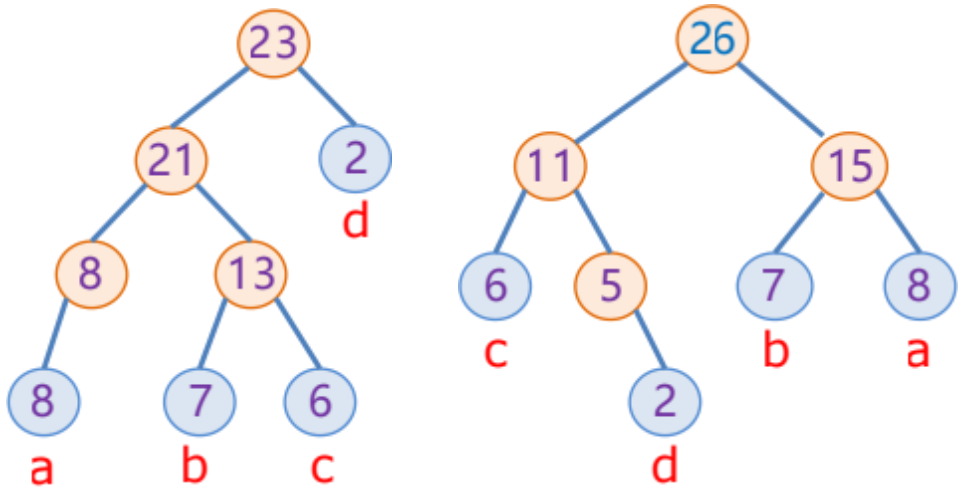
$WPL_1 > WPL_2$

T2 是一棵比 T1 性能更好 的二叉树

什么是树的性能？

5.5.1 Huffman树

- 加权路径长度WPL的内涵
  - 以字符编码应用场景为例：
    - 每个叶结点表示一个字符
    - **叶结点的权值**表示该字符在文本中出现的频度
    - **叶结点的路径长度**表示该字符的编码长度



$WPL_1 = 67$　　$WPL_2 = 48$

WPL表示一个文本转换为编码后最终的总编码长度

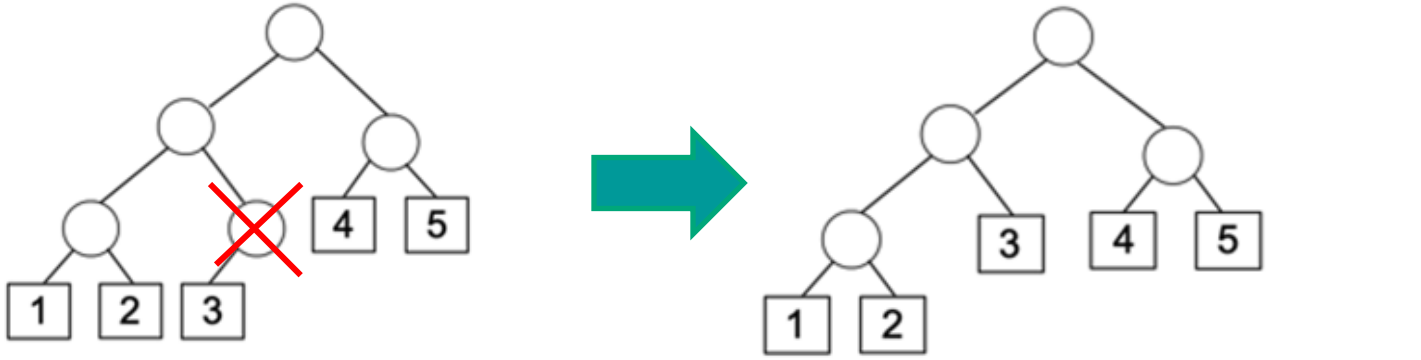
如何判断树 T2 是不是就是最优二叉树呢？

如何构建具有最小加权路径长度的二叉树以实现总编码长度的压缩？

5.5.1 Huffman树

- **定理5-4：** 最优二叉树（Huffman树）是满二叉树

证明：假设带权二叉树中存在度为1的中间结点。删除度为1的中间结点并把其唯一的子结点与父结点直接相连，使得从树根到该中间结点的所有子孙结点的路径长度减1，能够减小树的带权路径长度。因此，最优二叉树不含度为1的中间结点。



删除度为1的中间结点

减小树的带权路径长度，使其成为最优二叉树

5.5.1 Huffman树

- **命题5-5**：最优二叉树中，如果两个叶结点的权重值不同，则权重值小的叶结点在树中的层数大于等于权重值大的叶结点

证明：反证法。

- 设最优二叉树T的带权路径长度为WPL(T)，其中叶结点u的权重w_u小于叶结点v的权重w_v，即$w_u < w_v$
- 首先假定u在树中的层数比v的层数小，即level(u)<level(v)
- 交换叶结点u和v，可得新的带权二叉树T*, 且

$$WPL(T^*) = WPL(T) + w_u(level(v) - level(u)) + w_v(level(u) - level(v)) < WPL(T)$$

与T是最优二叉树矛盾。证明 level(u) ≥ level(v)。

交换权重值相同或者在树中同一层上的两个叶结点，不会改变二叉树的带权路径长度

5.5.1 Huffman树

- **命题5-6**：给定一组叶结点权重，存在最优二叉树，权重最小和次小的叶结点在树的最下层并且互为兄弟结点。

证明：

- 最优二叉树是满树，因此最下层必有两个以上的叶结点
- 如果权重最小的叶结点不在最下层，则最下层所有结点的权重都必须等于最小值，因此可以通过交换把权重最小的叶结点移到最优二叉树的最下层
- 同理可证权重次小的叶结点也在最下层
- 在最下层权重最小叶结点必有兄弟结点（满二叉树）。如果权重最小结点和次小结点不是兄弟，可以把最小结点的兄弟与次小结点交换，使两个结点共有一个父结点

5.5.2 Huffman算法

- **中间结点的权重**：除了叶结点带有权重，带权二叉树各中间结点也可定义权重，等于它的左子结点和右子结点的权重之和
- **Huffman算法**：一种至下而上构建最优二叉树的方法，通过不断合并两个带权二叉树，最终生成最优二叉树，具体过程如下：

(1) 对于给定的一组权重$w_1,w_2,\ldots,w_n$（$n\geq 2$），首先创建n棵只有一个结点（叶结点）的二叉树 $T=\{T_1, T_2, \ldots, T_n\}$，其中$T_j$的根结点权重为$w_j$（$1\leq j\leq n$）；

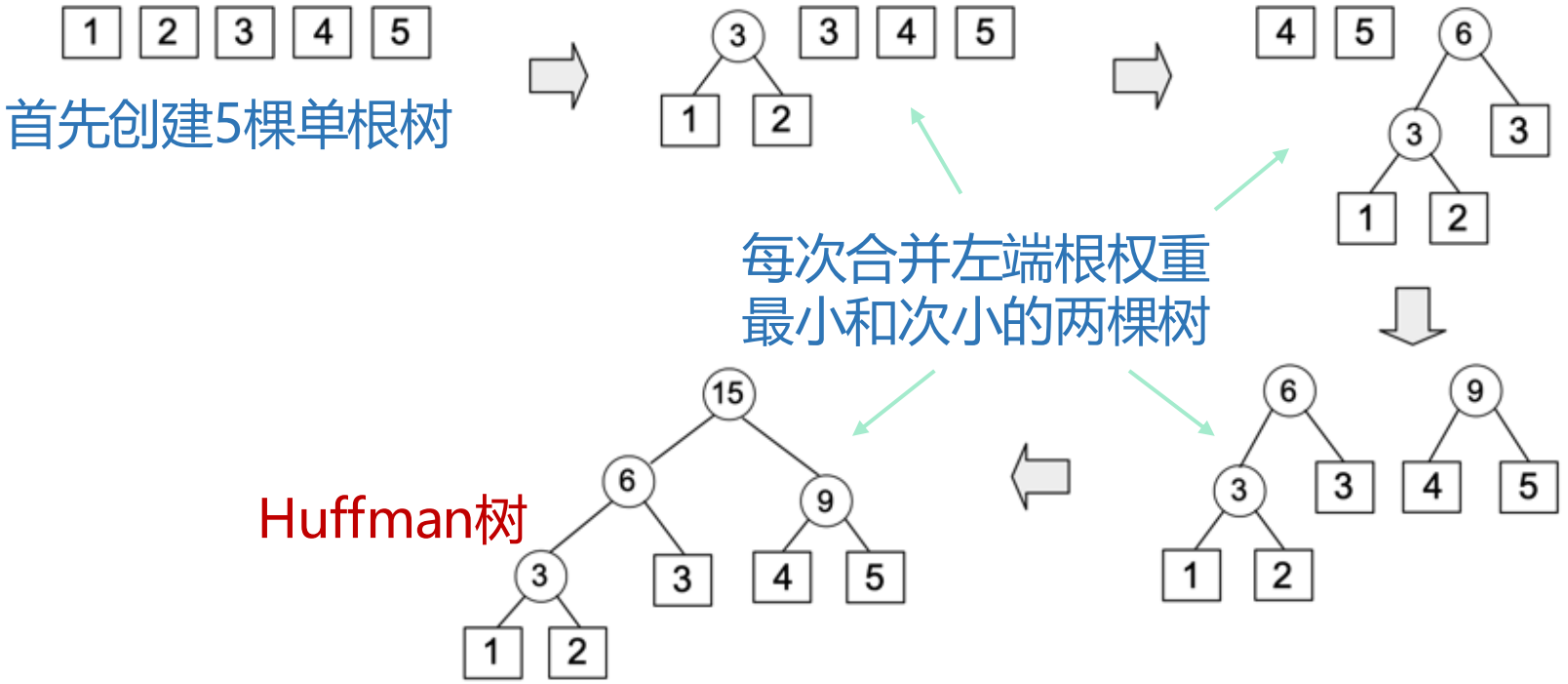
(2) 创建新的结点，并从T中取出根结点权重最小和次小的两个二叉树， 分别作为新结点的左右子树，设置结点的权重为左右子树的根结点权重之和；

(3) 把(2)构成的新二叉树插入二叉树集T中；

(4) 重复(2)和(3)的操作，直到T中只剩一个二叉树。最后剩下的二叉树就是所要构建的Huffman树。

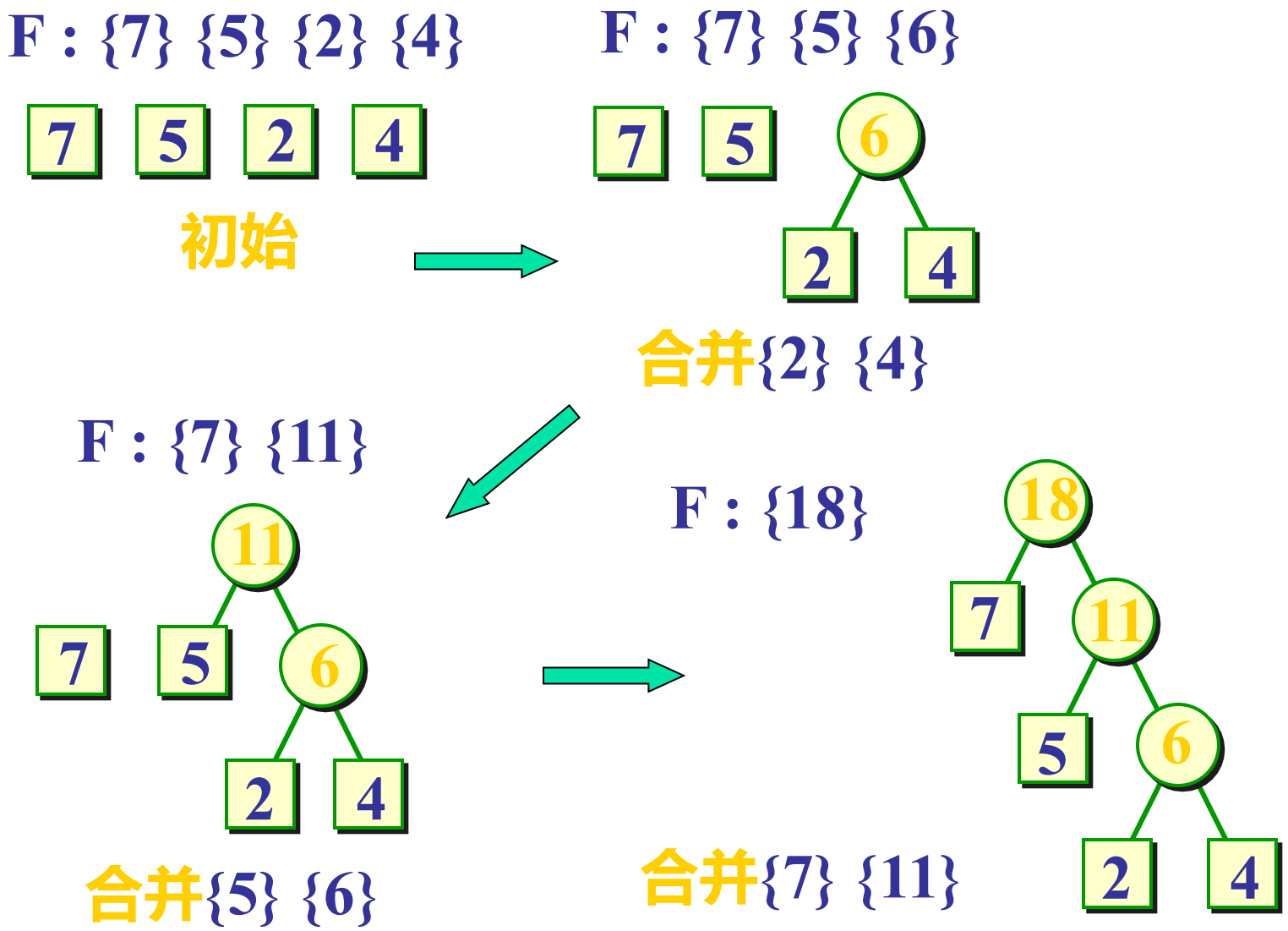
5.5.2 Huffman算法

- **Huffman算法**：一种至下而上构建最优二叉树（Huffman树）的方法，通过不断合并两个带权（最小、次小）二叉树，最终生成最优二叉树



最优二叉树的构建过程

5.5.2 Huffman算法



5.5.2 Huffman算法

- 算法5-15：创建Huffman树 CreateHuffmanTree(w)

输入：权重值的数据集w，$|w| \geq 2$
输出：Huffman树

```
tree_set ← ∅  //二叉树集合的初始化
n ← Length(w)        //n个权重
for i ← 1 to n do      //初始化n个二叉树
| tree ← new BinaryTreeNode( ) //创建叶结点
| tree.left  ← NIL
| tree.right ← NIL
| tree.weight ← Extract(w)    //从数据集w中取出一个值，作为结点权重
| Insert(tree_set, tree)                  //将单结点二叉树放入集合tree_set
end
|     //合并二叉树
```

- 时间$T_{W_Extract}$
- 时间T_{Q_Insert}

5.5.2 Huffman算法

- 算法5-15：创建Huffman树 CreateHuffmanTree(*w*)

```
|
for i ← 1 to n-1 do      //合并二叉树，共n-1次
| tree ← new BinaryTreeNode( )      //新建树根结点
| tree.left ← ExtractMin(tree_set)      //取出根权重最小树作为左子树
| tree.right ← ExtractMin(tree_set)      //取出根权重次小树作为右子树
| tree.weight ← tree.left.weight + tree.right.weight  //设置新树的根权重
| Insert(tree_set, tree)                              //将新树插入集合tree_set
end
tree ← Extract(tree_set)      //取出集合中唯一的二叉树，即Huffman树
return tree
```

- 时间$T_{Q_ExtractMin}$
- 时间$T_{Q_ExtractMin}$
- 时间T_{Q_Insert}
- 时间$T_{Q_Extract}$

```
HuffmanTree CreateHuffmanTree(List w)
{
    HuffmanTree tree;
    TreeSet tree_set;
    int n, i;
    tree_set = (TreeSet)malloc(sizeof(struct HeadNode));
    InitTreeSet(tree_set); /* 二叉树集合的初始化 */
    n = Length(w); /* n个权重 */
    for (i=0; i<n; i++) { /* 初始化n个二叉树 */
        tree = (HuffmanTree)malloc(sizeof(struct BinaryTreeNode)); /* 创建叶结点 */
        tree->left = NULL;
        tree->right = NULL;
        tree->weight = w->data[i]; /* 从数据集w中取出一个值，作为结点权重 */
        Insert(tree_set, tree); }
    for (i=1; i<n; i++) { /* 合并二叉树，共n-1次 */
        tree = (HuffmanTree)malloc(sizeof(struct BinaryTreeNode)); /* 新建树根结点 */
        tree->left = ExtractMin(tree_set); /* 取出根权重最小树作为左子树 */
        tree->right = ExtractMin(tree_set); /* 取出根权重最小树作为右子树 */
        tree->weight = tree->left->weight + tree->right->weight; /* 设置新树的根权重 */
        Insert(tree_set, tree); /* 将新树插入集合tree_set */ }
    tree = ExtractMin(tree_set); /* 取出集合中唯一的二叉树，即Huffman树 */
    return tree;
}
```

5.5.2 Huffman算法

- **Huffman算法**：一种至下而上构建最优二叉树的方法，通过不断合并两个带权二叉树，最终生成最优二叉树

- **算法分析**：
  - 用n个权重值创建了n个单根二叉树，并依次放入集合tree_set中，时间复杂度为$O(n)*(T_{Q_Insert} + T_{W_Extract})$
  - 合并两个二叉树的时间是$2T_{Q_ExtractMin} + T_{Q_Insert}$
  - 算法的时间复杂度等于 $O(n)*T_{Q\_Insert} + O(n)*T_{Q\_ExtractMin} + O(n)*T_{W\_Extract}$
  - 假设数据集w和tree_set直接用线性表实现，T_{Q_Insert}和$T_{W_Extract}$都是O(1)，但$T_{Q_ExtractMin} = O(n)$，即在tree_set中顺序查找权重最小二叉树的时间，因此构建Huffman树的时间复杂度达到$O(n^2)$
  - 更高效的构建方案是使用比线性表更复杂的数据结构，比如堆，这将在第6章中介绍。

5.5.3 Huffman编码

- **问题**：如何传输由字母a、b、c、w、z组成的字符串“baaacabwbzc”?
- **关键**：需要将文字符串转换成计算机能处理的二进制字符串（**编码**）
- **定长码**：把每个字母转换成固定长度的二进制字符串

baaacabwbz

定长码：a–000, b–001, c–010, w–011, z–100 → 编码

00100000000001000000101100l100010 长度33

- **不定长码**：使用频率高的字母采用短编码，频率小的采用长编码

baaacabwbzc → 频率：a(4), b(3), c(2), w(1), z(1)

编码 → 不定长码：a–01, b–00, c–10, w–110, z–111

000101011001001100011110 长度24

不定长码比定长码的编码效率高！

5.5.3 Huffman编码

- **前缀码**：一种常用的不定长码，每个字母的编码都不是其它字母编码的前缀

a–1, b–00, c–10, w–110, z–111

非前缀码

对二进制字符串解码会出现歧义

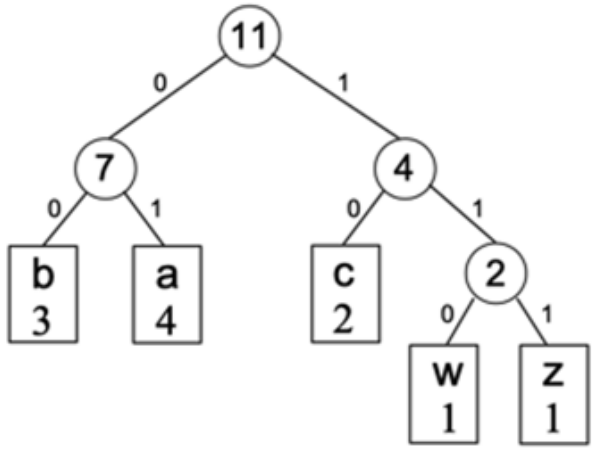
例：“1101” → “wa”

“1101” → “aca”

要设计不等长编码，必须使任一字符的编码都不是另一个字符的前缀——>前缀码

a–01, b–00, c–10, w–110, z–111

前缀码



可用二叉树表示前缀码

5.5.3 Huffman编码

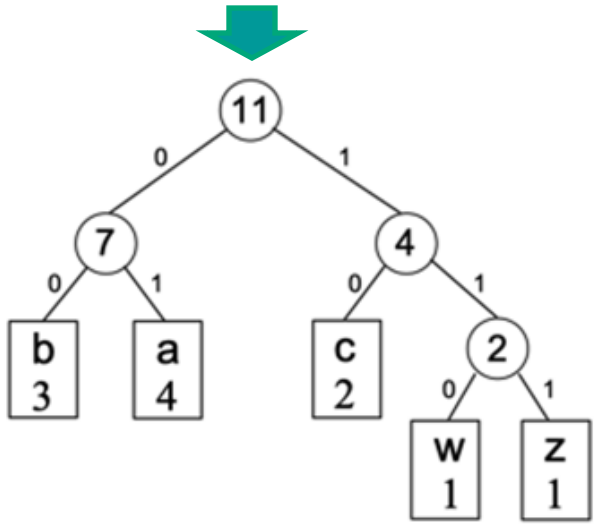
- 使用二叉树设计前缀码

前缀码树

- 各结点的左分支对应0，右分支对应1
- 每个叶结点记录唯一的一个字母
- 从根到各叶结点经过的分支序列表示对应字母的编码，同时编码长度等于路径长度
- 各叶节点中的数值表示权重，等于对应字母在字符串“baaacabwbzc”中出现的次数（频率）

a–01, b–00, c–10, w–110, z–111

前缀码



前缀码树的带权路径长度WPL = 字符串“baaacabwbzc”的编码长度

5.5.3 Huffman编码

- **问题**：给定一个字符串，求最优前缀码，使编码出的二进制字符串长度最短
  - 等价于：给定权重集，构造带权路径长度WPL最短的前缀码树（Huffman树）

前缀码树的带权路径长度WPL

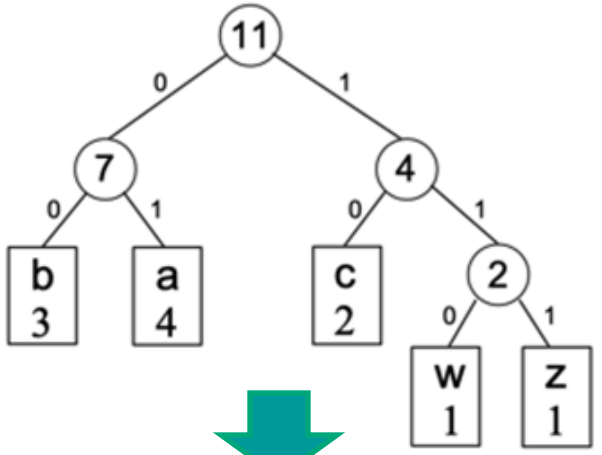
=

字符串的编码长度

↓

最优前缀码可用Huffman算法求解，
并把求得的前缀码称为Huffman编码

字符串“baaacabwbzc”对应权重集{3,4,2,1,1}的Huffman树



↓

a–01, b–00, c–10, w–110, z–111

是字符串“baaacabwbzc”的Huffman编码

5.5.3 Huffman编码

- 算法5-16：使用Huffman树对二进制字符串解码

输入： 非空前缀码树*tree*，二进制字符串*binary_code*
输出： 解码后的文字符序列

```
p ← tree    //指向树根
n ← Length(binary_code)  //二进制字符串长度
for i ← 0 to n-1 do
| if binary_code[i] = 0 then
| | p ← p.left      //遇到0, 沿左分支下移
| else   //binary_code[i] = 1
| | p ← p.right     //遇到1, 沿右分支下移
| end
| if p.left = NIL 且 p.right = NIL then  //到达叶结点
| | print  p.data     //输出文字符
| | p ← tree         //返回树根，重新开始解码
| end
end
```

算法分析
对长度为n的二进制字符串，使用Huffman（前缀码）树只需要O(n)的时间就能解码生成原来的文字串

5.5.3 Huffman编码

- 算法5-16：使用Huffman树对二进制字符串解码

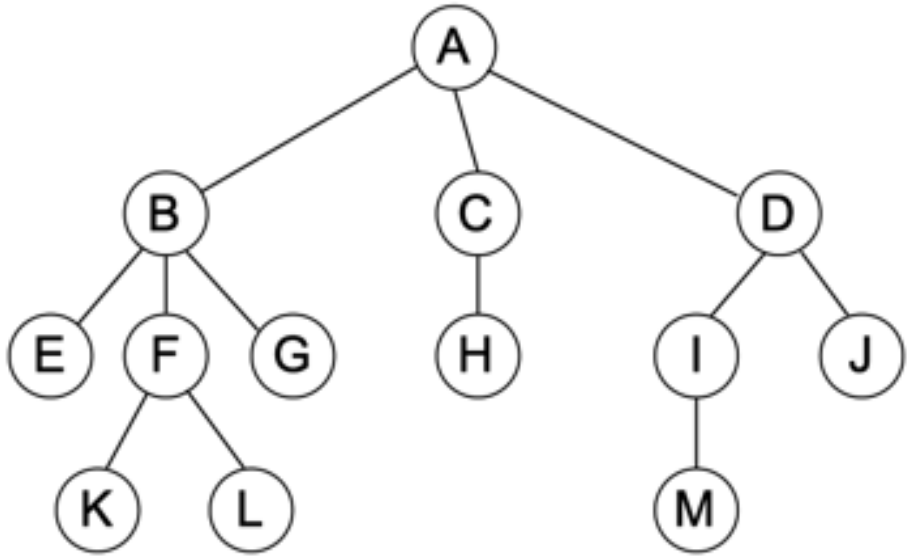
```
void Decoding(HuffmanTree tree, char binary_code[])
{
    HuffmanTree p;
    int n, i;
    p = tree; /* 指向树根 */
    n = strlen(binary_code); /* 二进制字符串长度 */
    for (i=0; i<n; i++) {
        if (binary_code[i]=='0') {
            p = p->left; /* 遇到0, 沿左分支下移 */ }
        else { /* binary_code[i]=='1' */
            p = p->right;  }
        if (p->left==NULL && p->right==NULL) { /* 到达叶结点 */
            printf("%c", p->data); /* 输出文字符 */
            p = tree; /* 返回树根，重新开始解码 */ }
    }
}
```

提纲

- 5.5 Huffman树与Huffman编码
- 5.6 树与森林
- 5.7 扩展延伸
- 5.8 应用场景：决策树

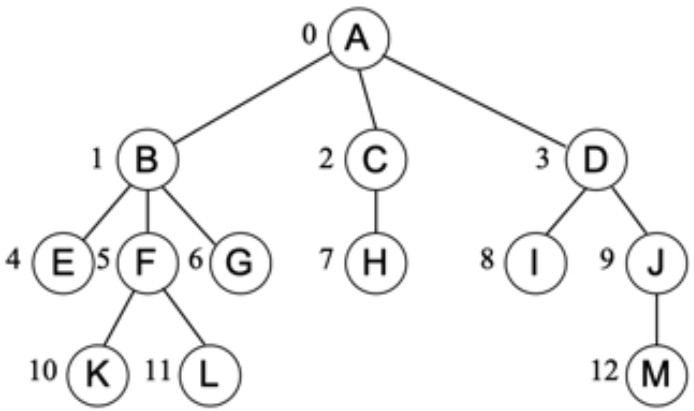
5.6.1 树的存储结构

- 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
  - 孩子兄弟表示法



5.6.1 树的存储结构

- 父亲表示法：要求每个结点保存父结点的位置信息
- 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

5.6.1 树的存储结构

- 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*

```
Position FindRoot(Tree tree, Position x)
{
    while (tree[x].parent != -1) {
        x = tree[x].parent;
    }
    return x;
}
```

时间复杂度O(H), H表示树的高度

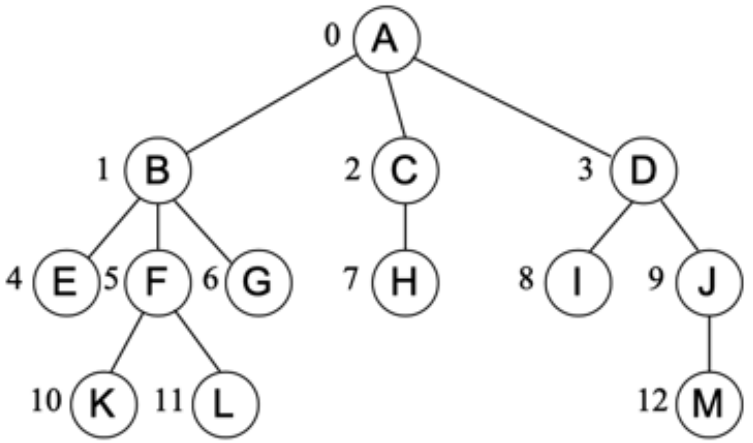
5.6.1 树的存储结构

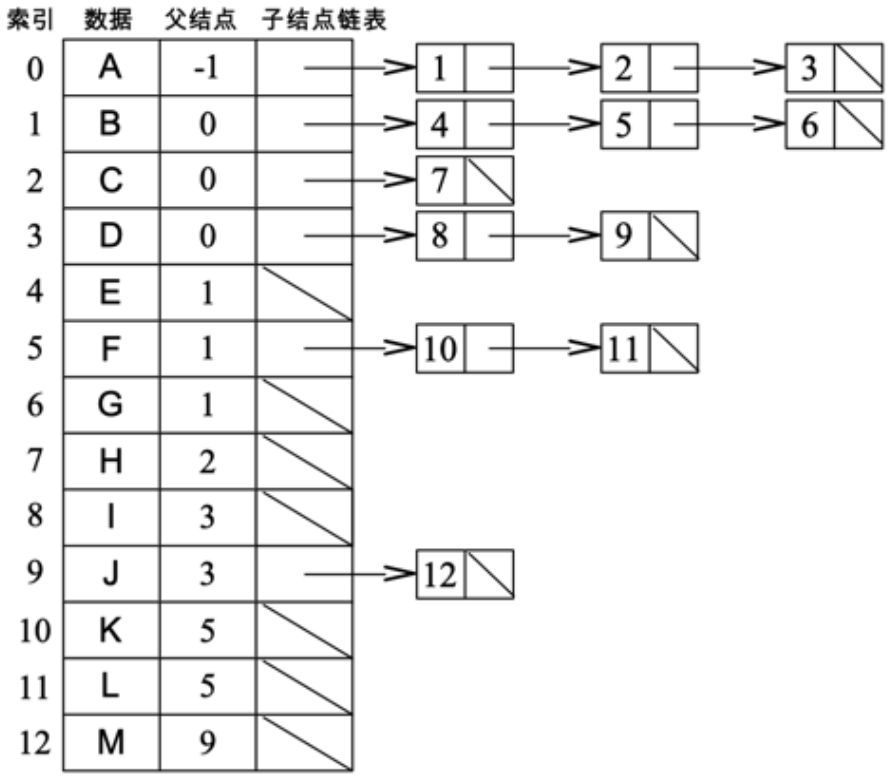
- 父亲表示法优点
  - 便于查找祖先结点
  - 每个结点仅需存放父结点位置，节省存储空间
  - 可用于实现并查集（不相交集）
- 父亲表示法缺点
  - 查找子结点或兄弟结点，需对整个树进行遍历，时间效率低

5.6.1 树的存储结构

- 孩子表示法：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

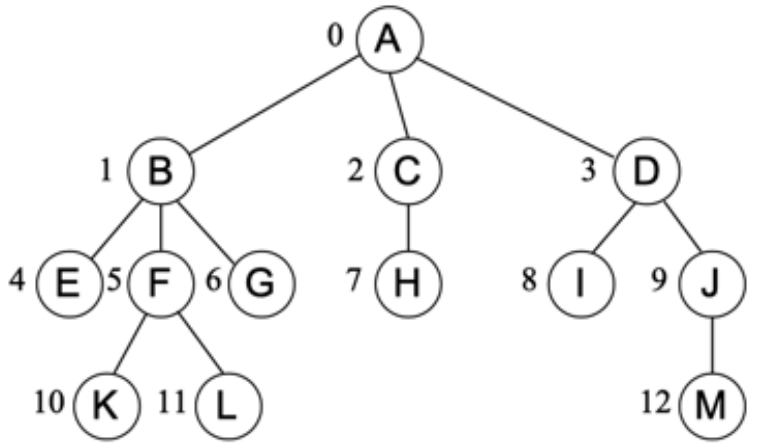
链表中各子结点按从左向右的顺序排列





5.6.1 树的存储结构

- 孩子表示法：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

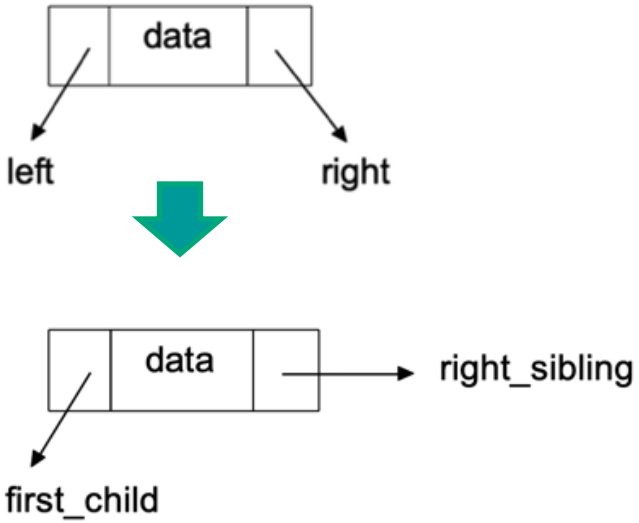
结点B的第一个孩子: E，下一个兄弟：C

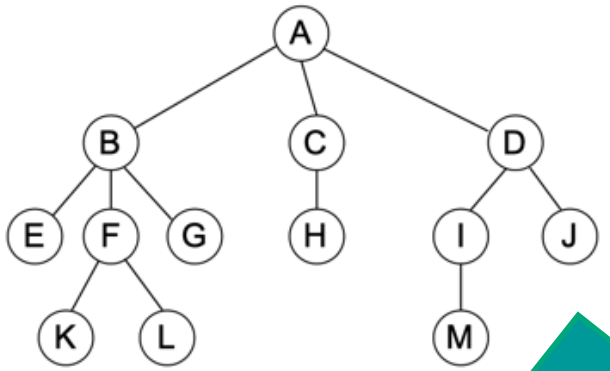
5.6.1 树的存储结构

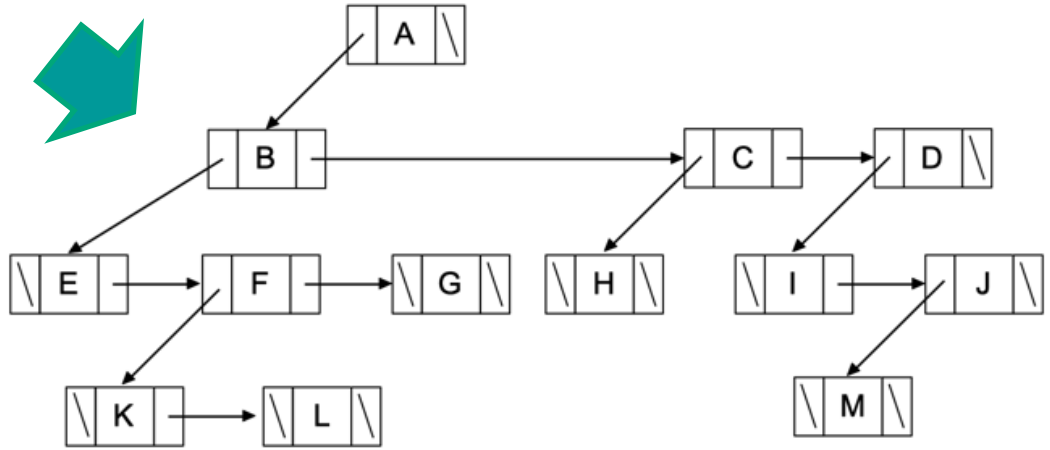
- 孩子兄弟表示法
  - 树应用最广泛的存储结构。
  - 每个结点存放其第一个孩子和下一个兄弟的信息，可以直接使用二叉链表实现，因此这种表示法也称作二叉链表表示法
  - 二叉链表中的指针域left 改称 first_child，指向结点的第一个子结点；同时，指针域right 改称next_sibling，指向右侧的兄弟结点

5.6.1 树的存储结构

- 孩子兄弟表示法
  - 第一个孩子
  - 下一个兄弟





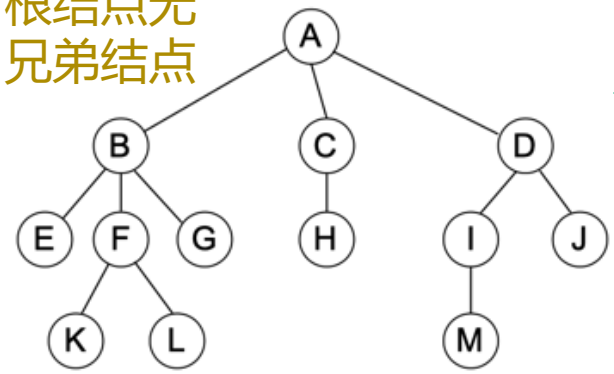


二叉链表结构

5.6.2 树与二叉树的转换

树

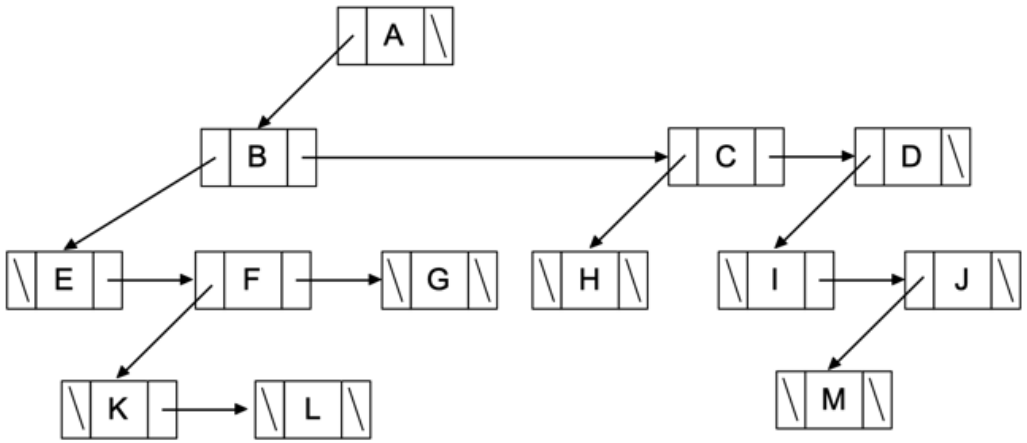
根结点无
兄弟结点



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

孩子兄弟表示法

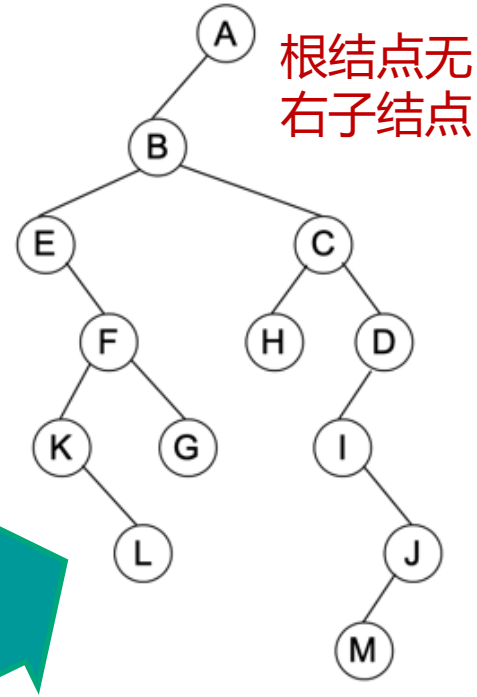


二叉链表结构

链接存储结构

二叉树

根结点无
右子结点



5.6.2 树与二叉树的转换

- 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

5.6.2 树与二叉树的转换

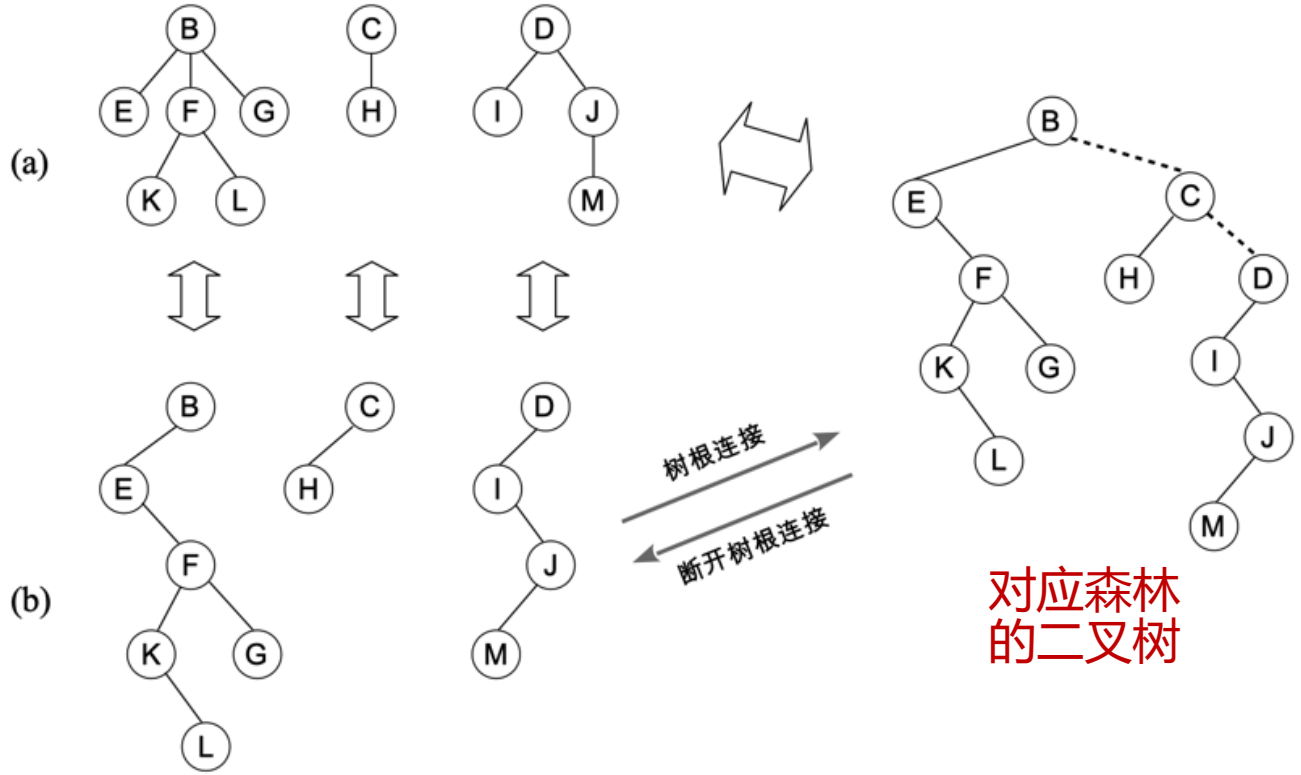
- 算法5-18：查找树中带有指定数据的结点 Search(*tree*, *x*)

```
Position Search(Tree tree, TElemSet x)
{
    Tree node_ptr;
    node_ptr = tree;
    if (node_ptr != NULL) {
        if (node_ptr->data != x) {
            node_ptr = Search(tree->first_child, x); /* 在子孙结点中查找 */
            if (node_ptr == NULL) { /* 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 */
            }
        }
    }
    return node_ptr;
}
```

5.6.2 树与二叉树的转换

- 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**

三棵独立的树



对应每棵树的二叉树

对应森林的二叉树

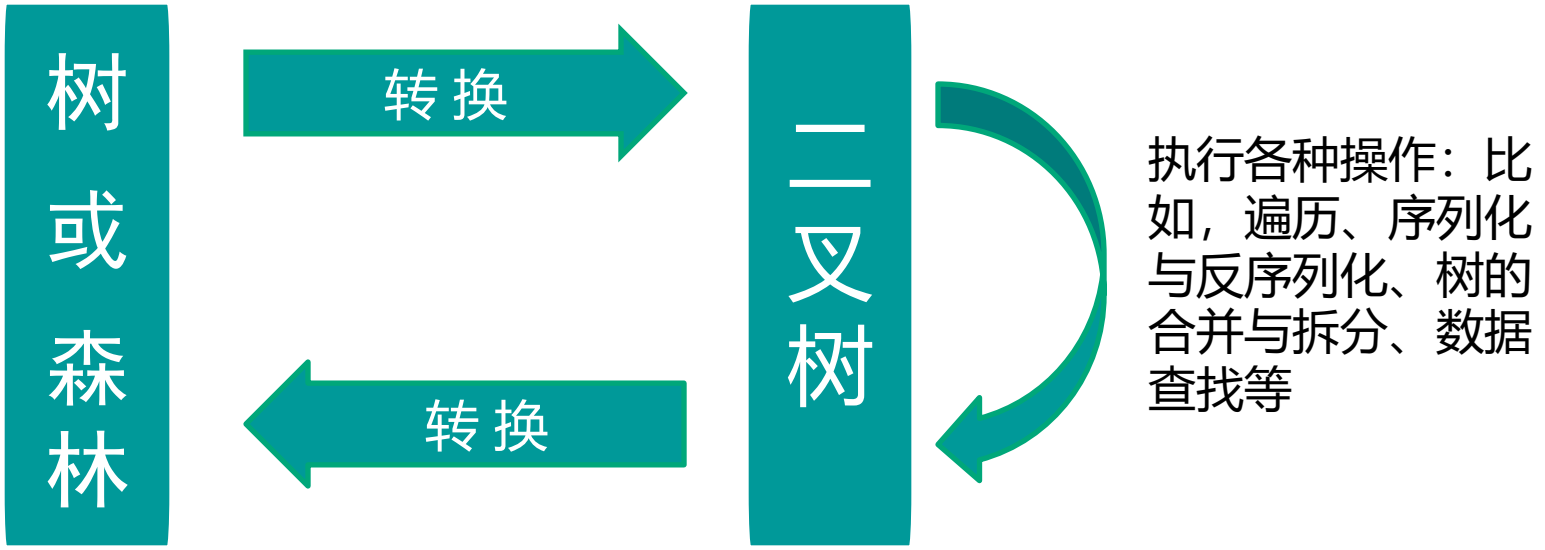
森林转换成二叉树

(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

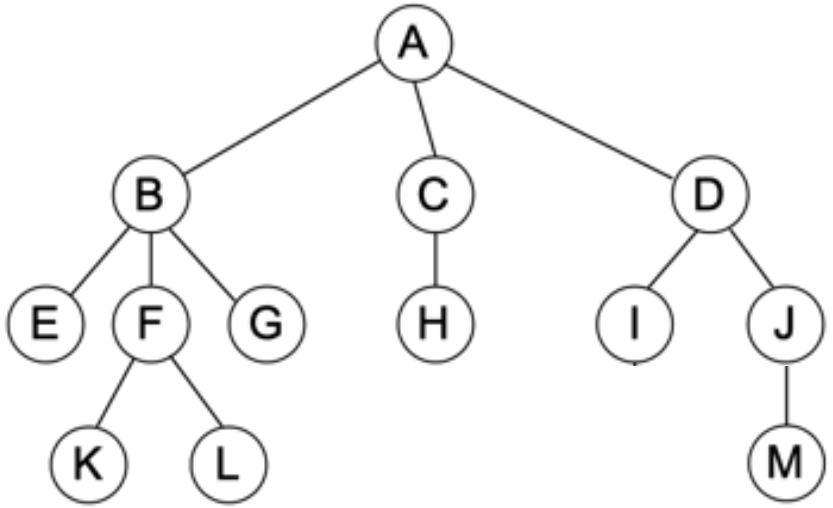
5.6.2 树与二叉树的转换

- 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



5.6.3 树与森林的遍历

- **树的深度优先遍历**：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

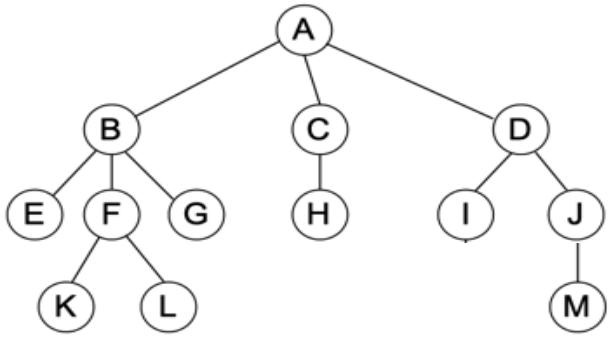


前序遍历：<A, B, E, F, K, L, G, C, H, D, I, J, M>

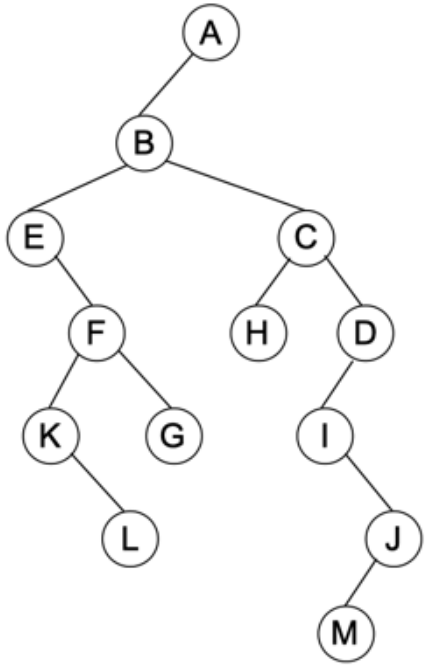
后序遍历：<E, K, L, F, G, B, H, C, I, M, J, D, A>

5.6.3 树与森林的遍历

- 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

5.6.3 树与森林的遍历

- 基于二叉树遍历方案的树遍历算法

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

二叉树的前序遍历算法

```
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
```

```
void PreOrder(Tree tree)
{
    if (tree != NULL) { /* 空树不做处理，直接返回 */
        Visit(tree); /* 先访问结点tree */
        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 */
    }
}
```

5.6.3 树与森林的遍历

- 基于二叉树遍历方案的树遍历算法

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

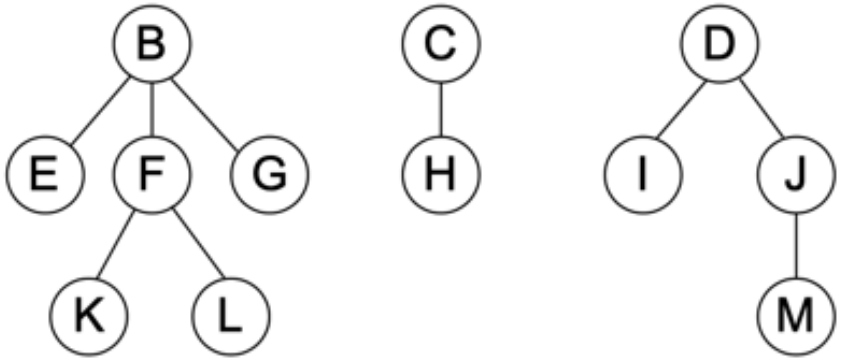
二叉树的中序遍历算法

```
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
```

```
void PostOrder(Tree tree)
{
    if (tree != NULL) { /* 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); /* 接下来访问结点tree */
        PostOrder(tree->next_sibling); /* 最后访问tree后序的兄弟结点及其子孙 */
    }
}
```

5.6.3 树与森林的遍历

- 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



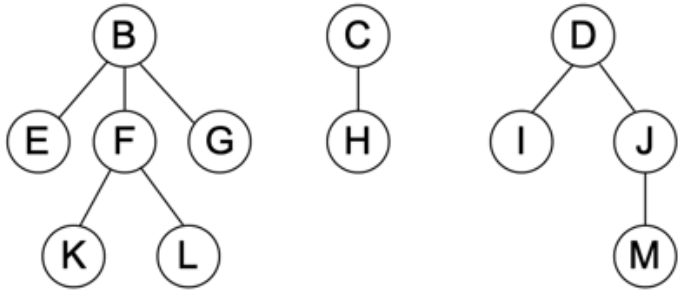
前序遍历：< B, E, F, K, L, G, C, H, D, I, J, M >

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

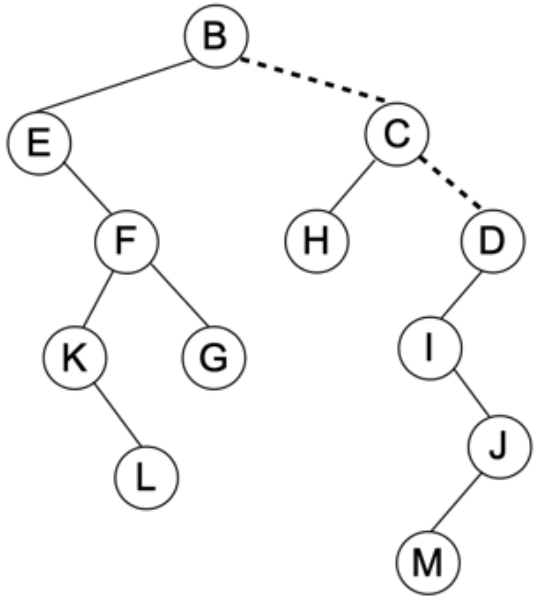
从左向右依次遍历每棵树!

5.6.3 树与森林的遍历

- 森林与对应的二叉树的遍历



对应



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

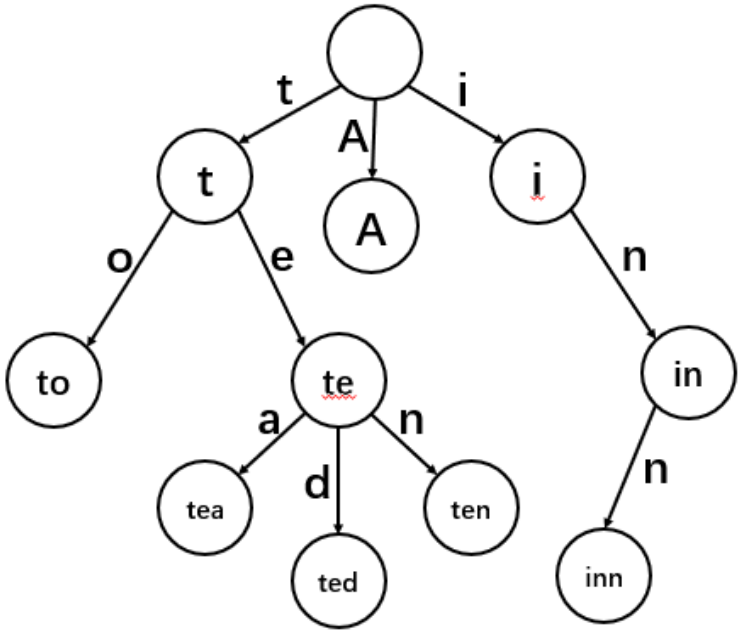
森林的后序遍历与对应的二叉树的中序遍历结果相同　算法5.20可用

提纲

- 5.5 Huffman树与Huffman编码
- 5.6 树与森林
- 5.7 扩展延伸
- 5.8 应用场景：决策树

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

5.7.1 拓展延伸：前缀树

- 前缀树是一种以空间换时间的算法

前缀树的时间效率

(1) 前缀树的2个基本操作：插入字符串、查找是否存放有指定的字符串

(2) 插入字符串以及查询字符串的时间开销均为O(n), n是字符串的长度，与前缀树本身的规模无关

(3) 前缀树各结点的数据不仅可以存放字符串，还能用于计数，比如统计其对应的字符串是多少个存放的字符串的公共前缀

前缀树的空间开销

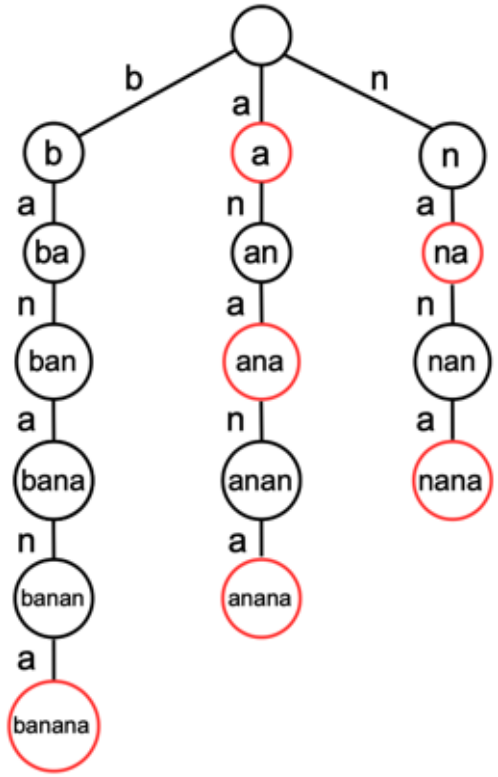
(1) 在插入新字符串时，字符串的每个字符都有可能创建一个新的（k叉树）结点，导致前缀树的内存消耗会非常大

(2) 最坏情况下，前缀树的空间复杂度可达O(KN)，其中N为所有字符串的长度之和，K为字符集规模

(3) 在实际应用中，前缀树通常用二维数组（顺序变）来实现

5.7.2 拓展延伸：后缀树与后缀自动机

- **后缀树**：用一个字符串的所有后缀构建的**前缀树**
- **典型应用**：最长重复子串问题、最长公共子串问题、精确字符匹配问题等

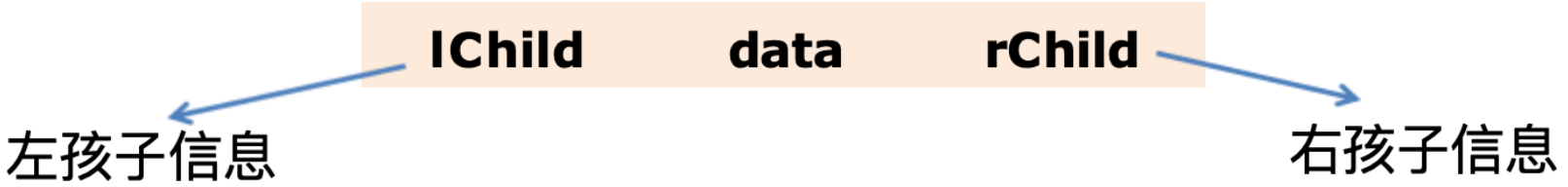


字符串“banana”构成的后缀树

- 使用字符串的所有后缀构建前缀树，可以在树上保留所有子串信息，由此大幅提升子串的查询效率
- 查询字符串S中是否包含字符串P，只需在由S构建的后缀树上检索P，并且查询的时间复杂度为O(|P|)，与后缀树规模无关，比KMP等算法效率高
- 建立字符串S的后缀树的时间与空间复杂度均为$O(K|S|^2)$， K是字符集规模
- 合并树中的相似结构，可以大幅压缩后缀树的规模，最终形成一个点和边的规模均为 O(|S|) 的有向无环图，这就是**后缀自动机**

习题 15*：线索二叉树

- 回顾： 二叉树的存储结构， 我们能从一个结点找到哪些相关信息？



  - 无法从标准的二叉树结点中获取直接前驱和直接后继；
  - 唯一的直接前驱和直接后继只能在遍历过程中获得。
- 如何改进？
  - 增加标志域来标识前驱和后继信息。

习题 15*：线索二叉树

- 线索二叉树
  - 包含线索结构的二叉链表结构称为线索链表，指向结点前驱和后继的指针称为线索；
  - 加上线索的二叉树称为线索二叉树（Threaded Binary Tree）
  - 对二叉树以某种遍历为基础转变为线索二叉树的过程称为线索化或序列化

若结点有左子树，则lTag=0，lChild域指向左孩子
若结点无左子树，则lTag=1，lChild域指向其前驱（线索）

lChild	lTag	data	rTag	rChild

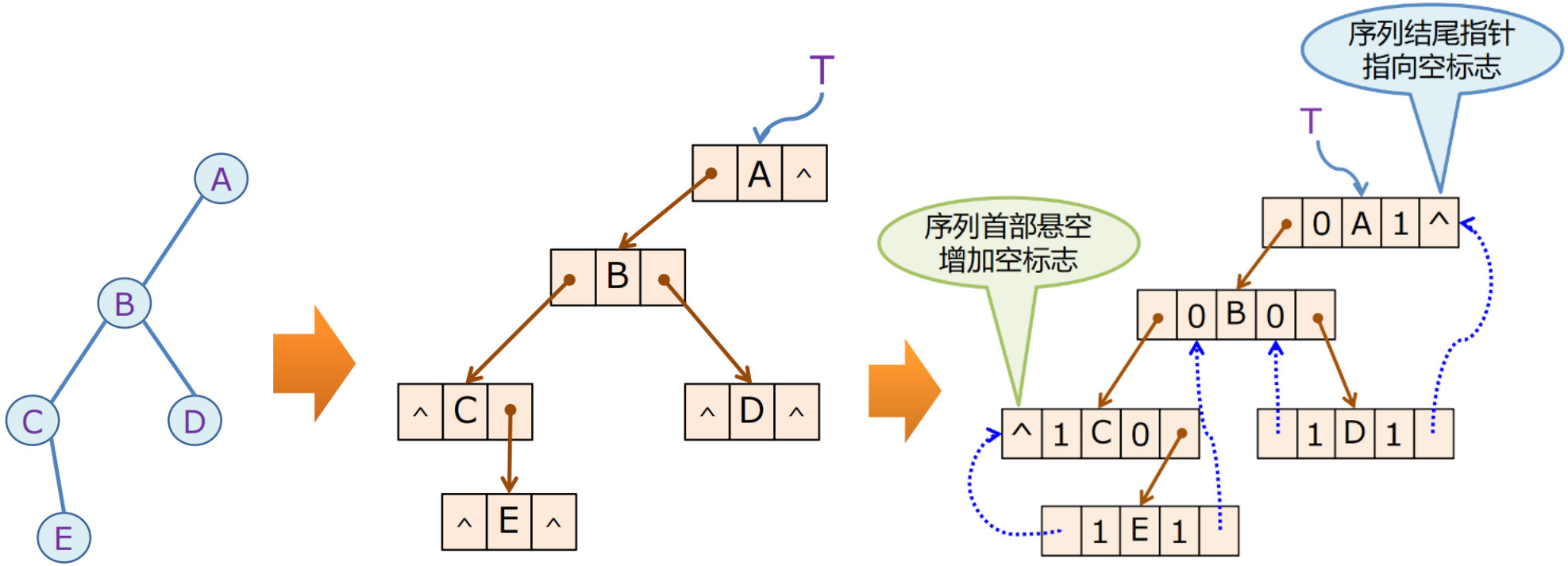
若结点有右子树，则 rTag=0，rChild域指向右孩子
若结点无右子树，则 rTag=1，rChild域指向其后继（线索）

二叉树的二叉线索存储表示

```
typedef struct TBNode
{
        char data;
        int ltag,rtag;
        struct TBNode *lchild;
        struct TBNode *rchild;
}TBNode;
```

习题 15*：线索二叉树

- 中序线索二叉树



中序遍历序列： C E B D A

lTag=0， lChild域指向左孩子； lTag=1， lChild域指向其前驱
rTag=0， rChild域指向右孩子； rTag=1， rChild域指向其后继

习题 15*：线索二叉树

- 遍历中序线索二叉树

```
TBNode *First(TBNode *p){
        while (p->ltag=0)//表示有左孩子（左孩子不为空）{
                return First(p->rchild); }
        return p;

}
TBNode *Next(TBNode *p){
        if (p->rtag==0) //表示有右孩子（右孩子不为空）{
                return First(p->rchild); }
        else{
                return p->rchild; //rtag==1,直接返回后继线索 }

}
void Inorder(TBNode *root){
        for (TBNode *p=First(root) ; p!=NULL ; p=Next(p)) {
                Visit(p); //Visit()是已经定义的访问p所指节点的函数}

}
```

提纲

- 5.5 Huffman树与Huffman编码
- 5.6 树与森林
- 5.7 扩展延伸
- 5.8 应用场景：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

- **实例**：

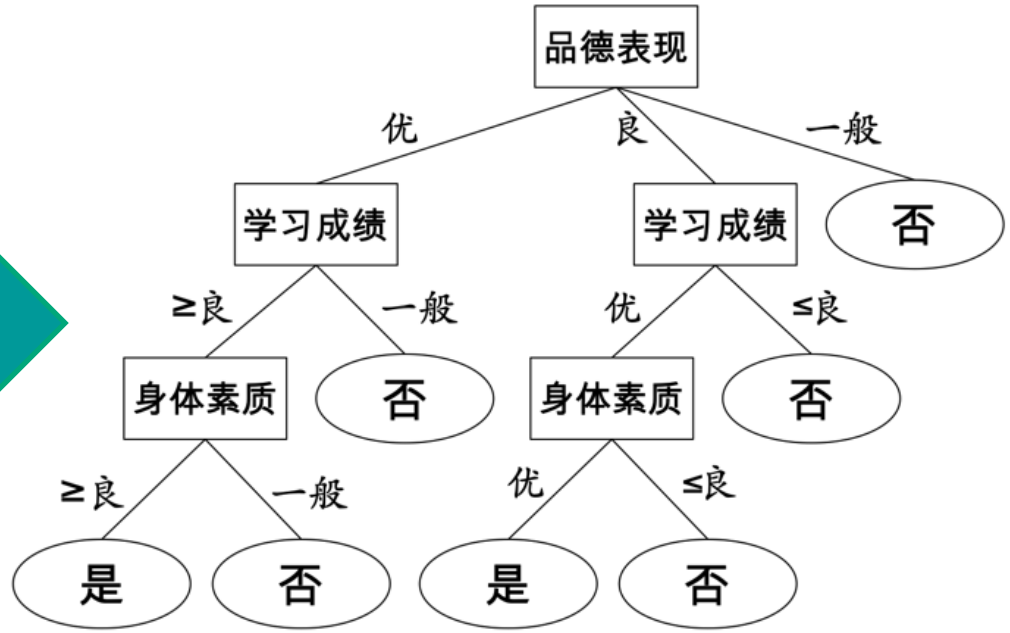
某高校的优秀学生选拔标准

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

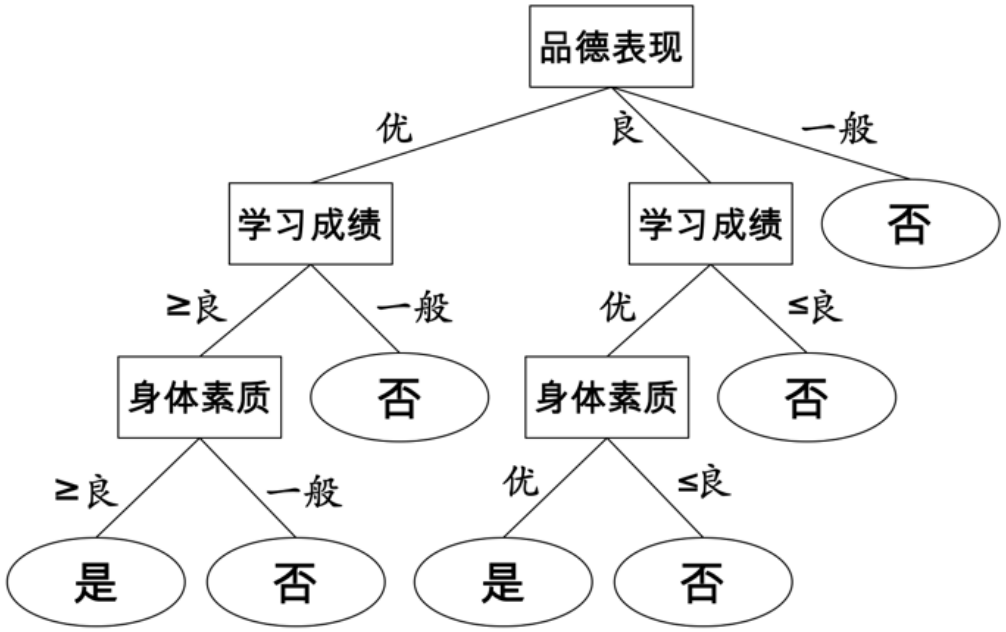


决策树

5.8 应用场景：决策树

- **决策树**：一种解决分类问题的算法

- **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

5.8 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：

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

5.8 应用场景：决策树

- ID3算法
  - 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
  - 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
  - 把数据集按特征值划分给每个子结点
  - 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$\{a_1, a_2, \ldots, a_m\}$ $(m \geq 1)$。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。

5.8 应用场景：决策树

- ID3算法案例

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

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

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：

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

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

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

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。

5.8 应用场景：决策树

维度	ID3	C4.5	CART（分类与回归树）
划分标准	信息增益（偏向多值特征）	信息增益率（解决 ID3 偏向性）	分类树：基尼系数；回归树：均方误差 / 绝对误差
数据类型	仅支持离散型特征	支持离散型 + 连续型特征（连续特征需离散化）	支持离散型 + 连续型特征（连续特征可直接处理）
树结构	多叉树（每个特征的取值对应一个分支）	多叉树	二叉树（每个节点仅分两个分支）
剪枝策略	无剪枝（易过拟合）	后剪枝（降低过拟合）	前剪枝 + 后剪枝（灵活控制复杂度）
缺失值处理	不支持	支持（通过权重调整样本）	支持（通过代理分裂或权重调整）
应用场景	简单分类任务（仅离散特征，无缺失值）	分类任务（需处理连续特征、缺失值，追求高鲁棒性）	分类 + 回归任务（需二叉树结构、灵活剪枝，适用场景最广）

小结

- Huffman树（最优二叉树）性质
  - 满二叉树
  - 权重小的叶结点所在层数大于等于权重大的叶结点所在层数
  - 权重最小和次小的叶结点在最下层且互为兄弟结点
- Huffman算法：不断合并两个带权路径最小二叉树，生成最优二叉树
- Huffman编码是前缀编码，任一字符编码都不是另一字符编码的前缀
- 编码过程：使用Huffman算法构建二叉树（Huffman树），然后根据Huffman树得到的编码表对字符进行编码。
- 译码过程：按照左0，右1的原则，从根结点开始依次对二进制码进行遍历。每次到达叶子结点时，完成一个字符的译码。反复执行该过程，直至所有二进制码都完成解译。

小结

- 树的存储结构有多种，例如：双亲表示法（顺序存储）、孩子表示法（二叉链表）、孩子兄弟表示法等。其中，孩子兄弟表示法是最常用的。任意一棵树都可以通过孩子兄弟表示法转换为对应的二叉树。
- 树和森林都能双向地转换成二叉树，并利用二叉树的操作解决一般树的问题。
- 根据结点访问次序的不同，树常见遍历方法包括先根遍历、后根遍历，其后根遍历与对应的二叉树的中序遍历一致。
- 在线索二叉树中，可以利用二叉链表中的空指针域（不含左右子树指针的结点）来存放指向某种遍历次序下的前驱结点和后继结点指针，这些附加的指针称为“线索”。
- 线索二叉树有利于加速查找结点前驱和后继的速度。

树与二叉树高频必刷题（LeetCode）

分类	题号	题目	考察点（高频原因）	难度
二叉树基础概念	104	Maximum Depth of Binary Tree	递归/迭代实现深度计算，二叉树基础操作入门	Easy
二叉树基础概念	226	Invert Binary Tree	镜像翻转二叉树，逻辑直观且高频考察	Easy
完全二叉树特性	222	Count Complete Tree Nodes	利用完全二叉树索引特性优化计数，避免暴力遍历	Easy
二叉树遍历（核心）	94	Binary Tree Inorder Traversal	中序遍历（递归＋迭代），BST相关题基础	Easy
二叉树遍历（核心）	102	Binary Tree Level Order Traversal	层序遍历（BFS），树的广度优先遍历模板题	Medium
二叉树遍历（综合）	236	Lowest Common Ancestor of a Binary Tree	后序遍历应用，最近公共祖先（面试高频考点）	Medium
二叉树重构	105	Construct Binary Tree from Preorder and Inorder Traversal	前序＋中序重构树，重构类题的经典模板	Medium
二叉树与表达式处理	224	Basic Calculator	中缀表达式求值（关联表达式树），综合逻辑题	Hard

树与二叉树高频必刷题（LeetCode）

分类	题号	题目	考察点（高频原因）	难度
Huffman树与编码	1733	Minimum Number of People to Teach	Huffman贪心思想应用，通过合并最小冲突语言对优化教学人数，关联前缀编码优化逻辑	Medium
	1167	Minimum Cost to Connect Sticks	直接模拟Huffman树构建过程，计算合并最小权值节点的总代价（WPL计算），高频贪心考点	Medium
树与森林	1110	Delete Nodes And Return Forest	树的存储结构调整，递归遍历实现节点删除与森林生成，覆盖树与森林转换核心考点	Medium
	543	Diameter of Binary Tree	树的后序遍历应用，统计子树高度以计算树的直径，树的遍历与属性计算基础题	Easy
	112	Path Sum	深度优先遍历（DFS）遍历树结构，判断路径和是否匹配，树的遍历与路径分析入门题	Easy
二叉树拓展-重构	106	Construct Binary Tree from Inorder and Postorder Traversal	中序＋后序序列重构二叉树，反向处理后序序列定位根节点，重构逻辑对称训练题	Medium
二叉树拓展-中序线索化	173	Binary Search Tree Iterator	实现BST迭代器，需用线索化思想（或Morris遍历）实现O(1)空间，线索化应用高频题	Medium