# 5.4.4 二叉树的序列化与反序列化

**二叉树的序列化**：按某种遍历方案访问所有结点并依次输出结点数据，由此形成结点的线性序列

**序列化的作用**：将树的非线性结构转换成线性结构，便于使用线性表或字符串等存储

**二叉树的反序列化**：根据线性序列重构原始的二叉树

**问题**：

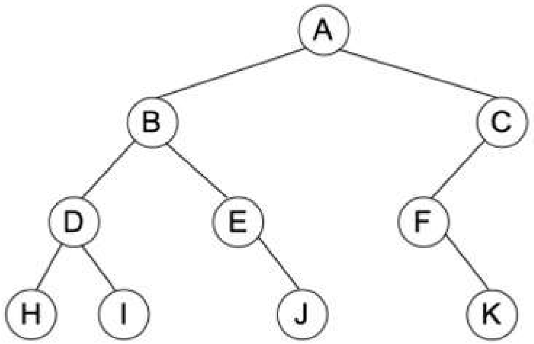
- 完全二叉树的顺序存储是一种序列化方案，并且可以根据结点间的相对位置确定它们之间的逻辑关系，重构出二叉树
- 但该方法对于一般的二叉树可能造成空间浪费，在最坏情况下，空间复杂度达到$O(2^n)$

- 常用的前序遍历或层序遍历算法，产生的结点序列只包含了树结构的部分信息，通常无法重构二叉树

# 5.4.4 二叉树的序列化与反序列化

**二叉树的序列化：**按某种遍历方案访问所有结点并依次输出结点数据，由此形成结点的线性序列

**二叉树的反序列化：**根据线性序列重构原始的二叉树




前序遍历：<A, B, D, H, I, E, J, C, F, K>

从序列中，最多只能确定A是根结点，其它信息，如左子树包含哪些结点等无法确定

**问题**：如何使前序遍历的结果能够重构二叉树？

# 5.4.4 二叉树的序列化与反序列化

**二叉树的序列化**：按某种遍历方案访问所有结点并依次输出结点数据，由此形成结点的线性序列

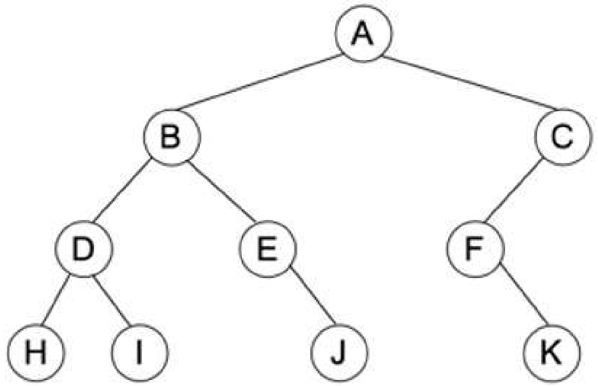
**二叉树的反序列化**：根据线性序列重构原始的二叉树

**序列化方法**

- 用特殊符号#表示空结点
- 当在前序遍历过程中遇到空结点或空子树时，不是直接返回，而是输出符号#，从而将空结点也标记在序列中

# 前序序列

**二叉树前序序列化：**前序遍历二叉树，如果结点非空，输出结点数据，否则输出#




通过在结点序列中插入空记号，记录二叉树的非线性结构

<A, B, D, H, #, #, I, #, #, E, #, J, #, #, C, F, #, K, #, #, #>

## 算法5-11：二叉树前序序列化 PreOrderSerialize(*tree*)

输入：二叉树*tree*
输出：二叉树的前序序列

```
if tree = NIL then  //空树
| print #    //输出特殊符号，代表空结点
else
| print tree.data   //输出结点数据
| PreOrderSerialize (tree.left)     //对左子树前序序列化
| PreOrderSerialize (tree.right)    //对右子树前序序列化
end
```

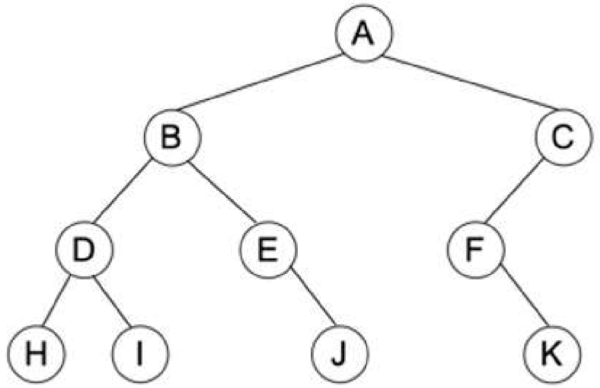
基于前序遍历算法

# 前序序列

## 前序序列的反序列化

**从前序序列先端依次读取数据，执行下面的操作：**

- 如果读取的数据是#，则返回NIL，表示空结点或空树
- 否则新建二叉树结点，把数据代入结点并递归地重构结点的左子树和右子树，然后返回结点。




<A, B, D, H, #, #, I, #, #, E, #, J, #, #, C, F, #, K, #, #, #>

前序序列

**算法5-12：二叉树前序序列的反序列化 PreOrderDeSerialize(*preorder, n*)**

输入：存放二叉树前序序列的线性表*preorder*，表中元素个数*n* (n>0)

输出：二叉树

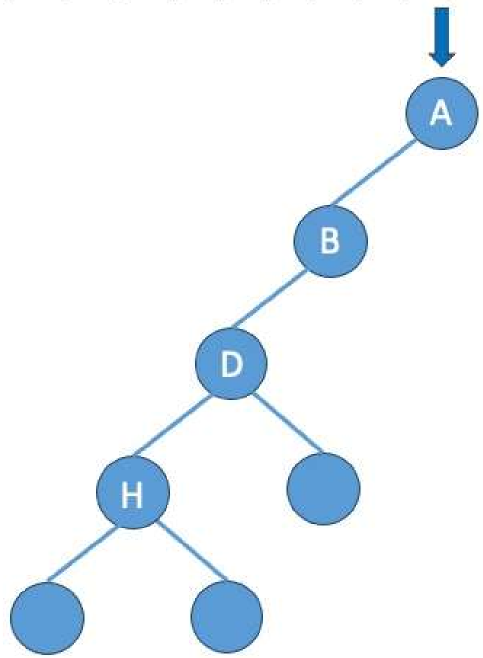
全局变量：k，初始值为-1

```
k ← k + 1
tree ← NIL  //初始化一个空树
if k < n then  //k是线性表的有效序号
| data ← Get(preorder, k)      //读出线性表第k个元素
| if data ≠ # then     //非空记号
| | tree ← new BinaryTreeNode( )  //新建二叉树结点
| | tree.data ← data             //代入数据
| | tree.left ← PreOrderDeSerialize(preorder, n)  //重构左子树
| | tree.right ← PreOrderDeSerialize(preorder, n)  //重构右子树
| end
end
return tree  //返回新建的二叉树或空树
```

# 前序序列

## 前序序列的反序列化

<A, B, D, H, #, #, I, #, #, E, #, J, #, #, C, F, #, K, #, #, #>




**从前序序列先端依次读取数据，执行下面的操作：**

- 如果读取的数据是#，则返回NIL，表示空结点或空树
- 否则新建二叉树结点，把数据代入结点并递归地重构结点的左子树和右子树，然后返回结点。

Continue...

# 5.4.4 二叉树的序列化与反序列化

**二叉树的序列化：**按某种遍历方案访问所有结点并依次输出结点数据，由此形成结点的线性序列

**二叉树的反序列化：**根据线性序列重构原始的二叉树

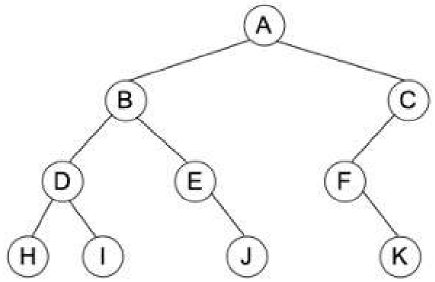
**思考：**

（1）与前序序列类似，在二叉树的层序遍历过程中，输出表示空结点的标记，可生成二叉树的层序序列，并根据层序序列重构二叉树（过程及算法参考教材）

（2）（经典问题）用前序遍历与中序遍历结果重构二叉树，分析重构条件及算法的时间复杂度

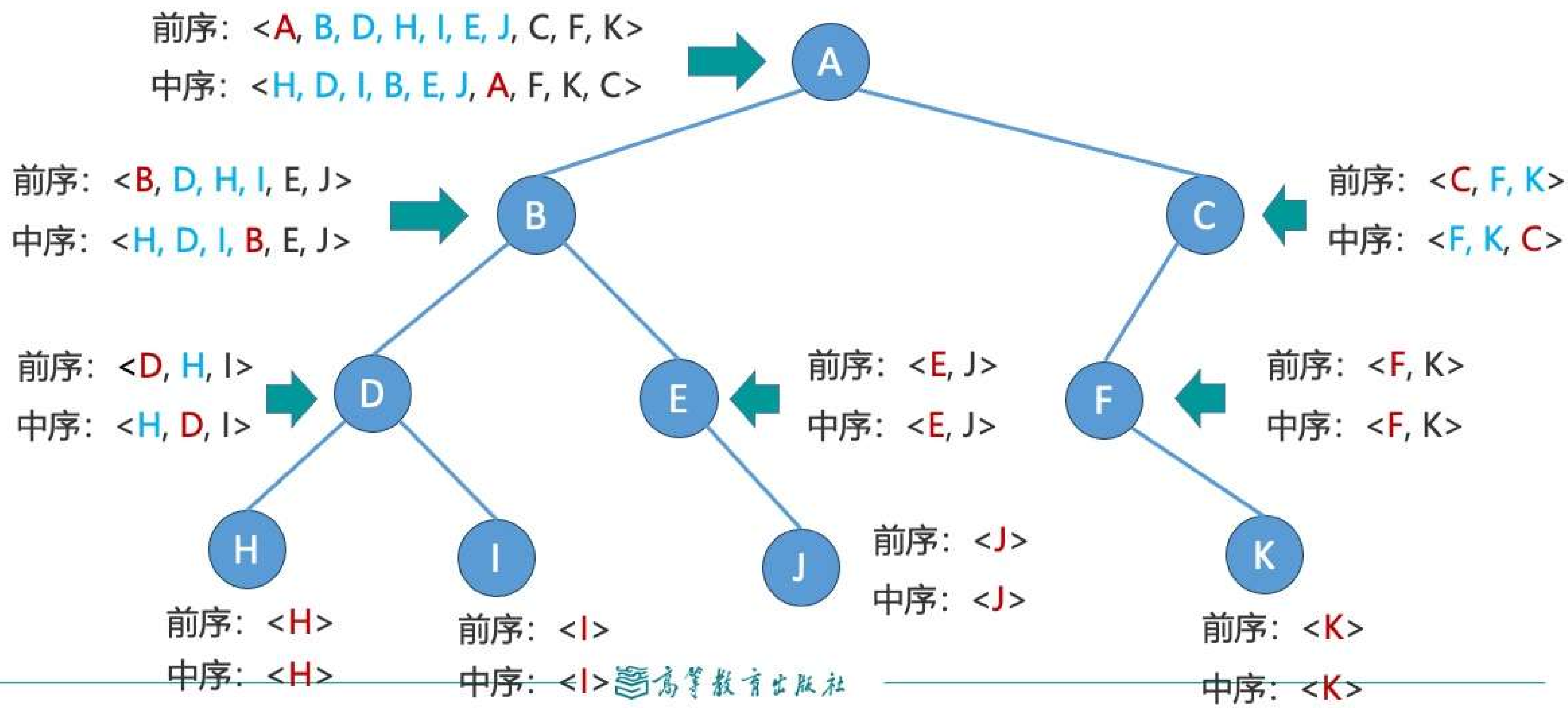
前序遍历：<A, B, D, H, I, E, J, C, F, K>

中序遍历：<H, D, I, B, E, J, A, F, K, C>



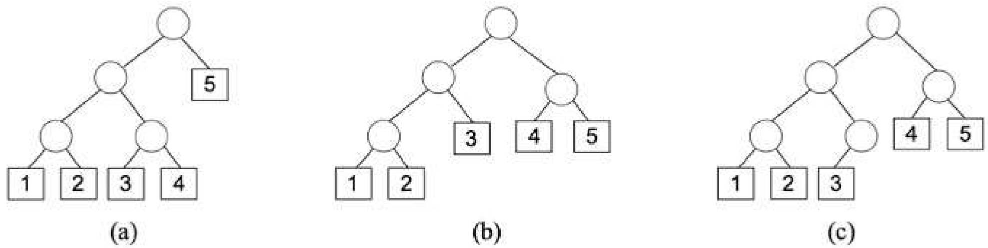
# 5.4.4 二叉树的序列化与反序列化

（经典问题）用前序遍历与中序遍历结果重构二叉树，分析重构条件及算法的时间复杂度



# 5.5.1 最优二叉树

**带权二叉树**：每个叶结点带有一个权重值（正数）的二叉树




带有相同权重集 {1, 2, 3, 4 ,5} 的三棵带权二叉树

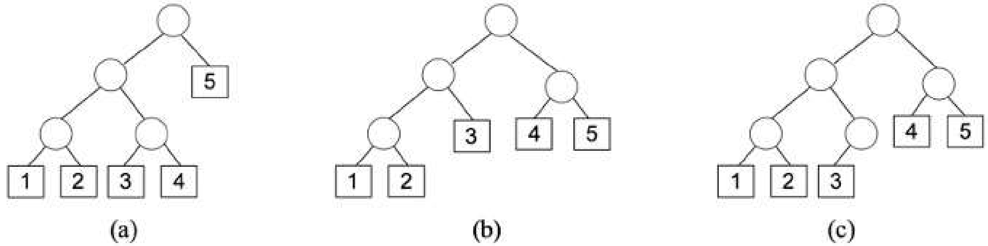
**叶结点的带权路径长度**：等于$w_i l_i$，其中$w_i$是叶结点的权重， $l_i$是从根到叶结点的路径长度，即从根到该结点经过的分支数

**树的带权路径长度**：树中所有叶结点的带权路径长度之和

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

# 5.5.1 最优二叉树

**带权二叉树**：每个叶结点带有一个权重值（正数）的二叉树




带有相同权重集 {1, 2, 3, 4 ,5} 的三棵带权二叉树

$$\mathrm{WPL}(a) = 1\times3 + 2\times3 + 3\times3 + 4\times3 + 5\times1 = 35$$

$$\mathrm{WPL}(b) = 1\times3 + 2\times3 + 3\times2 + 4\times2 + 5\times2 = 33$$

最小值

$$\mathrm{WPL}(c) = 1\times3 + 2\times3 + 3\times3 + 4\times2 + 5\times2 = 36$$

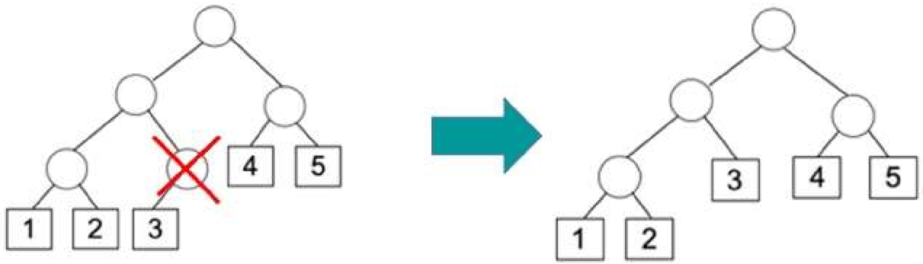
**最优二叉树**：给定一组叶结点权重，由此构建的所有带权二叉树中，带权路径长度最小的二叉树，亦称**哈夫曼树**

高等教育出版社

# 最优二叉树基本性质

**定理5-4**：最优二叉树（哈夫曼树）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

# 最优二叉树基本性质

**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

高等教育出版社

# 最优二叉树基本性质

**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

# 哈夫曼算法

**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

**哈夫曼算法**：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个只有一个结点（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;

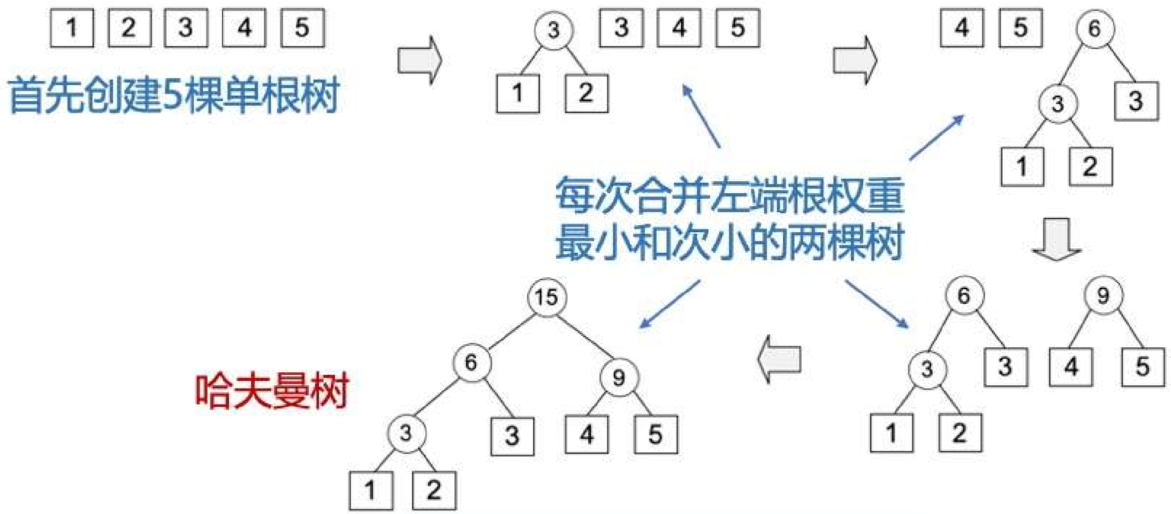
(3) 把(2)构成的新二叉树插入二叉树集T中;

(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哈夫曼树。

高等教育出版社

# 哈夫曼算法

**哈夫曼算法**：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




最优二叉树的构建过程

## 算法5-15：创建哈夫曼树 CreateHuffmanTree($w$)

输入：权重值的数据集$w$，$|w| \geq 2$
输出：哈夫曼树

```
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$T_{Q_Insert}$

## 算法5-15：创建哈夫曼树 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，即哈夫曼树
return tree
```

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

# 哈夫曼算法

**哈夫曼算法**：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，因此构建哈夫曼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。

# 哈夫曼树的应用：哈夫曼编码

**问题**：如何传输由字母a、b、c、w、z组成的字符串“baaacabwbzc”？

**关键**：需要将文字字符串转换成计算机能够识别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

baaacabwbzc

定长码：a–000, b–001, c–010, w–011, z–100

编码

001000000000010000001011001100010 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)

不定长码：a–01, b–00, c–10, w–110, z–111

编码

000101011001001100011110 长度24

不定长码比定长码的编码效率高！

高等教育出版社

# 哈夫曼编码

**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

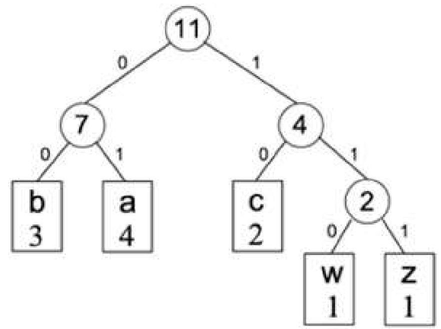
对二进制字符串解码会出现歧义

例：”1101” → “wa”

“1101” → “aca”

a–01, b–00, c–10, w–110, z–111

前缀码




可用二叉树表示前缀码

# 哈夫曼编码

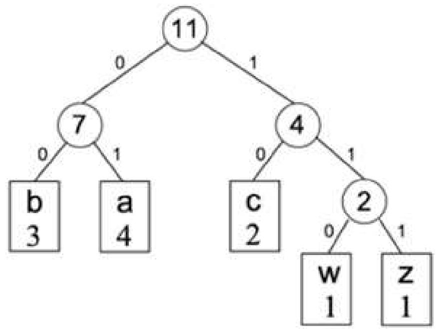
**前缀码**：常用的不定长码，每个字母的编码都不是其它字母编码的前缀

**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

# 哈夫曼编码

**前缀码**：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

**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

前缀码树的带权路径长度WPL

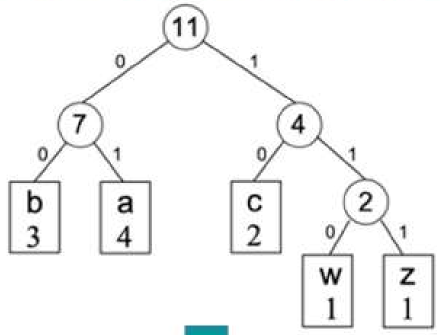
‖

字符串的编码长度

↓

最优前缀码可用哈夫曼算法求解，并把求得的前缀码称为哈夫曼编码

对应权重集{3,4,2,1,1}的哈夫曼树




a–01, b–00, c–10, w–110, z–111

是字符串“baaacabwbzc”的哈夫曼编码

## 算法5-16：使用哈夫曼树对二进制字符串解码 Decoding(*tree*, *binary_code*)

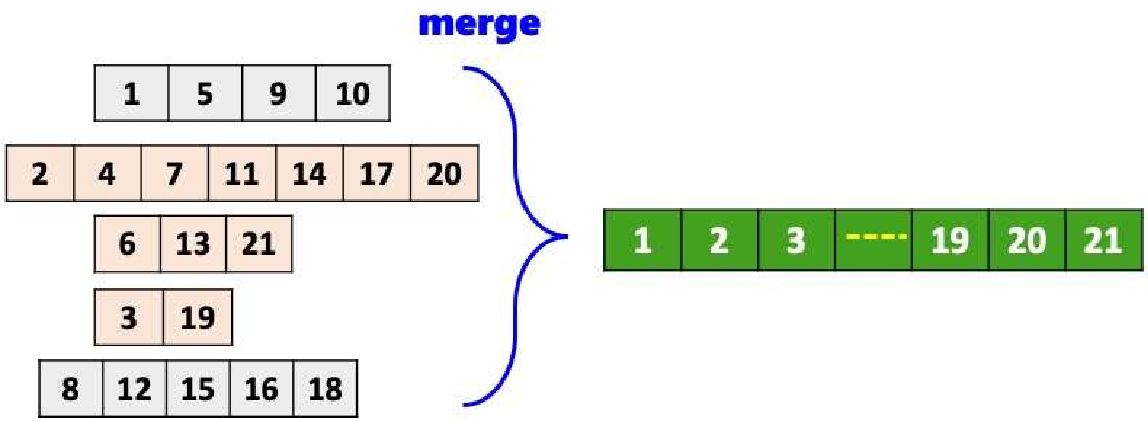
输入：非空前缀码树*tree*，二进制字符串*binary_code*
输出：解码后的文字符序列

```
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，使用哈夫曼（前缀码）树只需要O(n) 的时间就能解码生成原来的文字串

# 哈夫曼树的应用：多路合并

**问题**：把n个升序序列: $A_1, A_2, \ldots, \mathrm{A_n}$，合并成一个升序序列。简单起见，假设合并长度为x和y的两个序列需要x+y次比较，求比较次数最少的合并方案。



# 哈夫曼树的应用：多路合并

**问题**：把n个升序序列: $A_1, A_2, \ldots, A_n$，合并成一个升序序列。简单起见，假设合并长度为x和y的两个序列需要x+y次比较，求比较次数最少的合并方案。

方案一：同步归并

1. 查找所有（非空）序列先头元素的最小值
2. 从序列前端取出最小值并添加到合并序列的末尾
3. 重复上述操作，直到所有序列变空为止


## 时间复杂度

- 从n个序列的先头元素中查找最小值的时间为O(n)
- 合并时间为O(nN) ($N = |A_1| + |A_2| + \cdots + |A_n|$)
- 思考：如何快速查找先头元素的最小值?

# 哈夫曼树的应用：多路合并

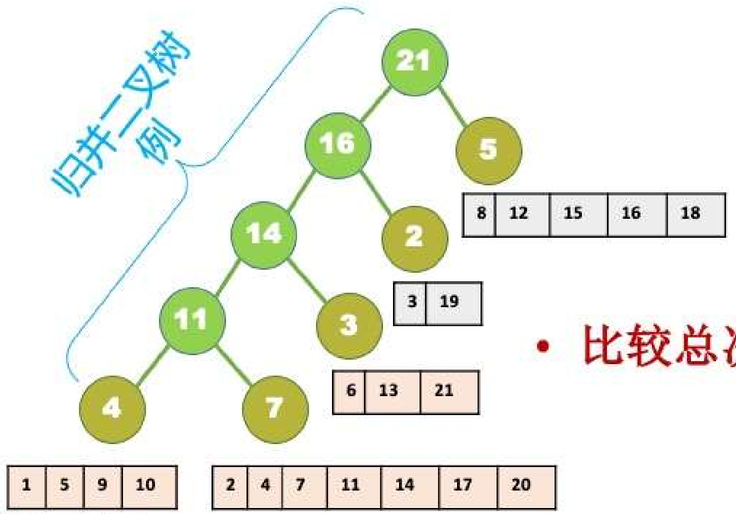
**问题**：把n个升序序列: $A_1, A_2, \ldots, A_n$，合并成一个升序序列。简单起见，假设合并长度为x和y的两个序列需要x+y次比较，求合并次数最少的合并方案。

- 方案二：按序合并（两两合并）

如果序列数量大于1，选出两个序并合并成一个序列。合并过程可用二叉树表达

## 归并二叉树特征

- 含n个叶结点的满二叉树
- 每个结点代表一个有序序列，根结点表示所有序列合并成的序列；结点的权重表示其对应的有序序列的长度
- 每个中间结点对应的序列由其两个子结点分别对应的序列合并而来，因此权重等于合并两个子序列的比较次数
- 合并所有序列的比较次数等于归并二叉树中所有中间结点的权重和




- 比较总次数：62

# 哈夫曼树的应用：多路合并

**问题**：把n个升序序列: $A_1, A_2, \ldots, A_n$，合并成一个升序序列。简单起见，假设合并长度为x和y的两个序列需要x+y次比较，求合并次数最少的合并方案。

- 方案二：按序合并（两两合并）

如果序列数量大于1，选出两个序并合并成一个序列。合并过程可用二叉树表达

**归并二叉树特征**

- 含n个叶结点的满二叉树
- 每个结点代表一个有序序列，根结点表示所有序列合并成的序列；结点的权重表示其对应的有序序列的长度
- 每个中间结点对应的序列由其两个子结点分别对应的序列合并而来，因此权重等于合并两个子序列的比较次数
- 合并所有序列的比较次数等于归并二叉树中所有中间结点的权重和

可以证明（数学归纳法？），归并二叉树中所有中间结点的权重和等于其所有叶结点的带权路径长度和。即二叉树的WPL

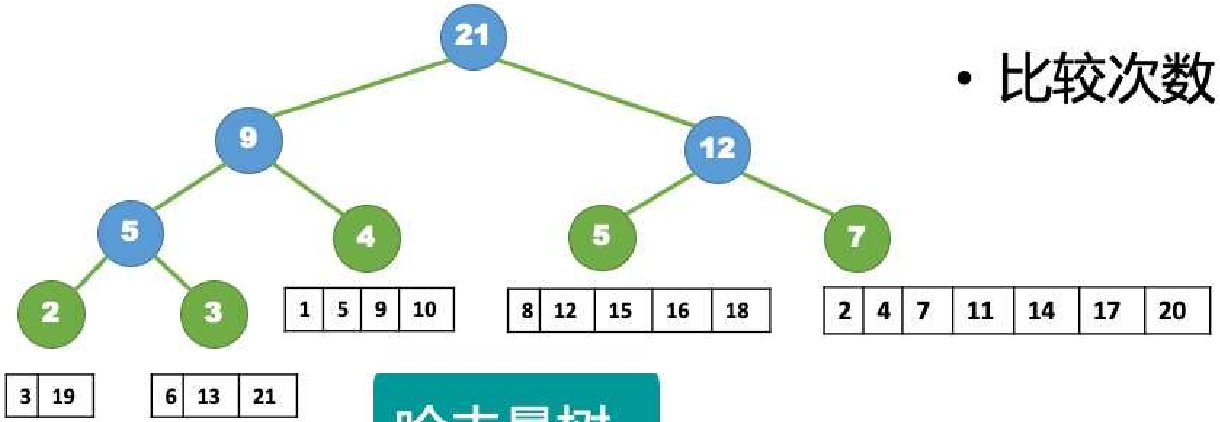
可用哈夫曼算法求最优合并顺序

# 哈夫曼树的应用：多路合并

**问题**：把n个升序序列: $A_1, A_2, \dots, A_n$，合并成一个升序序列。简单起见，假设合并长度为x和y的两个序列需要x+y次比较，求比较次数最少的合并方案。

- 方案二：按序合并（两两合并）

如果序列数量大于1，选出两个序并合并成一个序列。合并过程可用二叉树表达



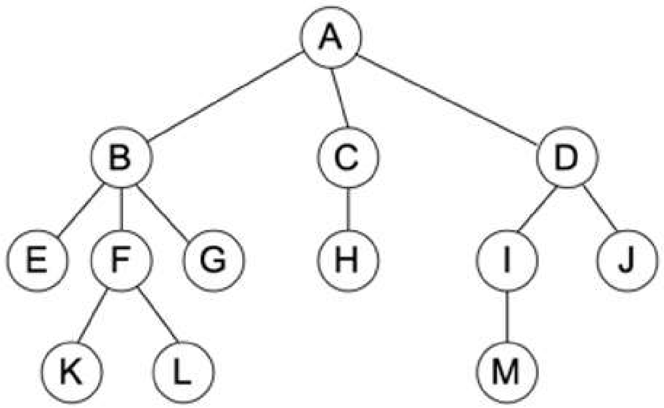

- 比较次数：47

哈夫曼树

# 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

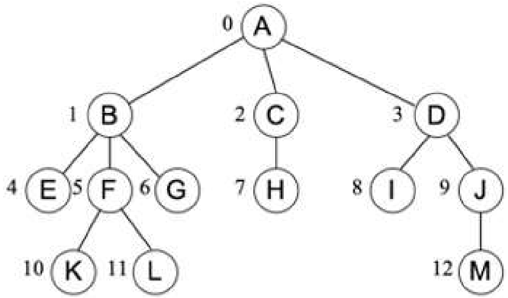
常用的树逻辑结构表示法：**父亲表示法**、**孩子表示法**以及**孩子兄弟表示法**



# 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