数据结构与算法(五)

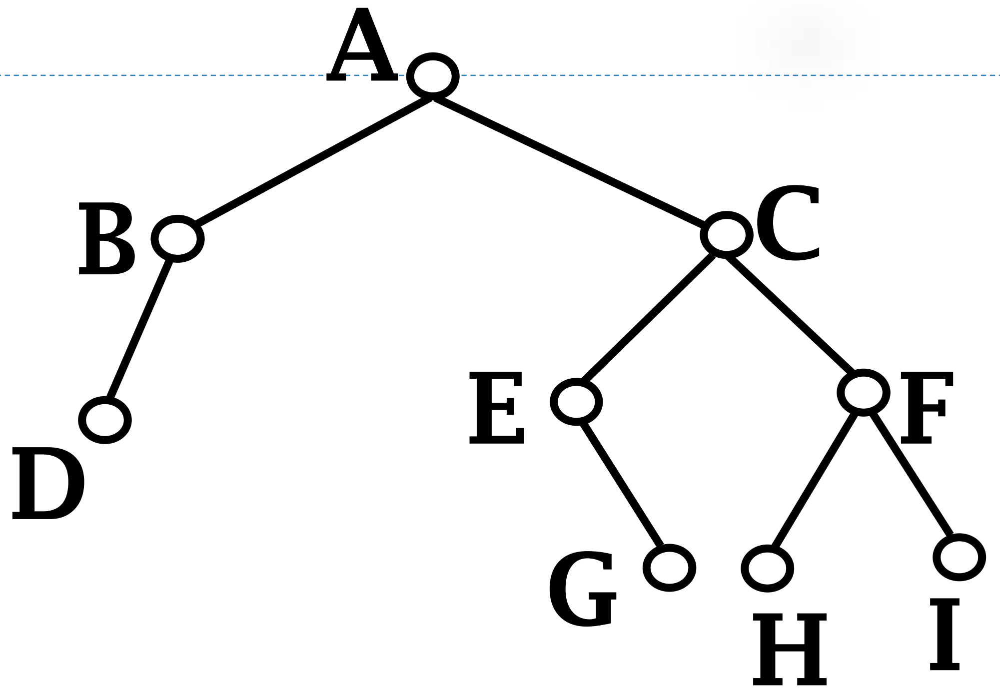
张铭 主讲

采用教材：张铭，王腾蛟，赵海燕 编写
高等教育出版社，2008. 6 （“十一五”国家级规划教材）

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

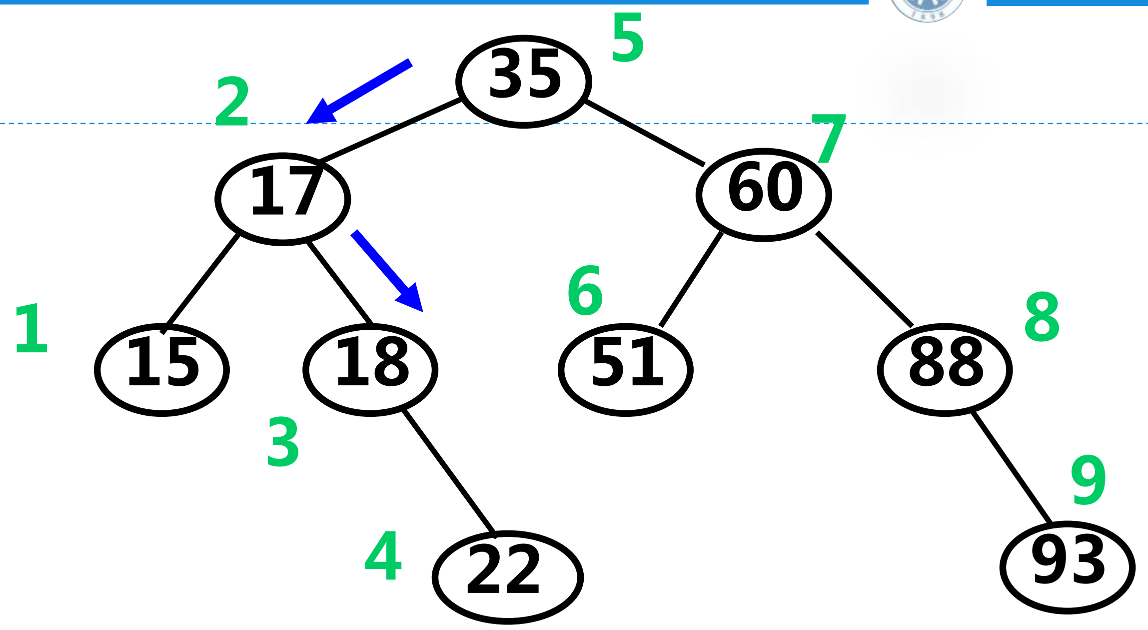
第五章 二叉树

- 二叉树的概念
- 二叉树的抽象数据类型
  - 深度优先搜索
  - 宽度优先搜索
- 二叉树的存储结构
- 二叉搜索树
- 堆与优先队列
- Huffman树及其应用



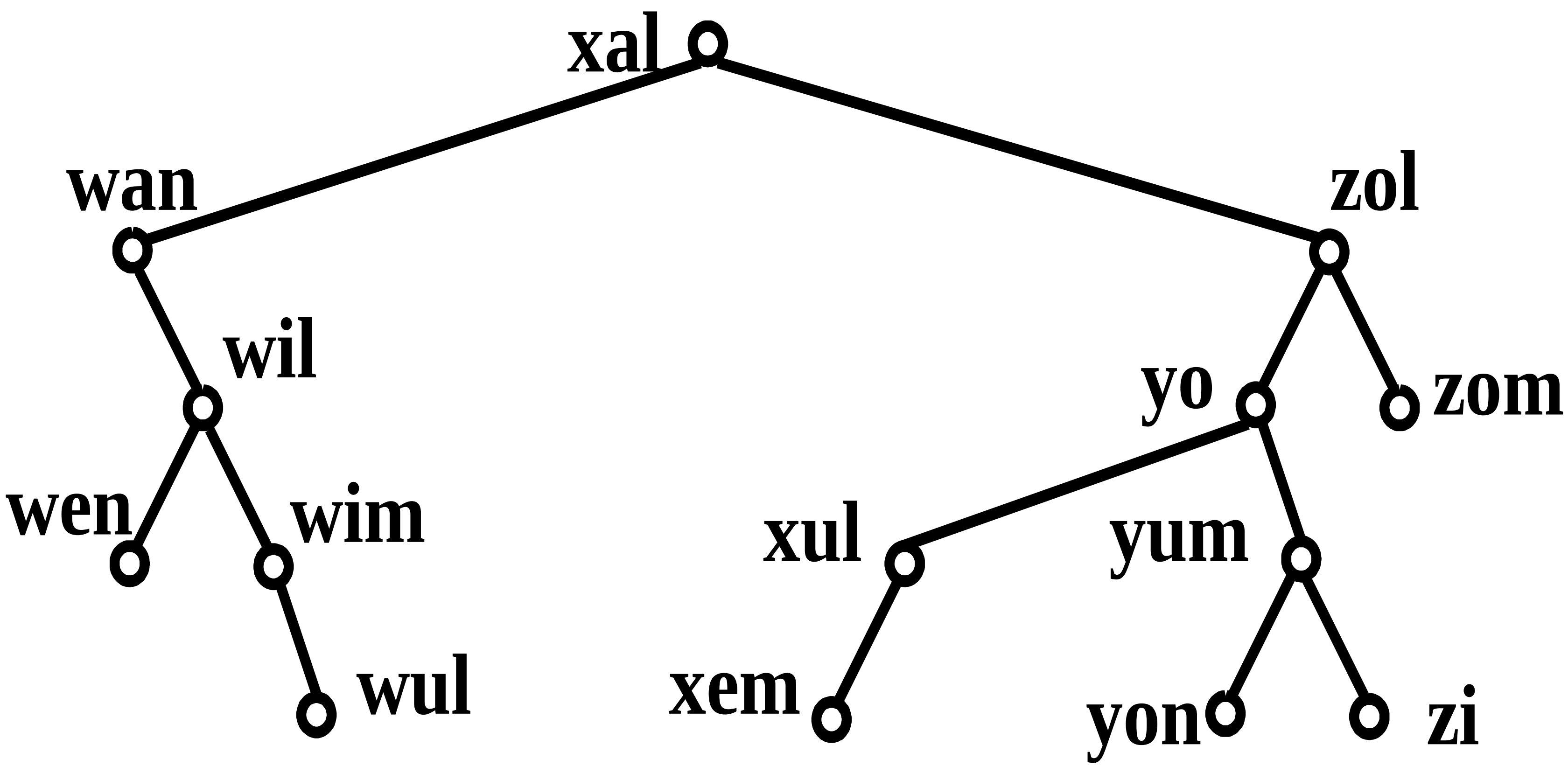
5.4 二叉搜索树

二叉搜索树

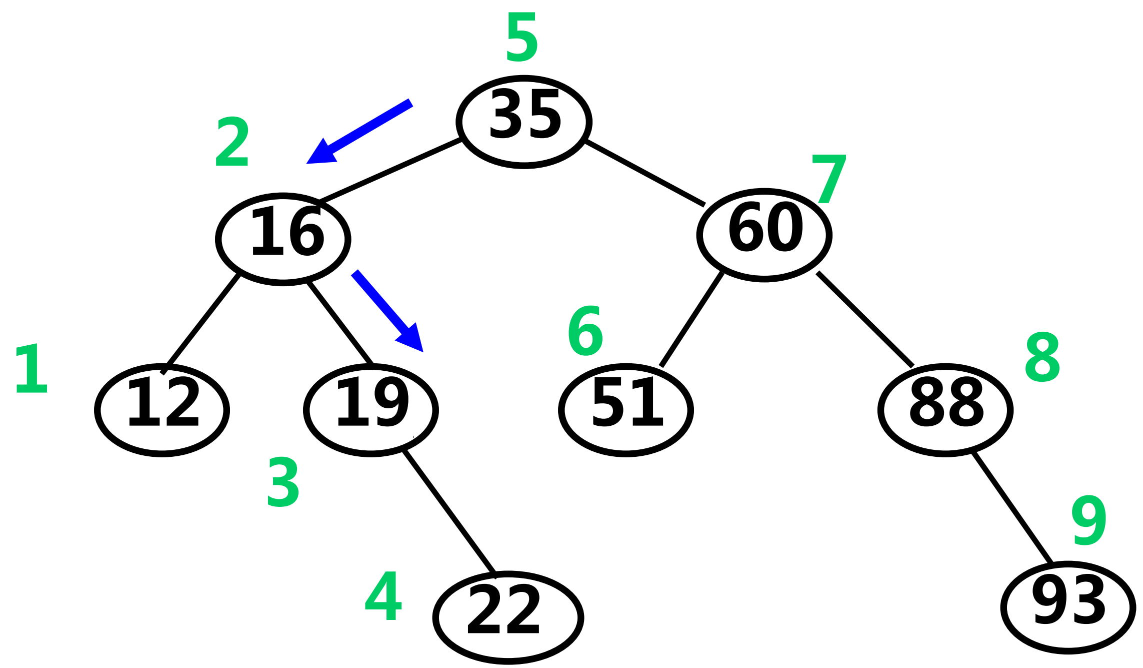


- Binary Search Tree（**BST**）
  - 或者是一棵空树；
  - 或者是具有下列性质的二叉树：
    - 对于任何一个结点，设其值为K
    - 则该结点的 左子树(若不空)的任意一个结点的值都 小于 K；
    - 该结点的 右子树(若不空)的任意一个结点的值都 大于 K；
    - 而且它的左右子树也分别为BST
- 性质: 中序遍历是正序的（由小到大的排列）

BST示意图



检索 19

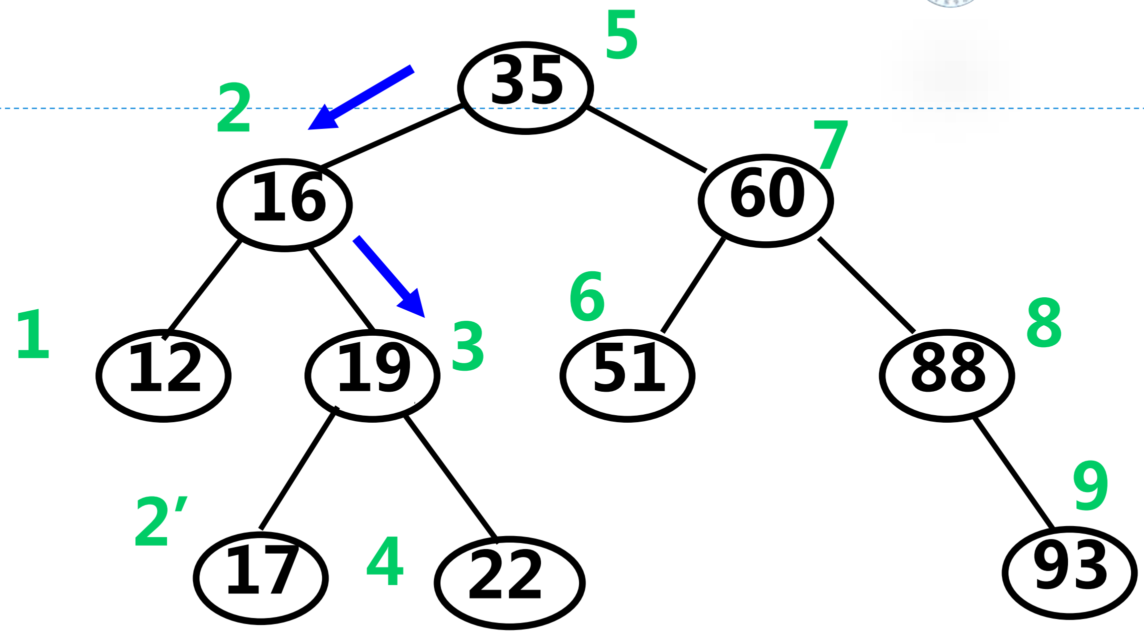


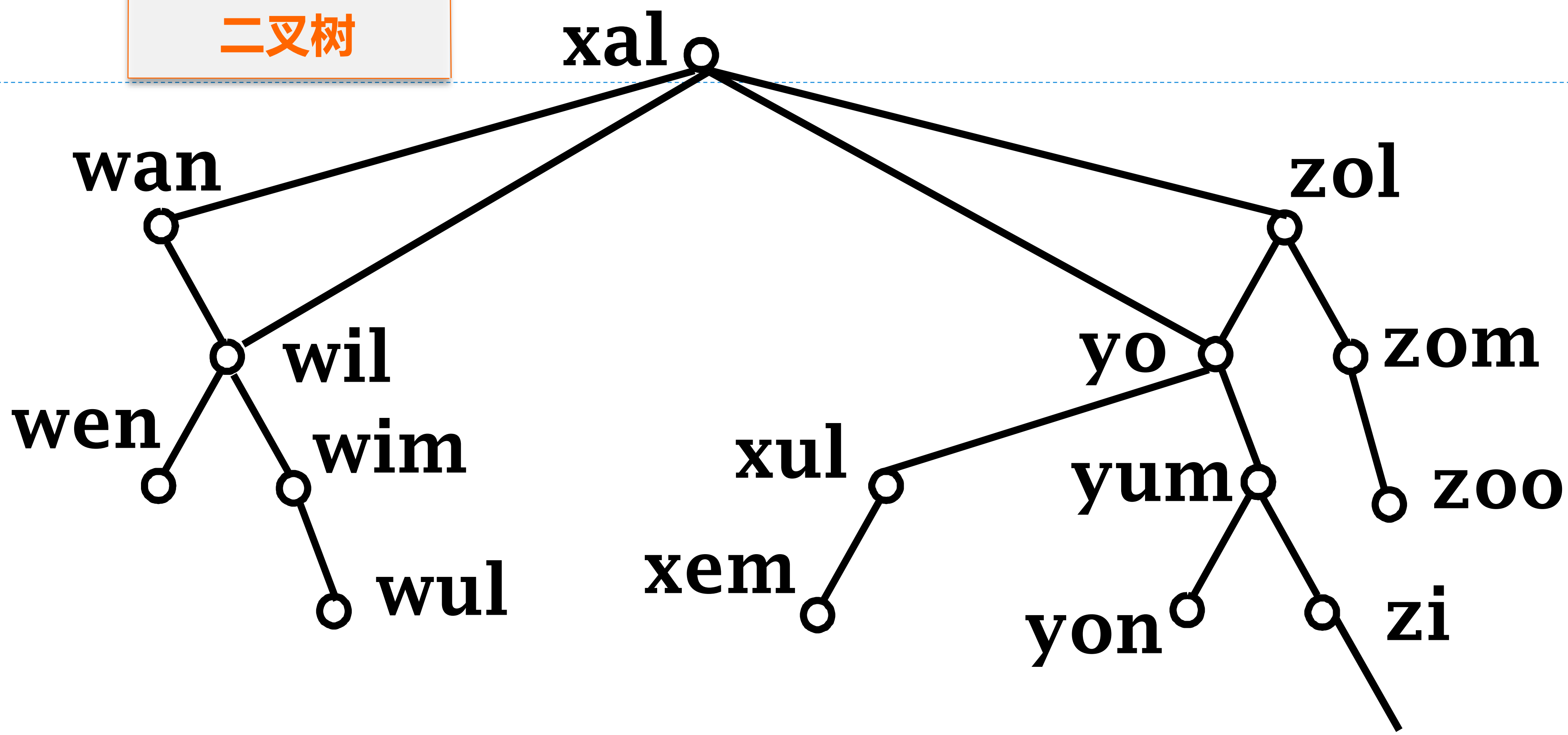
- 只需检索二个子树之一
  - 直到 K 被找到
  - 或遇上树叶仍找不到，则不存在

5.4 二叉搜索树

插入17

- 首先是检索，若找到则不允许插入
- 若失败，则在该位置插入一个新叶
- **保持BST性质和性能！**

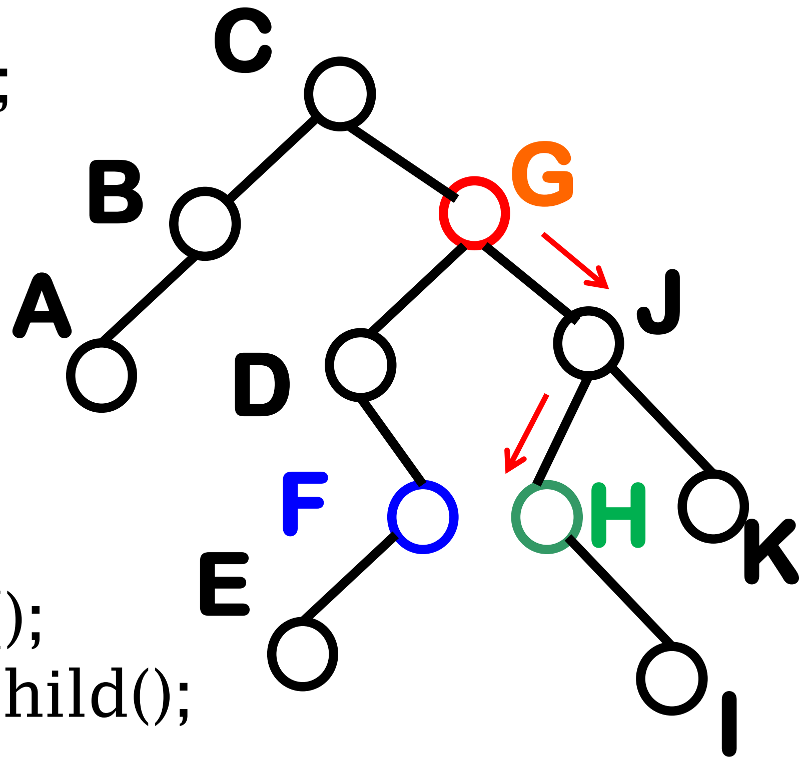




- 删除 wan
- 删除 zol

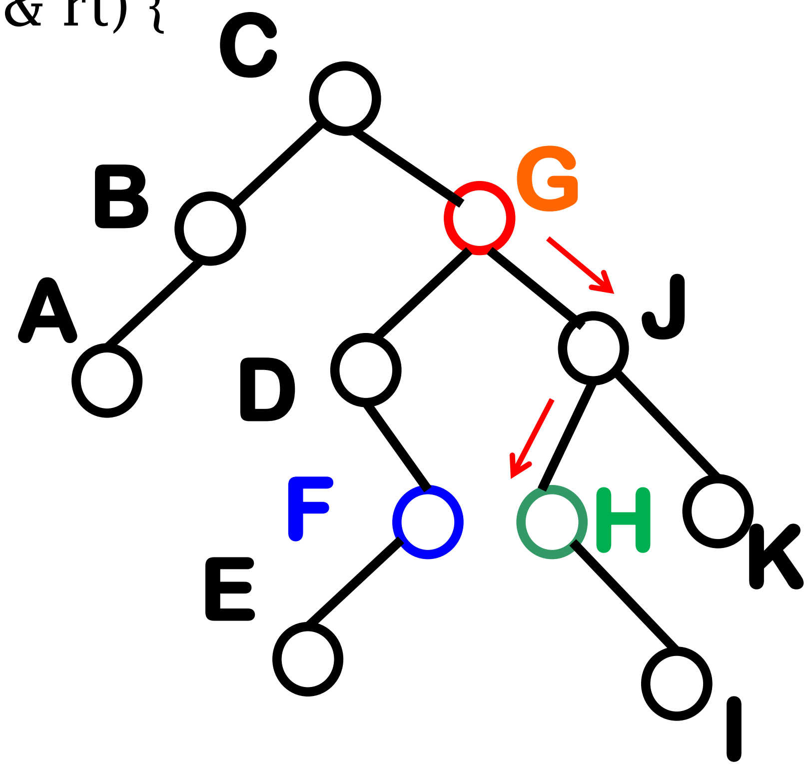
BST删除(值替换)

```
void BinarySearchTree<T>:::removehelp(BinaryTreeNode <T> *& rt,
const T val) {
  if (rt==NULL) cout<<val<<" is not in the tree.\n";
  else if (val < rt->value())
     removehelp(rt->leftchild(), val);
  else if (val > rt->value())
     removehelp(rt->rightchild(), val);
  else {                              // 真正的删除
     BinaryTreeNode <T> * temp = rt;
     if (rt->leftchild() == NULL)  rt = rt->rightchild();
     else if (rt->rightchild() == NULL)  rt = rt->leftchild();
     else {
        temp = deletemin(rt->rightchild());
        rt->setValue(temp->value());
     }
```



找rt右子树中最小结点，并删除

```
template <class T>
BinaryTreeNode* BST::deletemin(BinaryTreeNode <T> *& rt) {
  if (rt->leftchild() != NULL)
        return deletemin(rt->leftchild());
  else {           // 找到右子树中最小，删除
      BinaryTreeNode <T> *temp = rt;
      rt = rt->rightchild();
      return temp;
  }
}
```

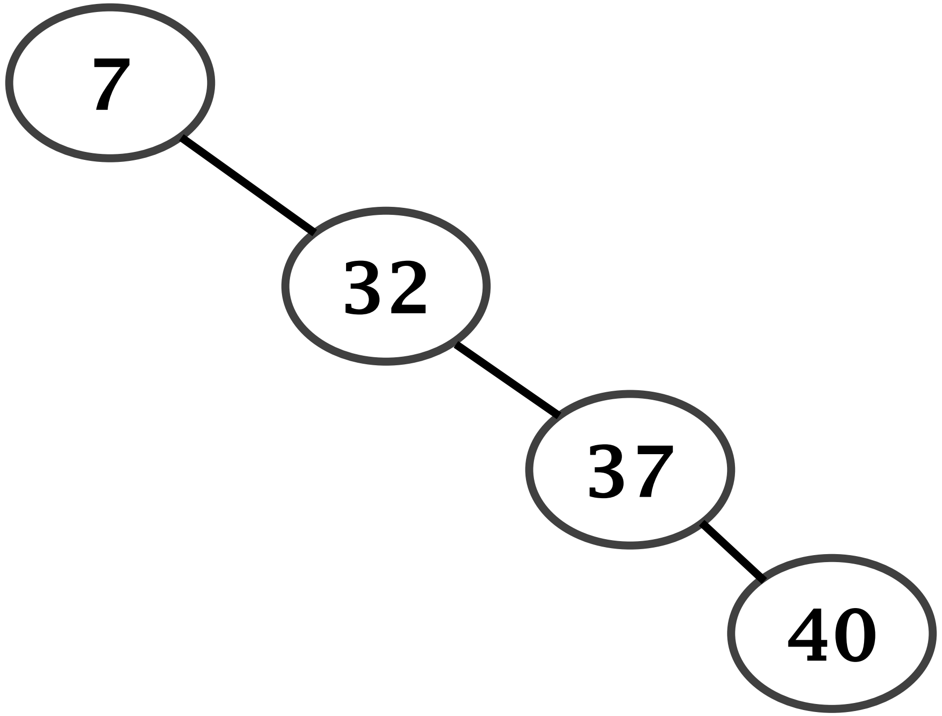


二叉搜索树总结

- 组织内存索引
  - 二叉搜索树是适用于内存储器的一种重要的树形索引
    - 常用红黑树、伸展树等，以维持平衡
  - 外存常用B/B+树
- 保持性质 vs 保持性能
  - 插入新结点或删除已有结点，要保证操作结束后仍符合二叉搜索树的定义

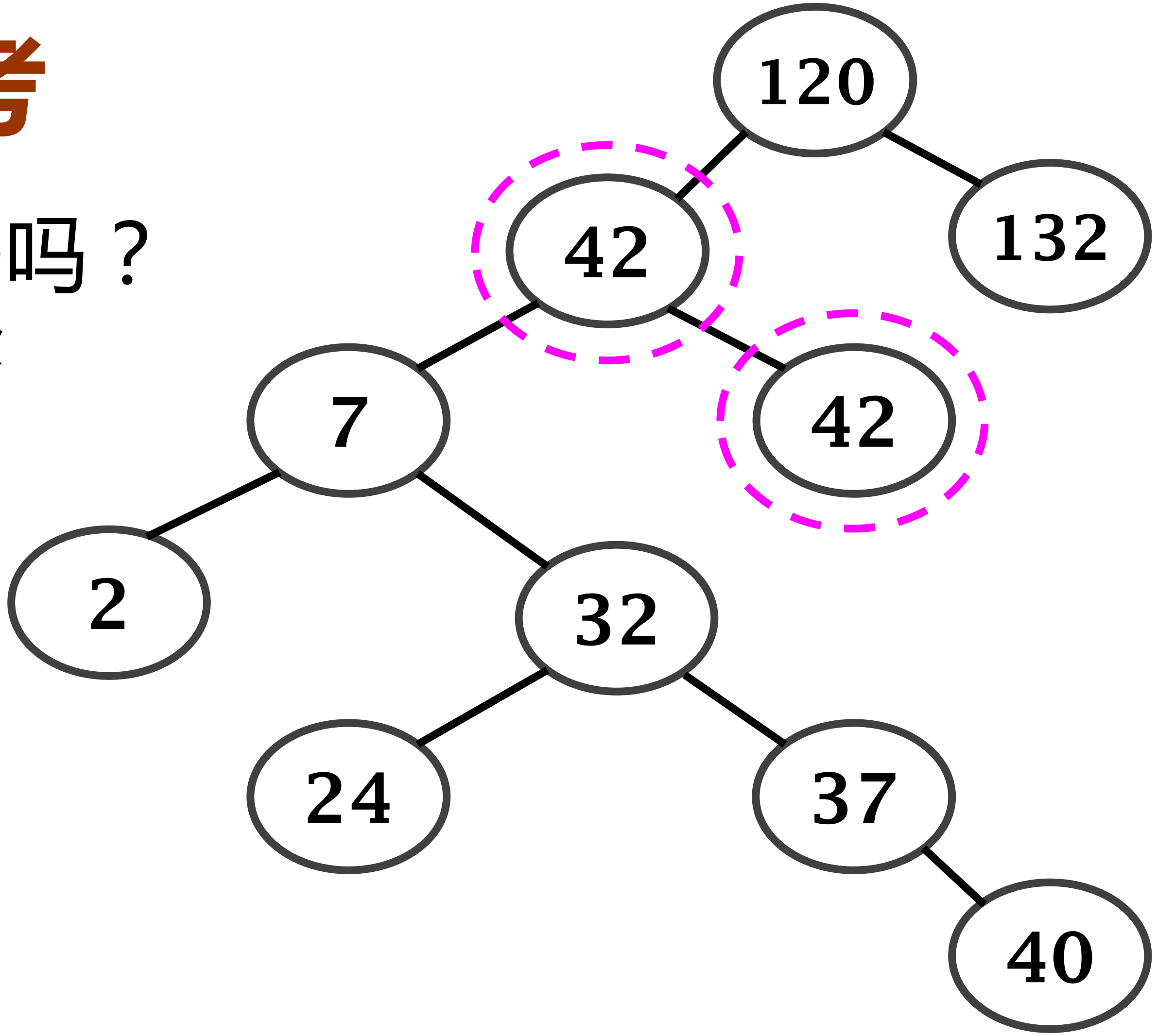
思考

- 怎样防止BST退化为线性结构？



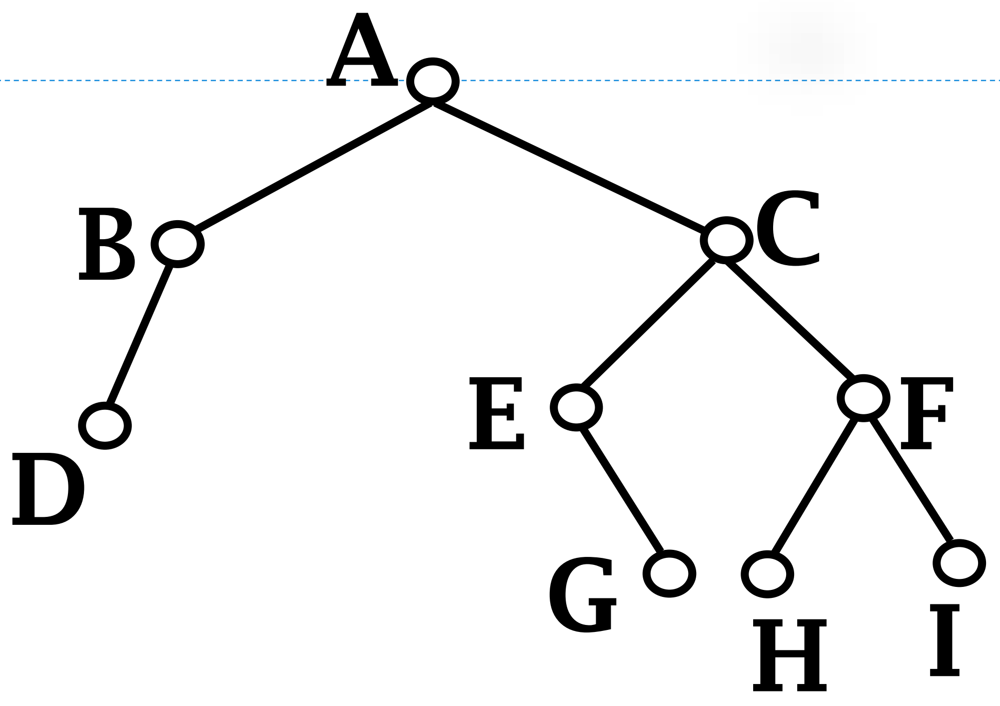
思 考

- 允许重复关键码吗？
  - 插入、检索、删除



第五章 二叉树

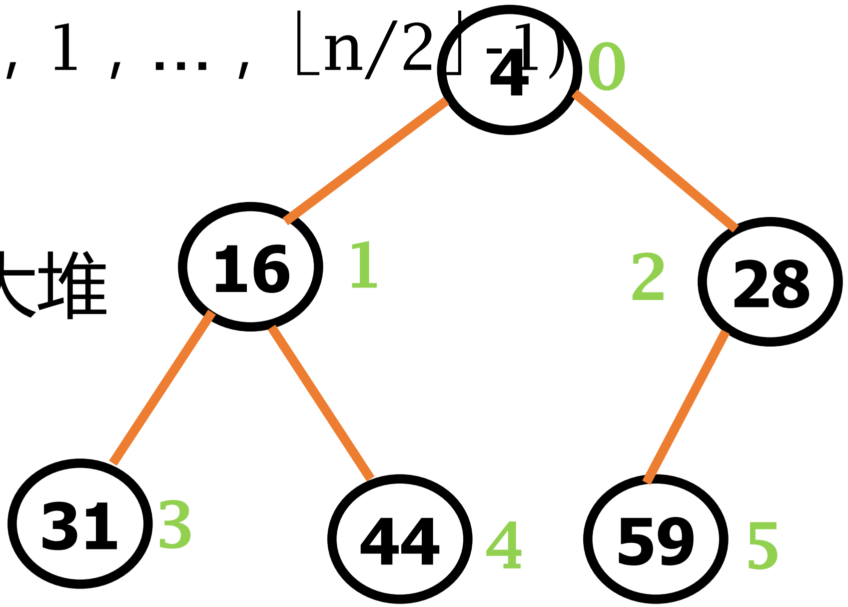
- 二叉树的概念
- 二叉树的抽象数据类型
  - 深度优先搜索
  - 宽度优先搜索
- 二叉树的存储结构
- 二叉搜索树
- 堆与优先队列
- Huffman树及其应用



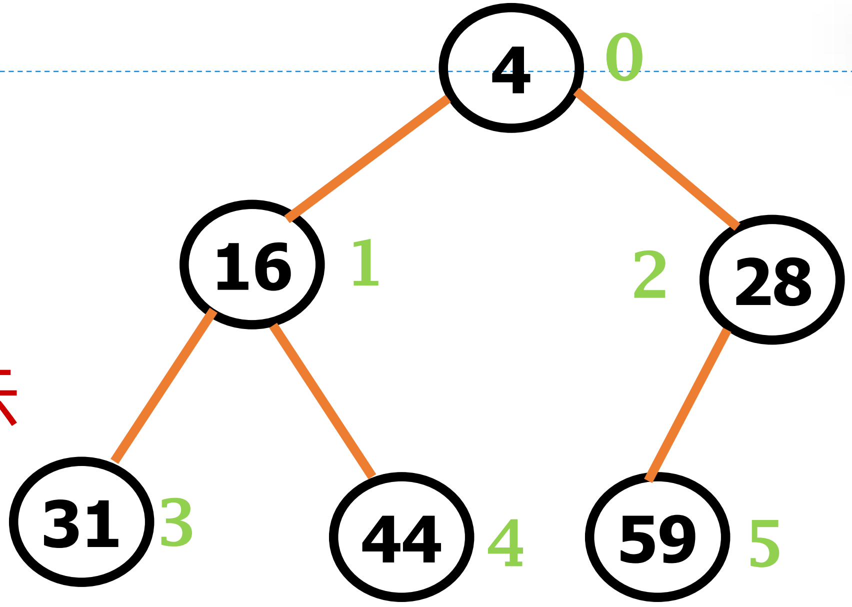
堆的定义及其实现

- 最小堆：最小堆是一个关键码序列

  $\{K_0, K_1, \ldots K_{n-1}\}$，它具有如下特性：
  - $K_i \le K_{2i+1}$ $(i=0, 1, \ldots, \lfloor n/2 \rfloor -1)$
  - $K_i \le K_{2i+2}$
- 类似可以定义最大堆



堆的性质



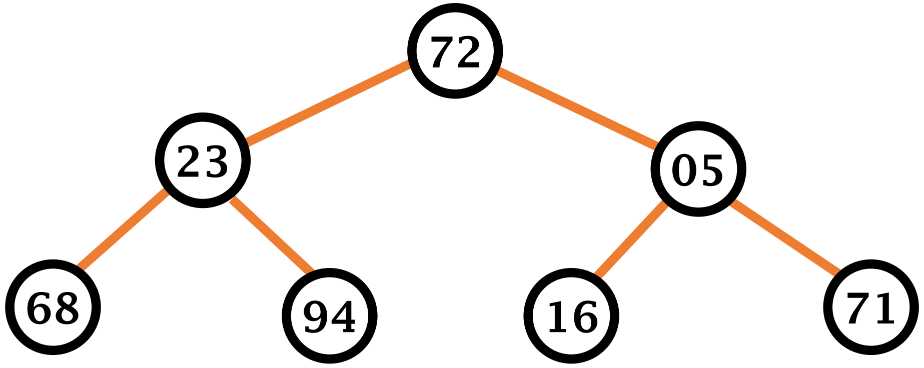
- 完全二叉树的层次序列，可以用数组表示
- 堆中储存的数是局部有序的，堆不唯一
  - 结点的值与其孩子的值之间存在限制
  - 任何一个结点与其兄弟之间都没有直接的限制
- 从逻辑角度看，堆实际上是一种树形结构

堆的类定义

```
template <class T>
class MinHeap  {                   // 最小堆ADT定义
private:
   T* heapArray;                   // 存放堆数据的数组
   int CurrentSize;             // 当前堆中元素数目
   int MaxSize;                    // 堆所能容纳的最大元素数目
   void BuildHeap();               // 建堆
public:
   MinHeap(const int n) ;                  // 构造函数,n为最大元素数目
   virtual ~MinHeap(){delete []heapArray;};   // 析构函数
   bool isLeaf(int pos) const;                // 如果是叶结点，返回TRUE
   int leftchild(int pos) const;              // 返回左孩子位置
   int rightchild(int pos) const;             // 返回右孩子位置
   int parent(int pos) const;                 // 返回父结点位置
   bool Remove(int pos, T& node);             // 删除给定下标的元素
   bool Insert(const T& newNode) ;            // 向堆中插入新元素newNode
   T& RemoveMin() ;                           // 从堆顶删除最小值
   void SiftUp(int position);                 // 从position向上开始调整，使序列成为堆
   void SiftDown(int left) ;                  // 筛选法函数，参数left表示开始处理的数组下标
}
```

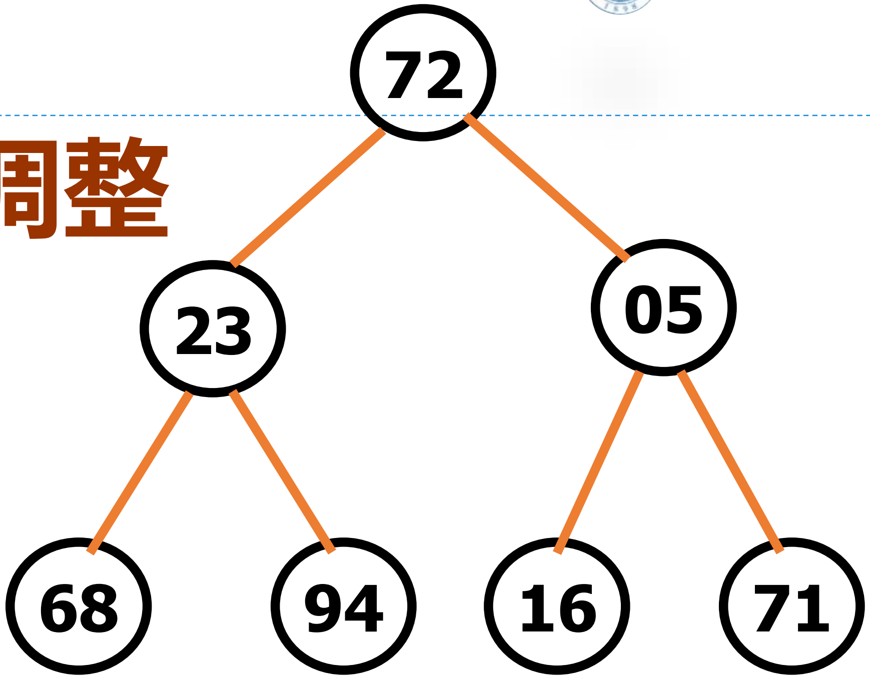
对最小堆用筛选法 SiftDown 调整

```
template <class T>
void MinHeap<T>::SiftDown(int position) {
    int i = position;              // 标识父结点
    int j = 2*i+1;                 // 标识关键值较小的子结点
    Ttemp = heapArray[i];    // 保存父结点
```



对最小堆用筛选法 SiftDown 调整

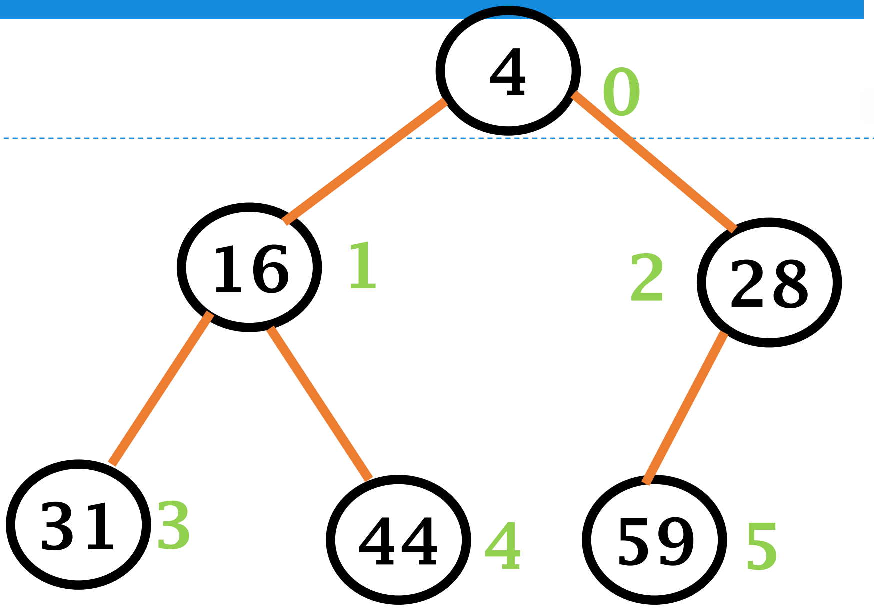
```
while (j < CurrentSize) {
    if((j < CurrentSize-1)&&
            (heapArray[j] > heapArray[j+1]))
        j++;        // j指向数值较小的子结点
    if (temp > heapArray[j]) {
     heapArray[i] = heapArray[j];
     i = j;
     j = 2*j + 1;        // 向下继续
    }
    else break;
}
heapArray[i]=temp;
}
```



对最小堆用筛选法 SiftUp 向上调整

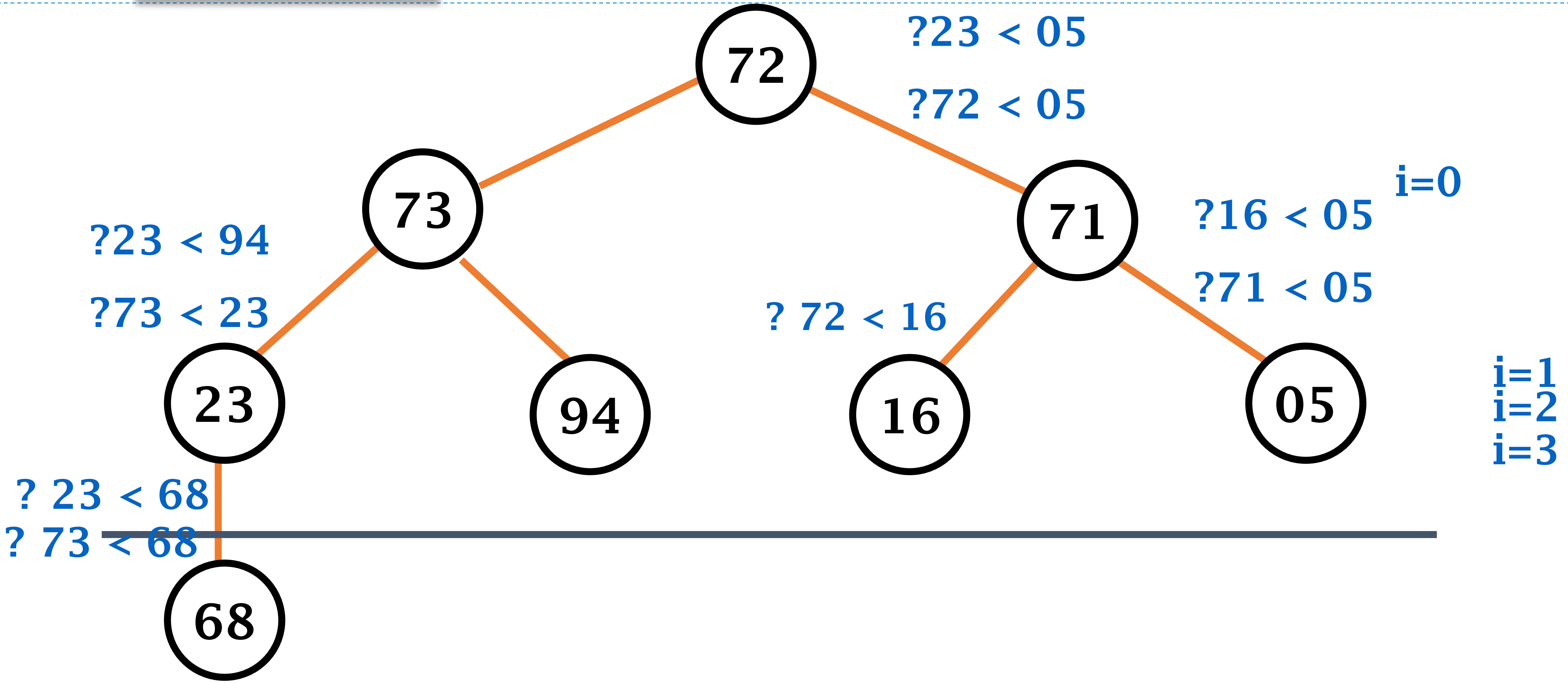
```
template<class T>
void MinHeap<T>::SiftUp(int position)  {
   // 从position向上开始调整，使序列成为堆
   int temppos=position;
   // 不是父子结点直接swap
   T temp=heapArray[temppos];
   while((temppos>0) && (heapArray[parent(temppos)] > temp))   {
          heapArray[temppos]=heapArray[parent(temppos)];
            temppos=parent(temppos);
   }
   heapArray[temppos]=temp;// 找到最终位置
}
```

建最小堆过程



- 首先，将 n 个关键码放到一维数组中
  - 整体不是最小堆
  - 所有叶结点子树本身是堆
    - 当 $i \geq \lfloor n/2 \rfloor$ 时，以关键码 K_i 为根的子树已经是堆
- 从倒数第二层，$i = \lfloor n/2 \rfloor - 1$ 开始从右至左依次调整
- 直到整个过程到达树根
  - 整棵完全二叉树就成为一个堆

5.5 堆与优先队列



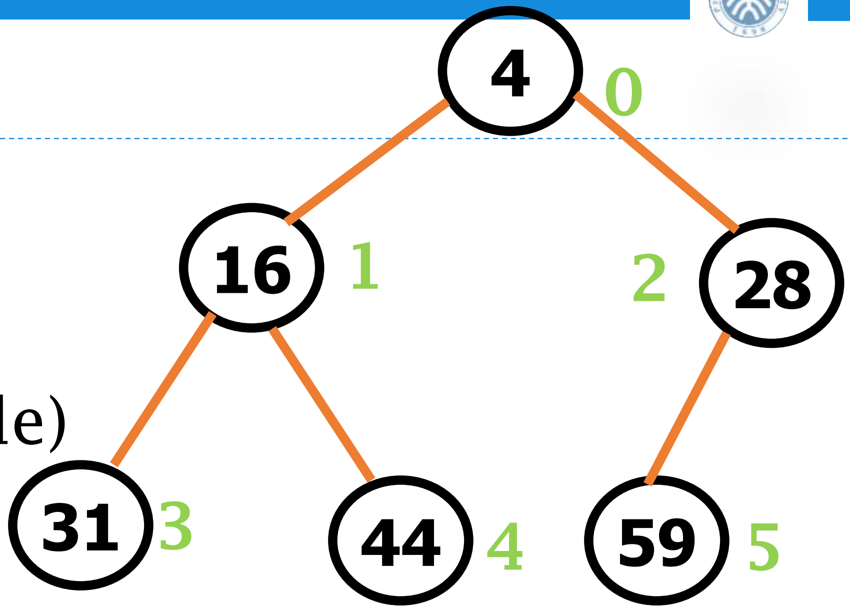
建最小堆过程示意图

建最小堆

从第一个分支结点 heapArray[CurrentSize/2-1] 开始，自底向上逐步把以子树调整成堆

```
template<class T>
void MinHeap<T>::BuildHeap()
{
    // 反复调用筛选函数
    for (int i=CurrentSize/2-1; i>=0; i--)
        SiftDown(i);
}
```

最小堆插入新元素

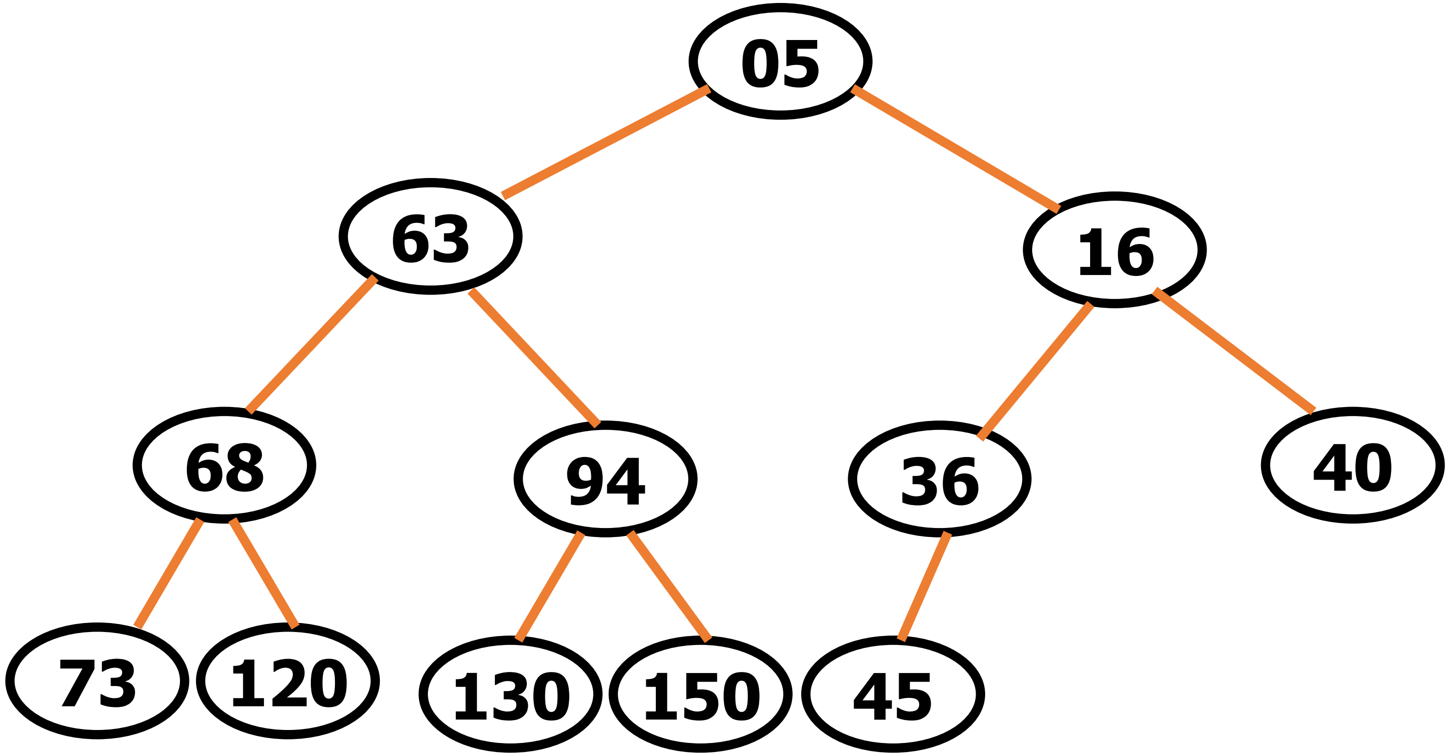



```
template <class T>
bool MinHeap<T>::Insert(const T& newNode)
//向堆中插入新元素newNode
{
    if(CurrentSize==MaxSize)   // 堆空间已经满
        return false;
    heapArray[CurrentSize]=newNode;
    SiftUp(CurrentSize);          // 向上调整
    CurrentSize++;
}
```

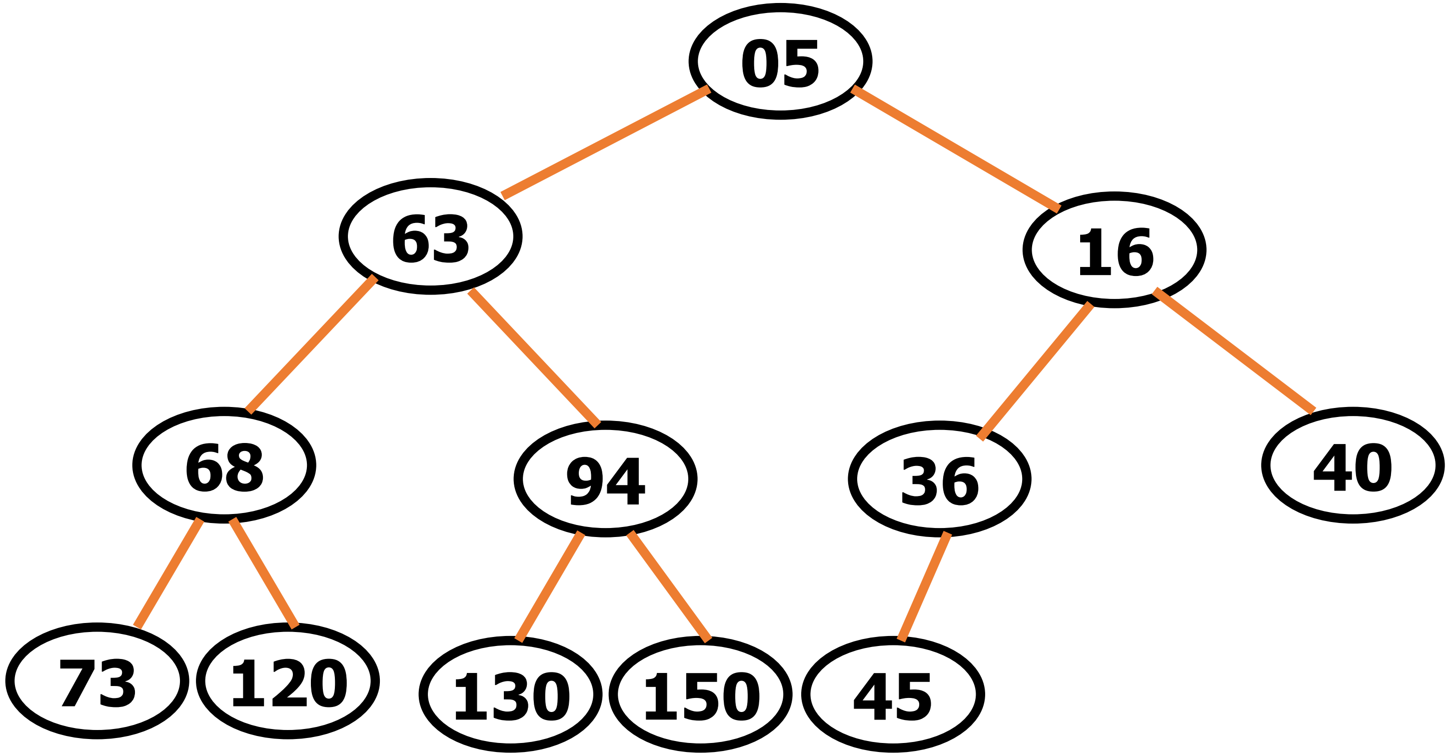
最小堆删除元素操作

```
template<class T>
bool MinHeap<T>::Remove(int pos, T& node)  {
    if((pos<0)||(pos>=CurrentSize))
        return false;
    T temp=heapArray[pos];
    heapArray[pos]=heapArray[--CurrentSize];
    if (heapArray[parent(pos)]> heapArray[pos])
        SiftUp(pos);  //上升筛
    else SiftDown(pos);   // 向下筛
    node=temp;
    return true;
}.
```

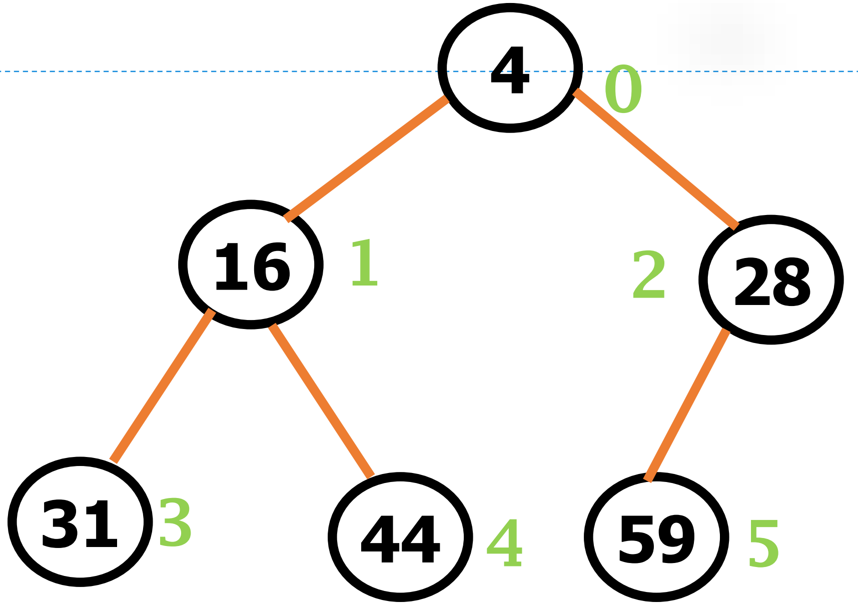
删除68



删除16



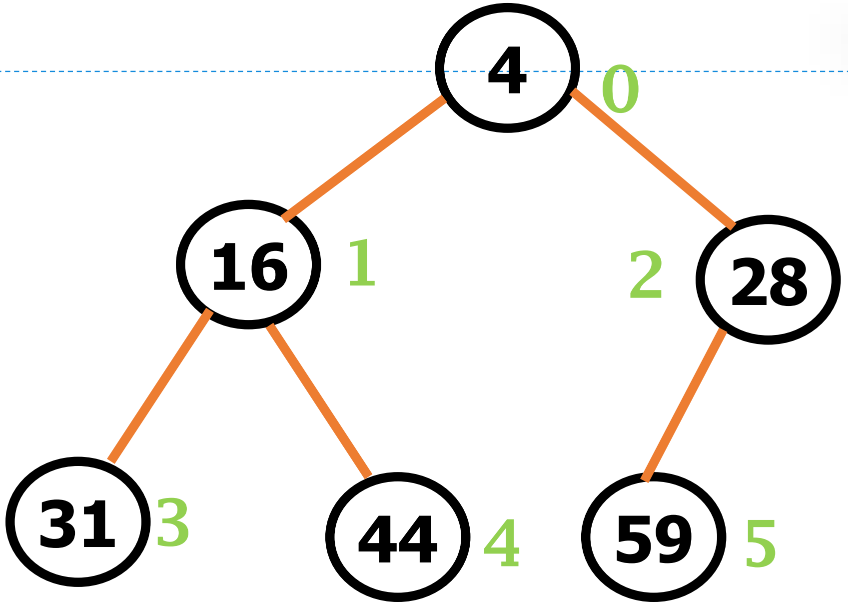
建堆效率分析



- n 个结点的堆，高度 $d = \lfloor \log_2 n + 1 \rfloor$。根为第 0 层，则第 i 层结点个数为 2^i，
- 考虑一个元素在堆中向下移动的距离。
  - 大约一半的结点深度为 d-1，不移动（叶）。
  - 四分之一的结点深度为 d-2，而它们至多能向下移动一层。
  - 树中每向上一层，结点的数目为前一层的一半，而子树高度加一。因而元素移动的最大距离的总数为

$$\sum_{i=1}^{\log n} (i-1)\frac{n}{2^i} = O(n)$$

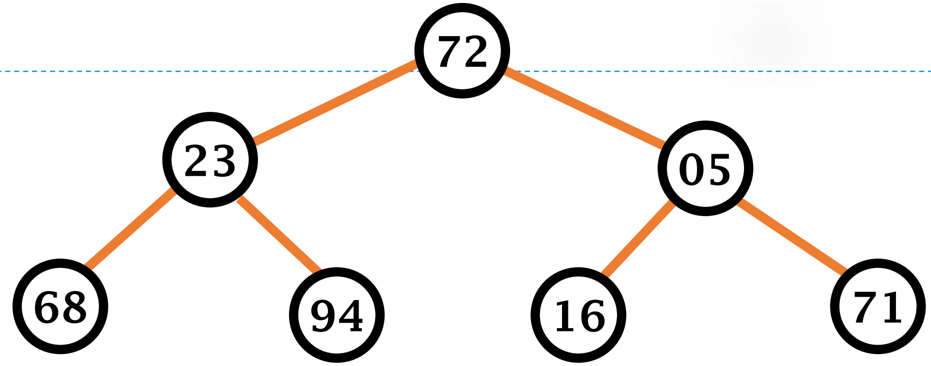
最小堆操作效率



- 建堆算法时间代价为 O(n)
- 堆有 log n 层深
  - 插入结点、删除普通元素和删除最小元素的平均时间代价和最差时间代价都是 O(log n)

优先队列

- 堆可以用于实现优先队列
- 优先队列
  - 根据需要释放具有最小（大）值的对象
  - 最大树、 左高树HBLT、WBLT、MaxWBLT
- 改变已存储于优先队列中对象的优先权
  - 辅助数据结构帮助找到对象

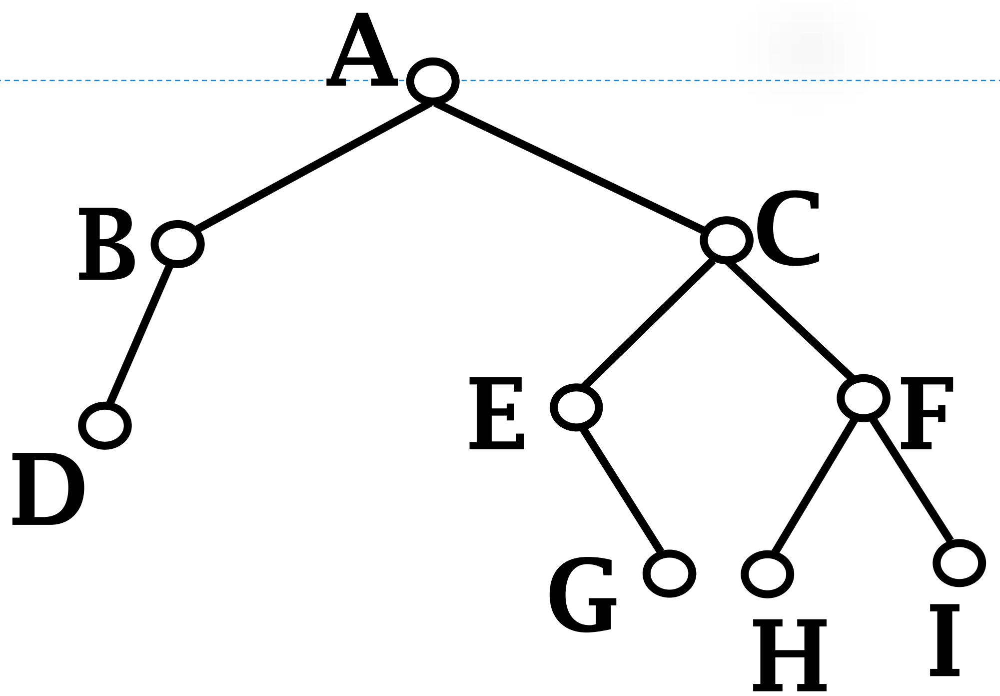


思考

- 在向下筛选SiftDown操作时，若一旦发现逆序对，就交换会怎么样？
- 能否在一个数据结构中同时维护最大值和最小值？（提示：最大最小堆）

第五章 二叉树

- 二叉树的概念
- 二叉树的抽象数据类型
    - 深度优先搜索
    - 宽度优先搜索
- 二叉树的存储结构
- 二叉搜索树
- 堆与优先队列
- Huffman树及其应用



等长编码

- 计算机二进制编码
  - ASCII 码
  - 中文编码
- 等长编码
  - 假设所有编码都等长
    表示 n 个不同的字符需要 $\log_2 n$位
  - 字符的使用频率相等
- 空间效率

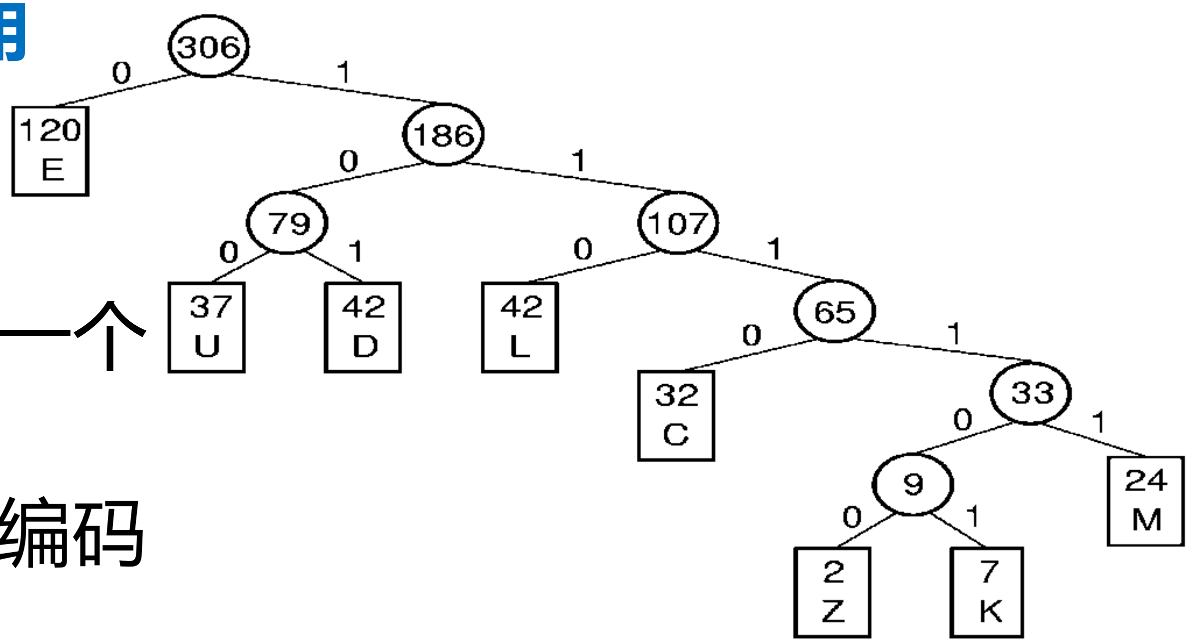
数据压缩和不等长编码

- 频率不等的字符

Z	K	F	C	U	D	L	E
2	7	24	32	37	42	42	120

- 可以利用字符的出现频率来编码
  - 经常出现的字符的编码较短，不常出现的字符编码较长
- 数据压缩既能节省磁盘空间，又能提高运算速度
（**外存时空权衡的规则**）

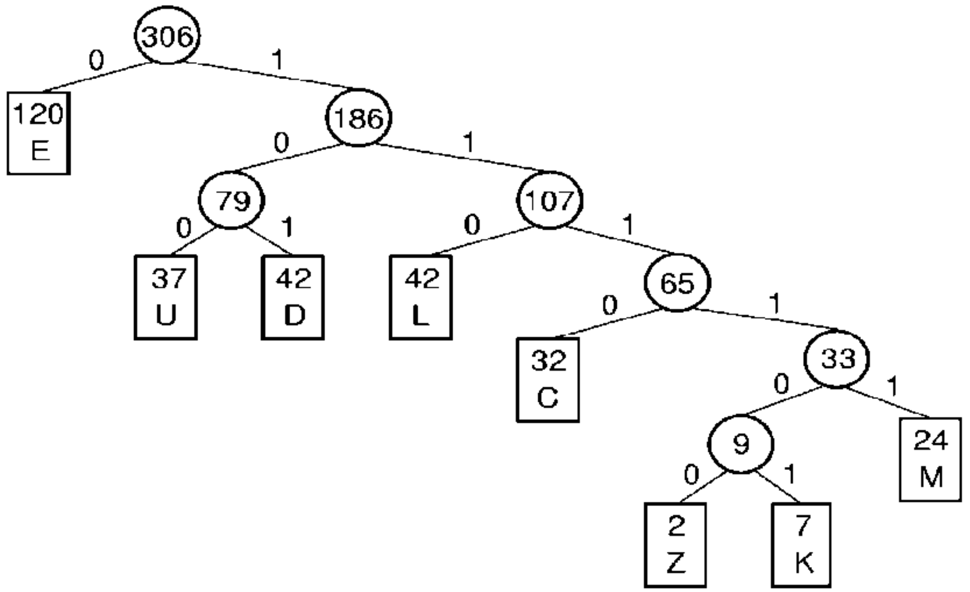
前缀编码



- 任何一个字符的编码都不是另外一个字符编码的前缀
- 这种前缀特性保证了代码串被反编码时，不会有多种可能。例如
  - 右图是一种前缀编码，对于 “000110” ，可以翻译出唯一的字符串 “EEEL” 。
  - 若编码为Z(00), K(01), F(11), C(0), U(1), D(10), L(110), E(010) 。 则对应：” ZKD” ,” CCCUUC” 等多种可能
- 编码 Z(111100), K(111101), F(11111), C(1110), U(100), D(101), L(110), E(0)

Huffman树与前缀编码

- Huffman编码将代码与字符相联系
  - 不等长编码
  - 代码长度取决于对应字符的相对使用频率或“权”



建立Huffman编码树

- 对于n个字符K_0，K_1，…，K_{n-1}，它们的使用频率分别为w_0, w_1，…，w_{n-1}，给出它们的前缀编码，使得总编码效率最高

- 给出一个具有n个外部结点的扩充二叉树
  - 该二叉树每个外部结点 K_i 有一个权 w_i 外部路径长度为 l_i
  - **这个扩充二叉树的叶结点带权外部路径长度总和**

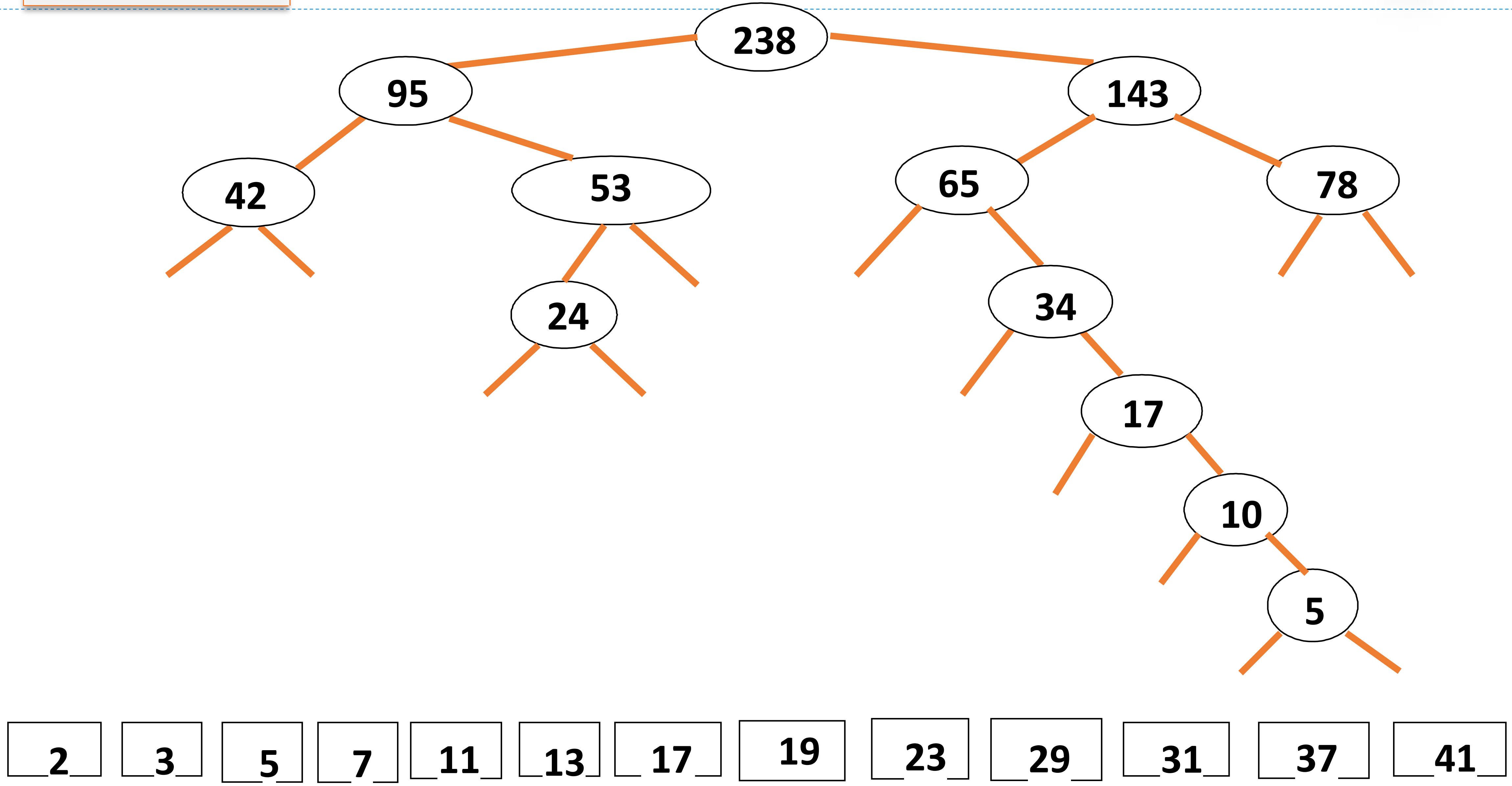
$$\sum_{i=0}^{n-1} w_i \cdot l_i$$

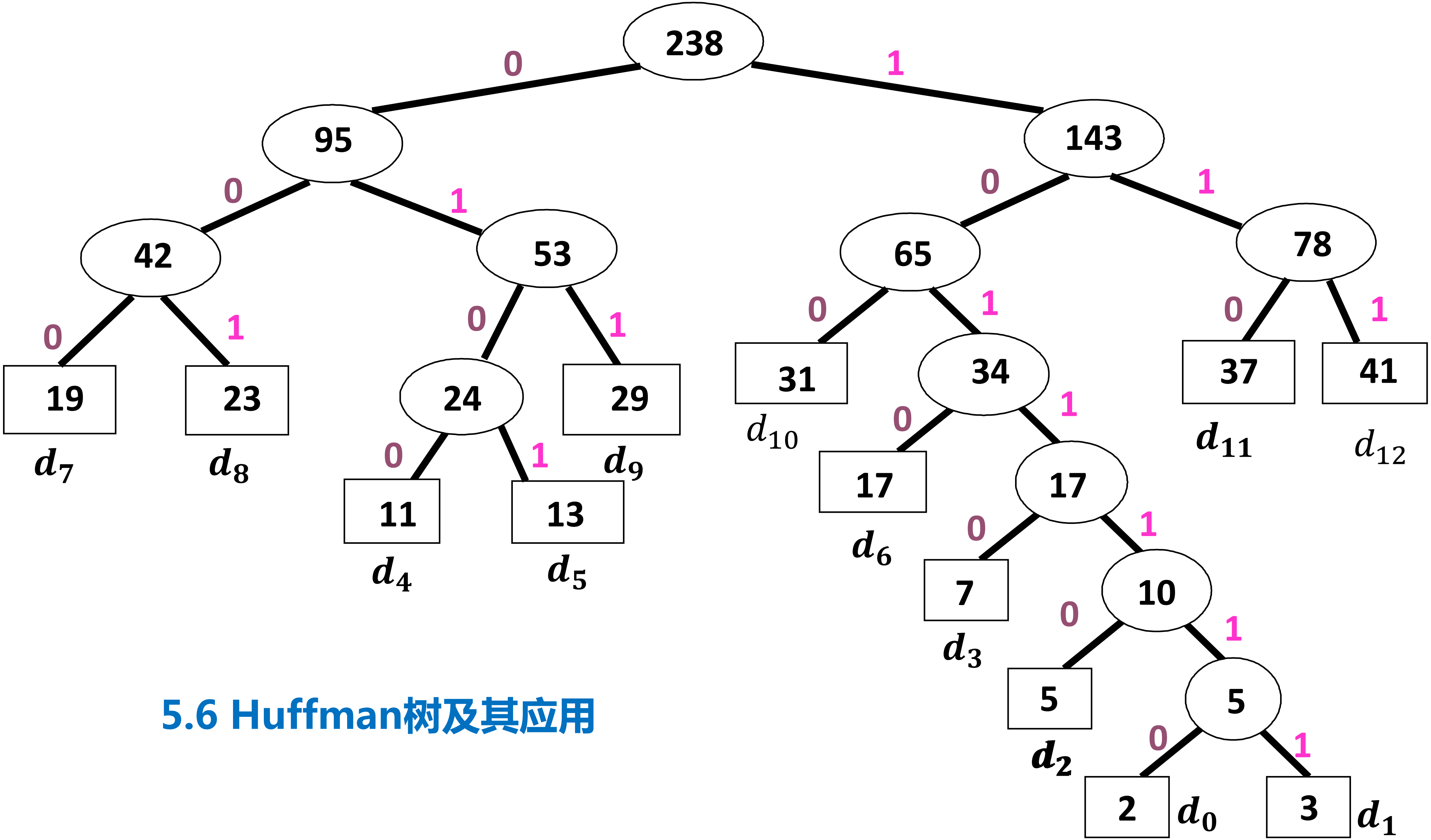
  - **权越大的叶结点离根越近**

建立Huffman编码树

- 首先，按照“权”(例如频率)将字符排为一列
  - 接着，拿走**前两个字符**(“权”最小的两个字符)
  - 再将它们标记为Huffman树的树叶，将这两个树叶标为一个分支结点的两个孩子，而该结点的权即为两树叶的权之和
- 将所得“权”放回序列，使“权”的顺序保持
- 重复上述步骤直至序列处理完毕

5.6 Huffman树及其应用





5.6 Huffman树及其应用

频率越大其编码越短

- 各字符的二进制编码为：

d_0 ： 1011110　　d_1 ： 1011111

d_2 ： 101110　　d_3 ： 10110

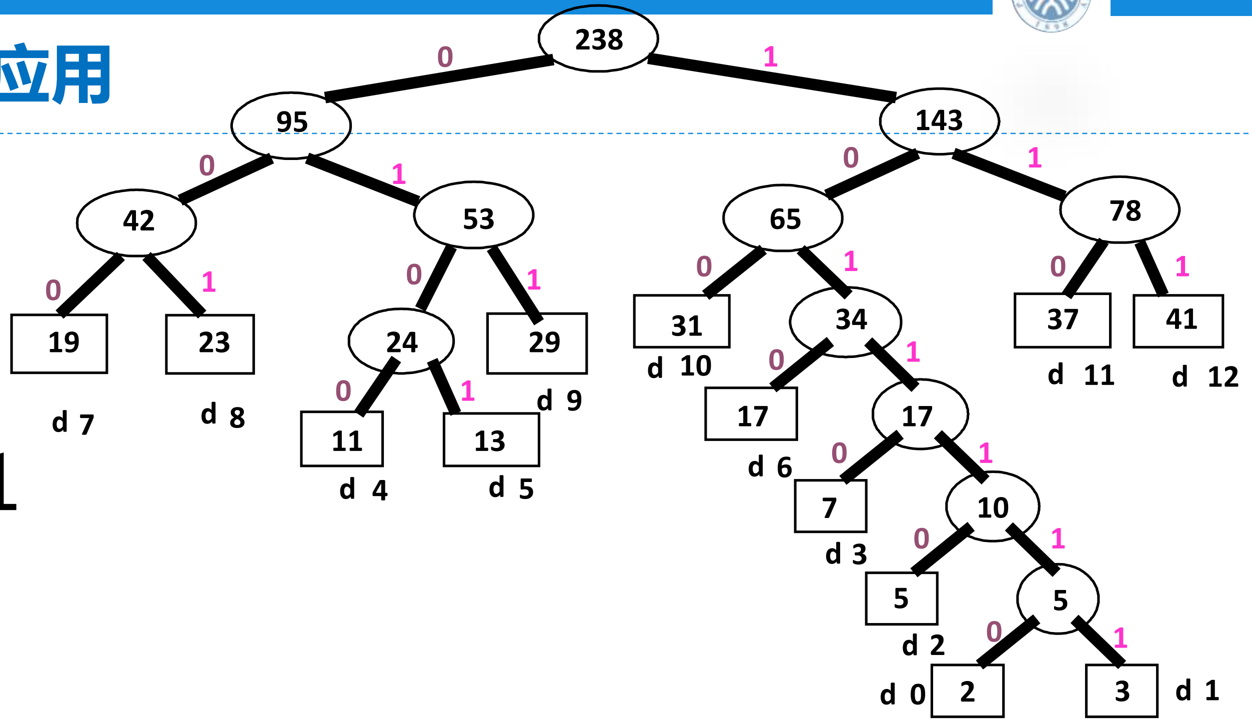
d_4 ： 0100　　d_5 ： 0101

d_6 ： 1010　　d_7 ： 000

d_8 ： 001　　d_9 ： 011

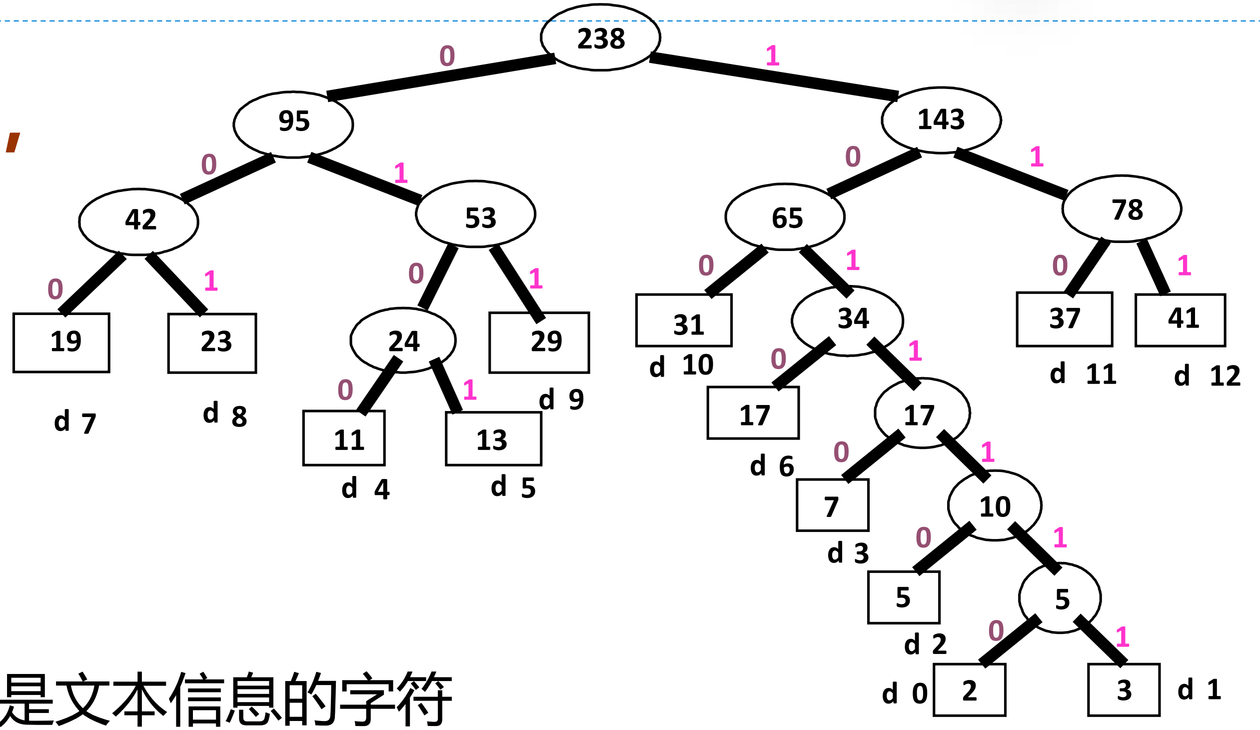
d_{10} ： 100　　d_{11} ： 110

d_{12} ： 111

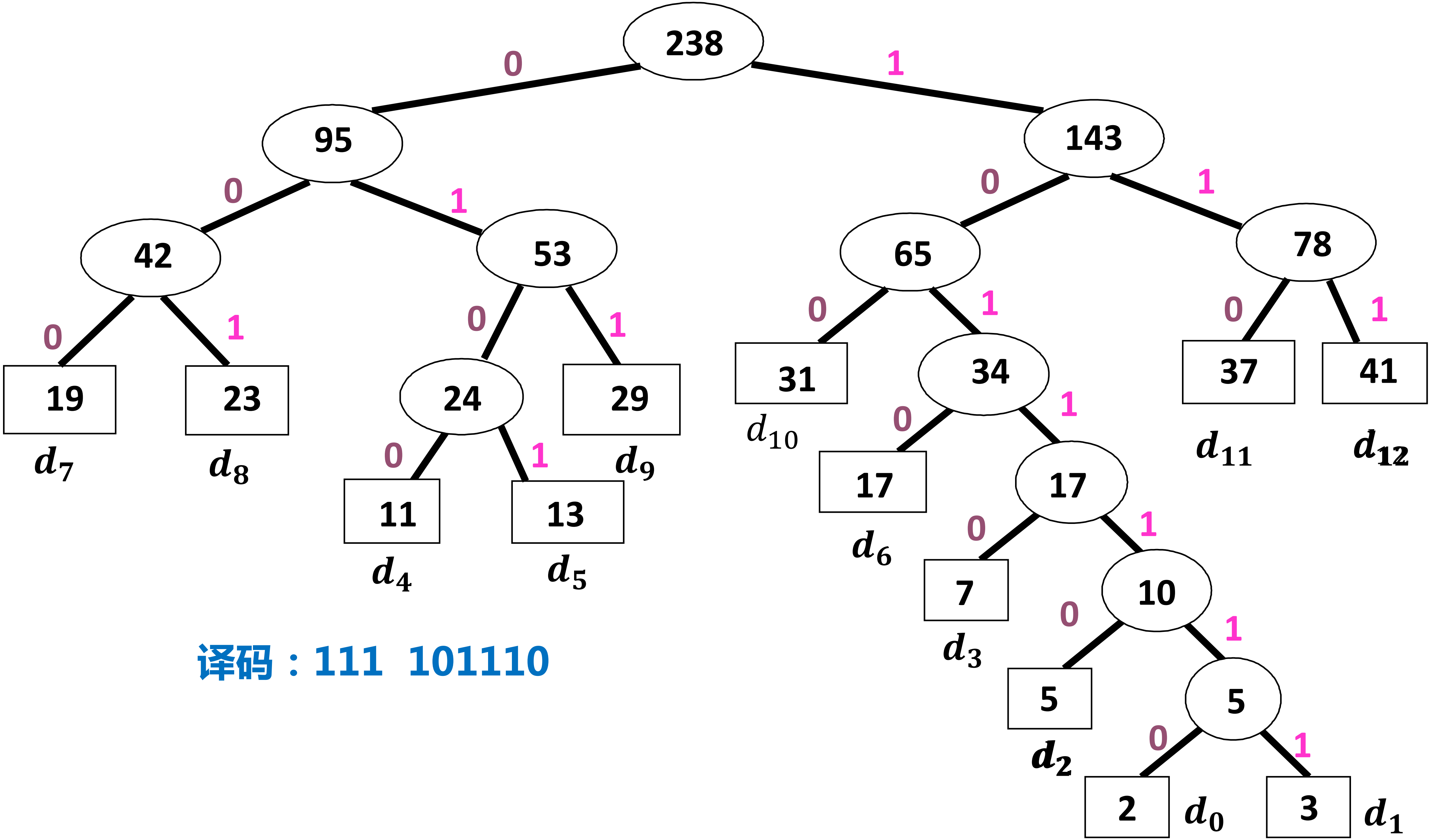


5.6 Huffman树及其应用

译码：从左至右逐位判别代码串，直至确定一个字符



- 与编码过程相逆
  - 从树的根结点开始
    - “0” 下降到左分支
    - “1” 下降到右分支
    - 到达一个树叶结点，对应的字符就是文本信息的字符
- 连续译码
  - 译出了一个字符，再回到树根，从二进制位串中的下一位开始继续译码



译码：111 101110

Huffman树类

```
template <class T> class HuffmanTree  {
private:
   HuffmanTreeNode<T>* root;//Huffman树的树根
   //把ht1和ht2为根的合并成一棵以parent为根的Huffman子树
   void MergeTree(HuffmanTreeNode<T> &ht1,
   HuffmanTreeNode<T> &ht2,  HuffmanTreeNode<T>* parent)；
public:
   //构造Huffman树，weight是存储权值的数组，n是数组长度
   HuffmanTree(T weight[],int n)；
   virtual ~HuffmanTree(){DeleteTree(root);}；  //析构函数
}
```

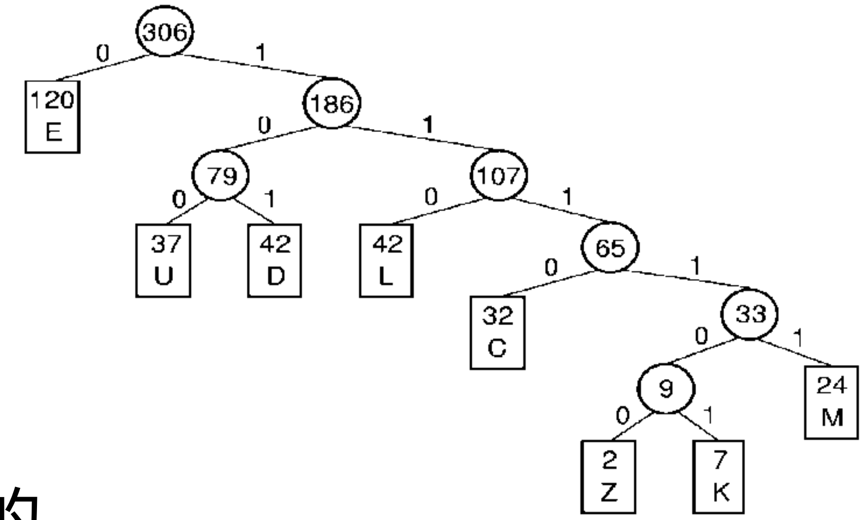
Huffman树的构造

```
template<class T>
HuffmanTree<T>::HuffmanTree(T weight[], int n) {
   MinHeap<HuffmanTreeNode<T>> heap;      //定义最小值堆
   HuffmanTreeNode<T> *parent,&leftchild,&rightchild;
   HuffmanTreeNode<T>* NodeList =
                              new HuffmanTreeNode<T>[n];
   for(int i=0; i<n; i++) {
      NodeList[i].element =weight[i];
      NodeList[i].parent = NodeList[i].left
                                    = NodeList[i].right = NULL;
      heap.Insert(NodeList[i]);                       //向堆中添加元
素
   } //end for
```

Huffman树的构造

```
for(i=0;i<n-1;i++) {                          //通过n-1次合并建立Huffman树
    parent=new HuffmanTreeNode<T>;
    firstchild=heap. RemoveMin();   //选值最小的结点
    secondchild=heap. RemoveMin();       //选值次小的结点
    MergeTree(firstchild,secondchild,parent); //合并权值最小的两棵树
    heap.Insert(*parent);                  //把parent插入到堆中去
    root=parent;                       //建立根结点
   }//end for
delete []NodeList;
}
```

Huffman方法的正确性证明



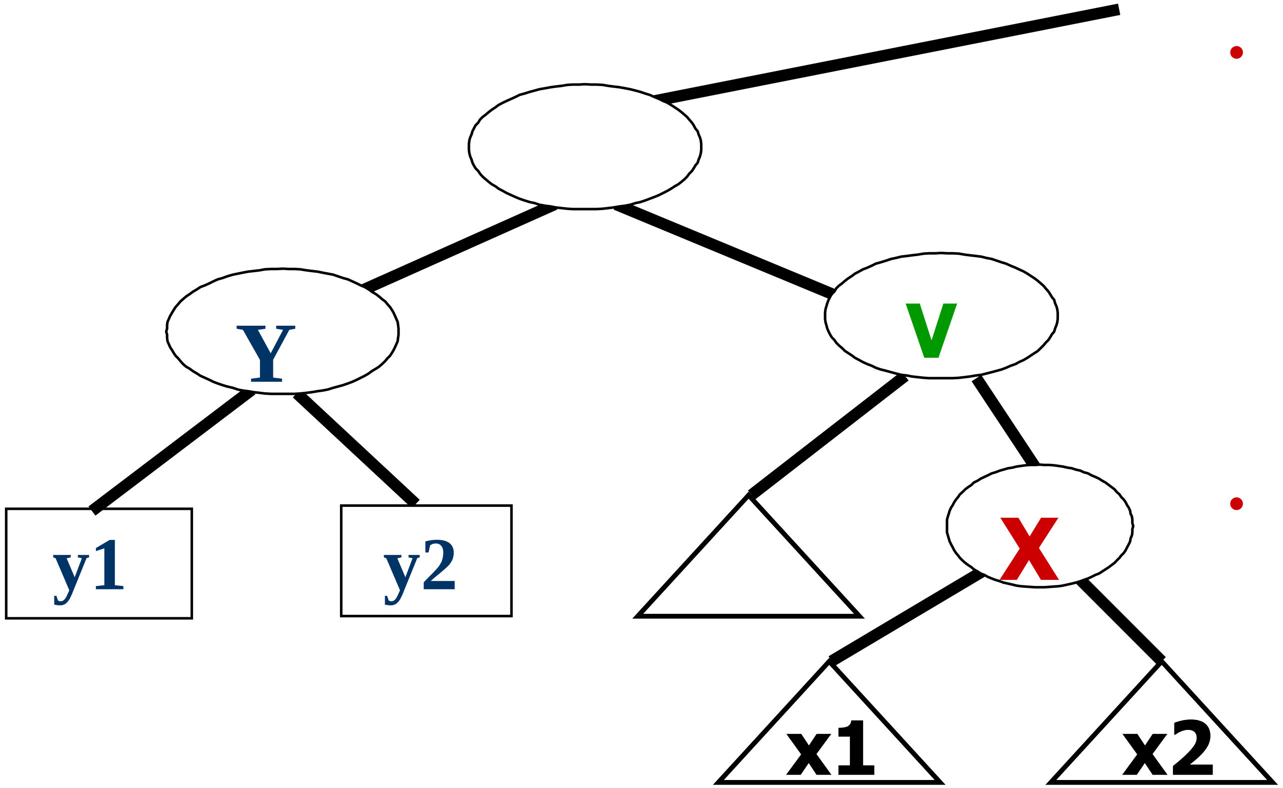
- 是否前缀编码？
- 贪心法的一个例子
  - Huffman树建立的每一步，“权”最小的两个子树被结合为一新子树
- 是否最优解？

Huffman性质

- 引理

含有两个以上结点的一棵 Huffman 树中，字符使用频率最小的两个字符是兄弟结点，而且其深度不比树中其他任何叶结点浅

证明



- 记使用频率最低的两个字符为 y1 和 y2
- 假设 x1, x2 是最深的结点
  - y1 和 y2 的父结点 Y 一定会有比 x 更大的“权”
  - 否则，会选择 Y 而不是 x 作为结点V的子结点
- 然而，由于 y1 和 y2 是频率最小的字符，这种情况不可能发生

5.6 Huffman树及其应用

- 定理：对于给定的一组字符，函数HuffmanTree实现了“最小外部路径权重”
- 证明：对字符个数n作归纳进行证明
- 初始情况：令n = 2, Huffman树一定有最小外部路径权重
  - 只可能有成镜面对称的两种树
  - 两种树的叶结点加权路径长度相等
- 归纳假设：

假设有n-1个叶结点的由函数HuffmanTree产生的

Huffman树有最小外部路径权重

5.6 Huffman树及其应用

归纳步骤：

- 设一棵由函数HuffmanTree产生的树 T 有 n 个叶结点，n>2，并假设字符的“权” $w_0 \le w_1 \le \ldots \le w_{n-1}$
  - 记 V 是频率为 w_0 和 w_1 的两个字符的父结点。根据引理，它们已经是树 T 中最深的结点
  - T 中结点 V 换为一个叶结点 V′（权等于$w_0 + w_1$），得到另一棵树 T′
- 根据归纳假设，T′ 具有最小的外部路径长度
- 把 V′ 展开为V（$w_0 + w_1$），T′ 还原为 T，
  则 T 也应该有最小的外部路径长度
- 因此，根据归纳原理，定理成立

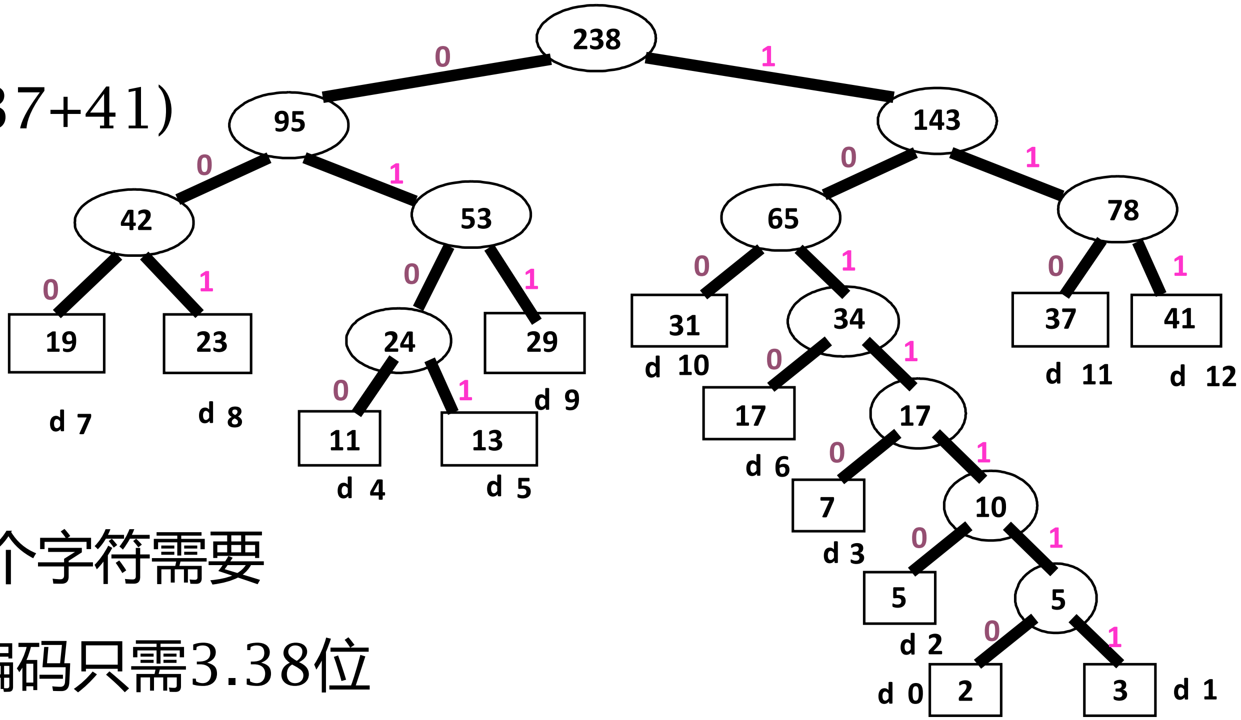
Huffman树编码效率

- 估计Huffman编码所节省的空间
  - 平均每个字符的代码长度等于每个代码的长度 c_i 乘以其出现的概率 p_i ，即:
  - $c_0p_0 + c_1p_1 + \ldots + c_{n-1}p_{n-1}$
    或 $(c_0f_0 + c_1f_1 + \ldots + c_{n-1}f_{n-1}) / f_T$

这里f_i为第i个字符的出现频率，而f_T为所有字符出现的总次数

Huffman树编码效率（续）

- 图中，平均代码长度为：
  (3*(19+23+24+29+31+34+37+41)
  + 4*(11+13+17)
  + 5*7
  + 6*5
  + 7*(2+3)) / 238
  = 804/238 ≈ 3.38



- 对于这13个字符，等长编码每个字符需要

  $\lceil \log 13 \rceil = 4$位，而Huffman编码只需3.38位

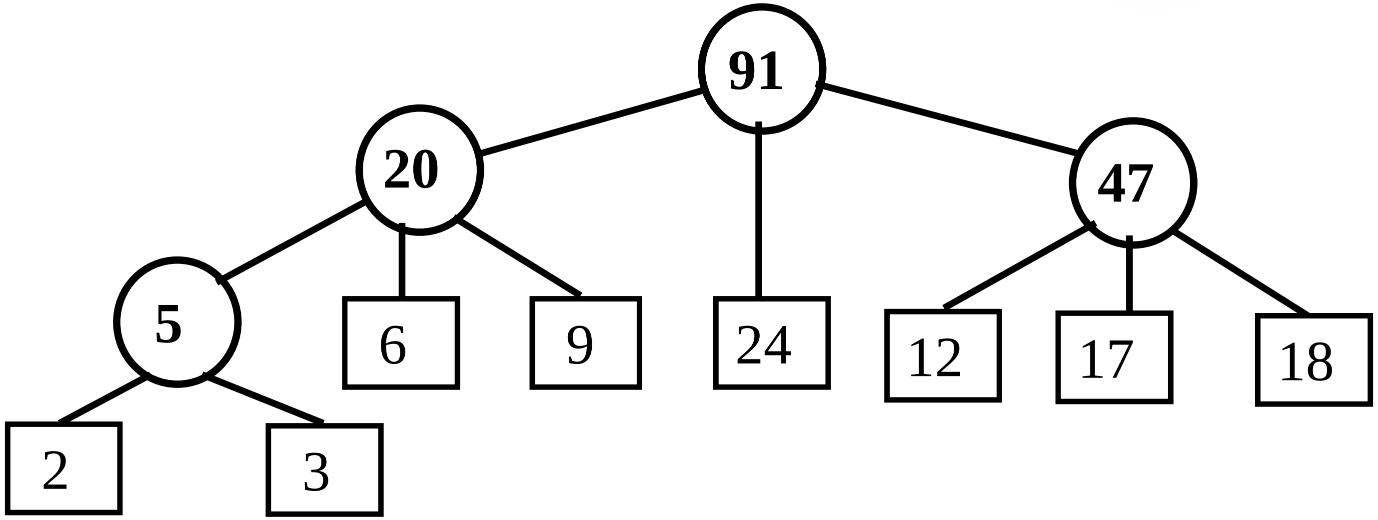
- Huffman编码预计只需要等长编码3.38/4≈ 84% 的空间

Huffman树的应用

- Huffman编码适合于
  字符 **频率不等，差别较大的情况**
- 数据通信的二进制编码
  - 不同的频率分布，会有不同的压缩比率
  - 大多数的商业压缩程序都是采用几种编码方式以应付各种类型的文件
    - Zip 压缩就是 LZ77 与 Huffman 结合
- 归并法外排序，合并顺串

5.6 Huffman树及其应用

思考



- 当外部的数目不能构成满 b叉 Huffman 树时，需附加多少个权为 0的“虚”结点？请推导
- R 个外部结点，b叉树
  - 若（r-1）% (b-1)==0，则不需要加“虚”结点
  - 否则需要附加 b -（r-1）% (b-1) - 1个“虚” 结点
    - 即第一次选取（r-1）% (b-1) + 1个非0权值
- 试调研常见压缩软件所使用的编码方式

数据结构与算法

谢谢聆听

国家精品课“数据结构与算法”
http://www.jpk.pku.edu.cn/pkujpk/course/sjjg/

张铭，王腾蛟，赵海燕
高等教育出版社，2008. 6。 “十一五”国家级规划教材