数据结构与算法

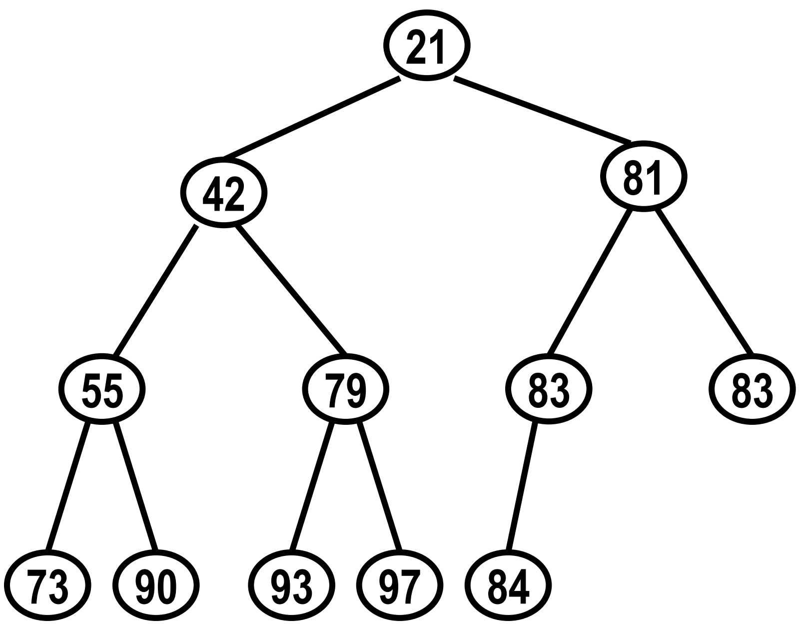
第 5 章 二叉树

主讲：赵海燕
北京大学信息科学技术学院
“数据结构与算法”教学组

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

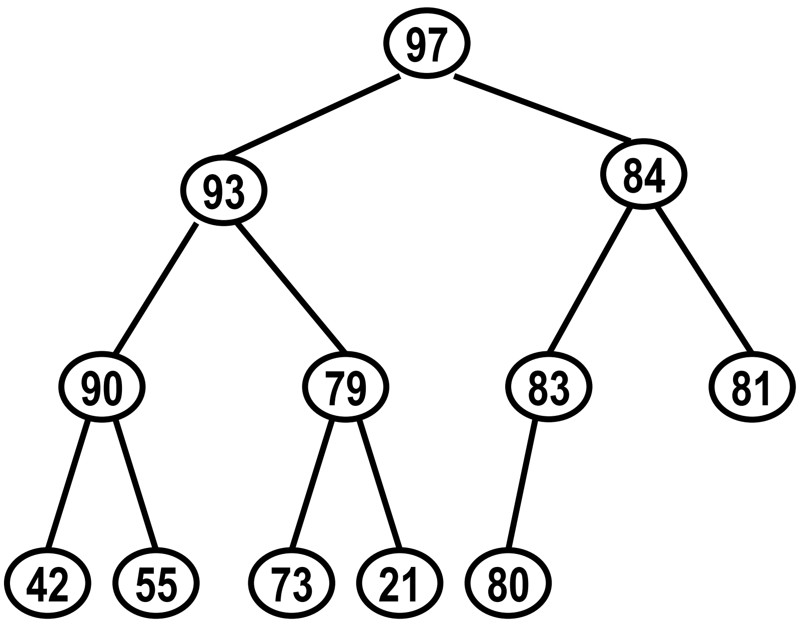
张铭，王腾蛟，赵海燕
高等教育出版社，2008. 6, “十一五”国家级规划教材

满足某种特性的二叉树



- **最小值堆（min-heap)**
  - 每个结点的（关键码）值都小于或等于其子结点的值
  - 根结点存储了树的所有结点的**最小值**
    - 根结点含有小于或等于其子结点的值，而其子结点又依次小于或等于各自子结点的值

满足某种特性的二叉树



- 最大值堆（max-heap)
  - 任一个结点的值都**大于或者等于**其任一子结点的值
  - 根结点存储着树中所有结点的**最大值**
    - 根结点含有**大于或等于**其子结点的值，而其子结点又依次**大于或等于**各自子结点的值

堆的定义（heap）

- 一个关键码序列{ K_0, K_1, ..., K_{n-1}}，具有如下**特性**：

  $K_i \geqslant K_{2i+1}$ / $K_i \leqslant K_{2i+1}$,

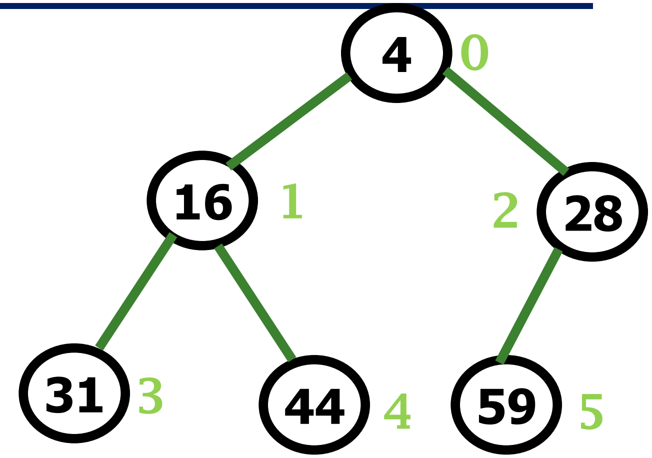
  $K_i \geqslant K_{2i+2}$ / $K_i \leqslant K_{2i+2}$,

  $(i = 0, 1, ..., \lfloor n/2 \rfloor)$

  则称其为堆。 即，

  - 最大值堆 /最小值堆（根据数据间的大小关系）

堆的性质

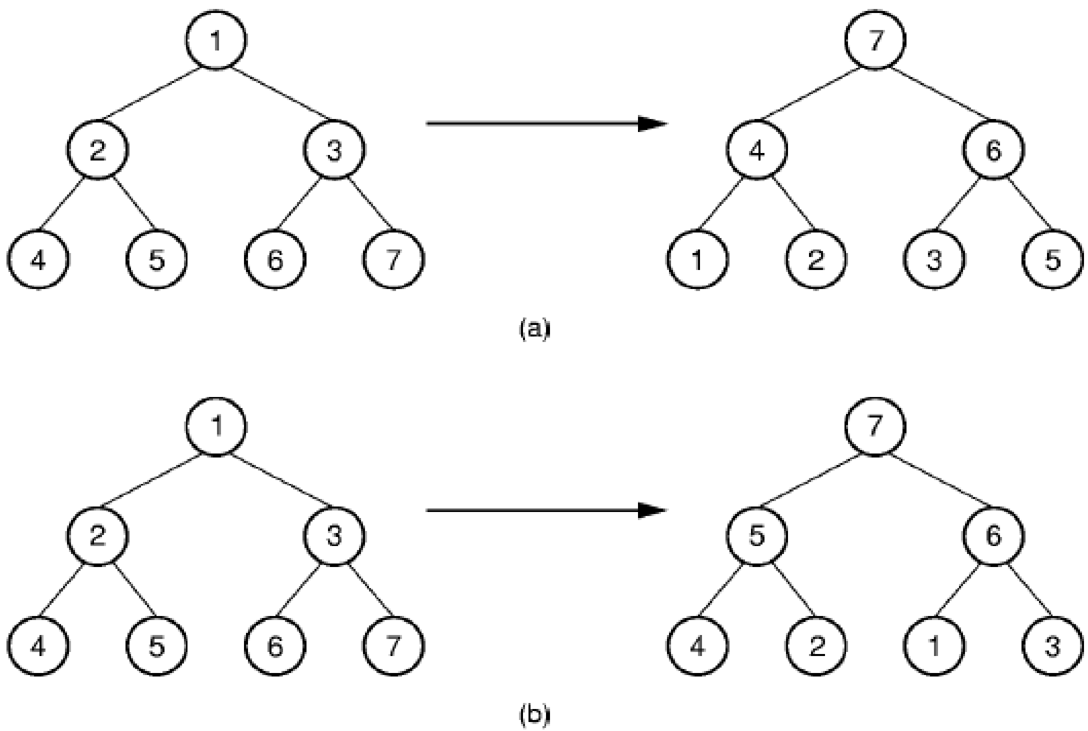


- **从逻辑角度看，堆实际上是一种树型结构**
  - 堆实际上是一个完全二叉树的层次序列，可以用数组表示
- **堆是局部有序的，堆不唯一**
  - 结点存储的值与其子结点存储的值之间存在某种联系
  - 堆中任何一个结点与其兄弟之间都没有必然的联系
  - 最小堆并非BST那样实现关键码的完全排序，而是局部有序，只有父子结点的大小关系可以确定

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
}
```

如何建堆?



（a）和（b）均建成一个堆，但经过的步骤大不相同，效率也就不同

(a) (4-2) (4-1) (2-1) (5-2) (5-4) (6-3) (6-5) (7-5) (7-6)

(b) (7-3) (5-2) (7-1) (6-1)

如何建堆?

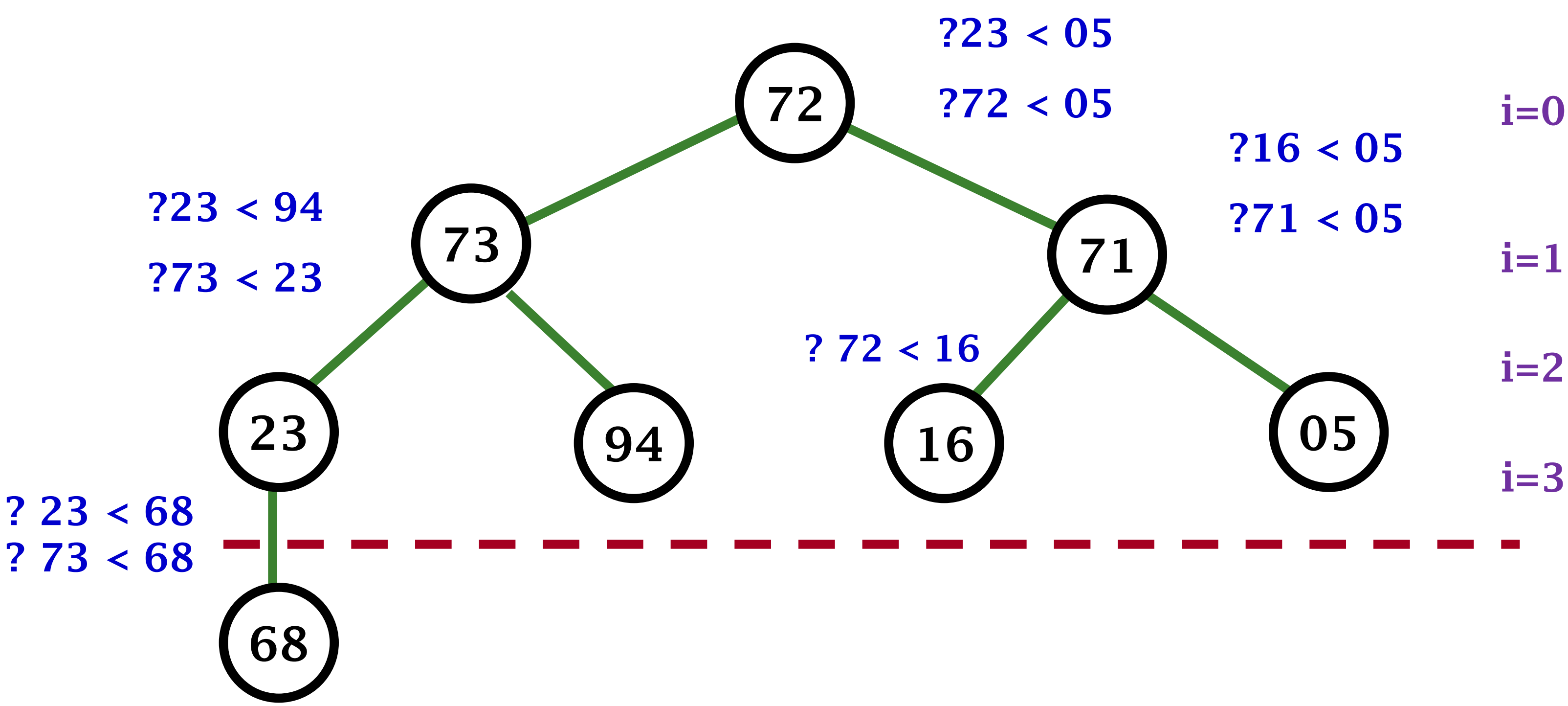
- 通过交换形成堆，而不必将值逐个插入堆中
  - 假设根的左、右子树都已是堆，且根为*R*，有两种可能（以最小值堆为例):
    1. *R* 的值小于或等于其两个子结点的值，此时为堆；
    2. *R* 的值大于其某一个或两个子结点的值，此时*R* 应与两个子结点中较小者交换，得到一个堆，除非*R* 仍然大于其新的子结点的值。这种情况下，需继续这种将*R*“拉下来”（*siftdown*）的过程，直至到达某一个层使它不大于其子结点的值，或者其已为叶结点

筛选法（**1964年Floyd提出**）

筛选法（siftdown）

- 首先，将 n 个关键码放到一维数组中
  1. 整体并非最小堆
  2. 所有以叶结点为根的子树为堆，即，当 $i \geq \lfloor n/2 \rfloor$ 时，以关键码 K_i 为根的子树已经是堆
- 从**最后一个分支结点**，$i = \lfloor n/2 \rfloor - 1$ 开始，**从右向左**、**自底向上**逐步将以各分支结点为根的子树调整成堆，直到树根为止

最小堆构建示意


?23 < 05
?72 < 05
72
i=0
?16 < 05
?71 < 05
?23 < 94
?73 < 23
73
71
i=1
? 72 < 16
i=2
23
94
16
05
i=3
? 23 < 68
? 73 < 68
68

建堆的SiftDown操作

```
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
   T  temp = heapArray[i];                   // 保存父结点
   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;
}
```

SiftDown的时间代价

- 最差情况
  - 2 次比较（判断子结点的大小，及结点是否需要筛选）
  - 1 次交换

一个具有 N 个结点的完全二叉树，最多具有 $\lceil \log(N+1) \rceil$层

每循环一次把目标结点下移一层，故循环最多为 log N 次

∴ 最差情况下 SiftDown 的时间代价为O(log N)

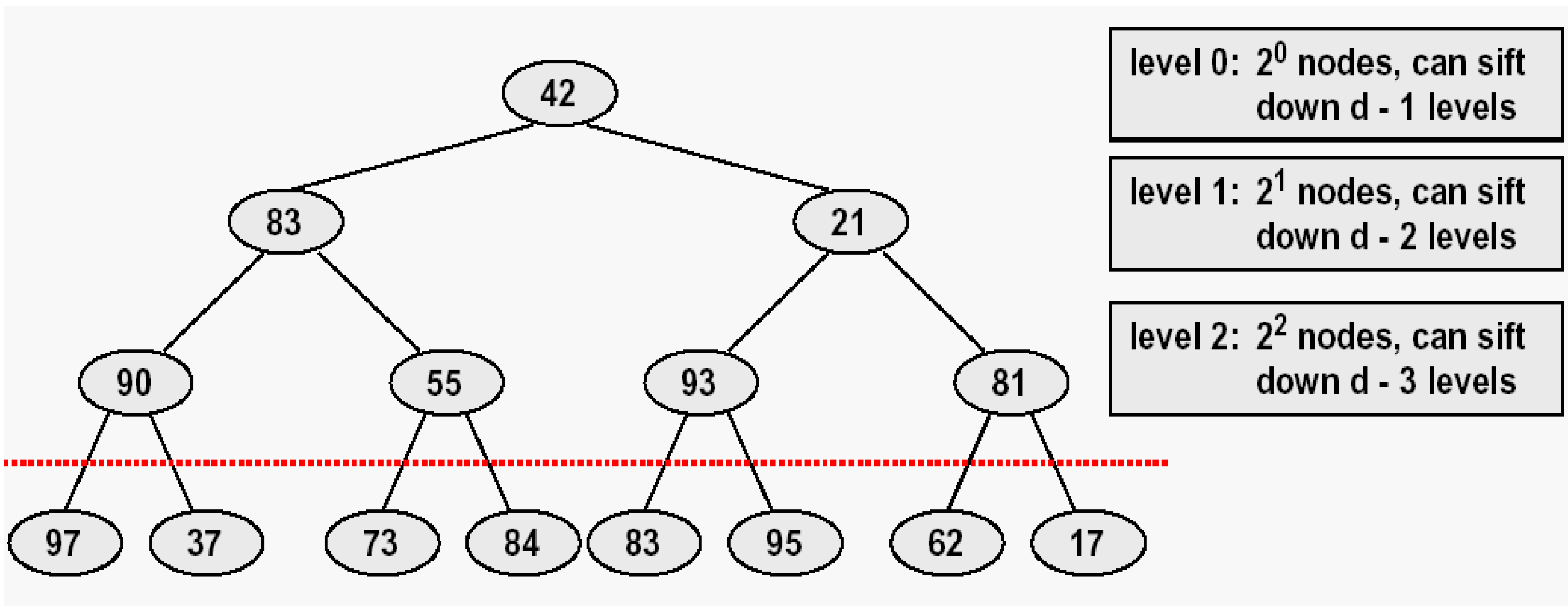
建堆过程小结

1. 先将所有元素组织成一维数组，此时所形成的完全二叉树尚不具备最小堆的特性，只有那些叶结点所构成的子树满足堆的性质
2. 从最后一个分支结点（完全二叉树的倒数第二层，此时i = >[(n-1)/2] ）开始，从右至左依次通过筛选法调整
3. 对一层调整完后，继续对上一层进行同样的调整工作，直到整个过程到达树根时，整棵完全二叉树就成为一个堆

建堆的时间代价

- 对于一个N个结点的完全二叉树，若同时为满二叉树，则筛选的层数可能最大，此时：

  $n = 2^d - 1, \quad d = \lceil \log n \rceil$



- 由性质 4知，第 k 层最多有2^k个结点，且离叶结点的距离为 d-k-1 层

建堆的时间代价

最差情况下，构建具有 n个结点的堆需要的比较次数为：

$$2\sum_{k=0}^{d-1} 2^k (d-k-1) = 2\left[(d-1)\sum_{k=0}^{d-1} 2^k - 2\sum_{k=0}^{d-1} k2^{k-1}\right]$$

$$= 2\left[2^d - d - 1\right] = 2\left[n - \lceil \log n \rceil\right]$$

最差情况下，**每两个比较**需要**一次交换操作**，故最大的交换次数为 n -⌈log n⌉

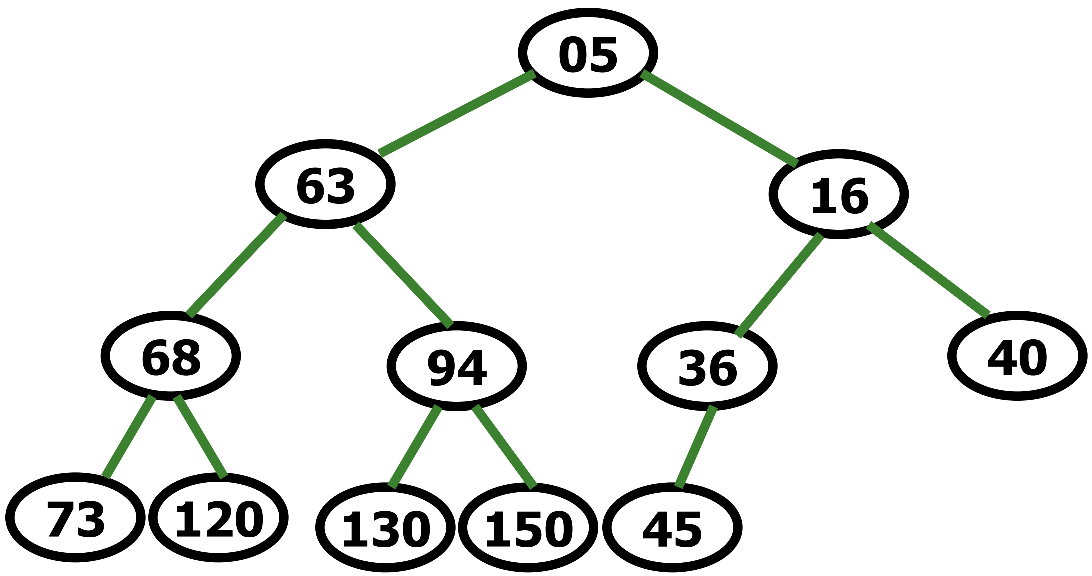
∴ 构建具有 n 个结点的堆，其比较和交换次数均为 O(n)

删除根结点

- 堆上最常用的操作是删除根结点（须在删除后保持堆的特性）

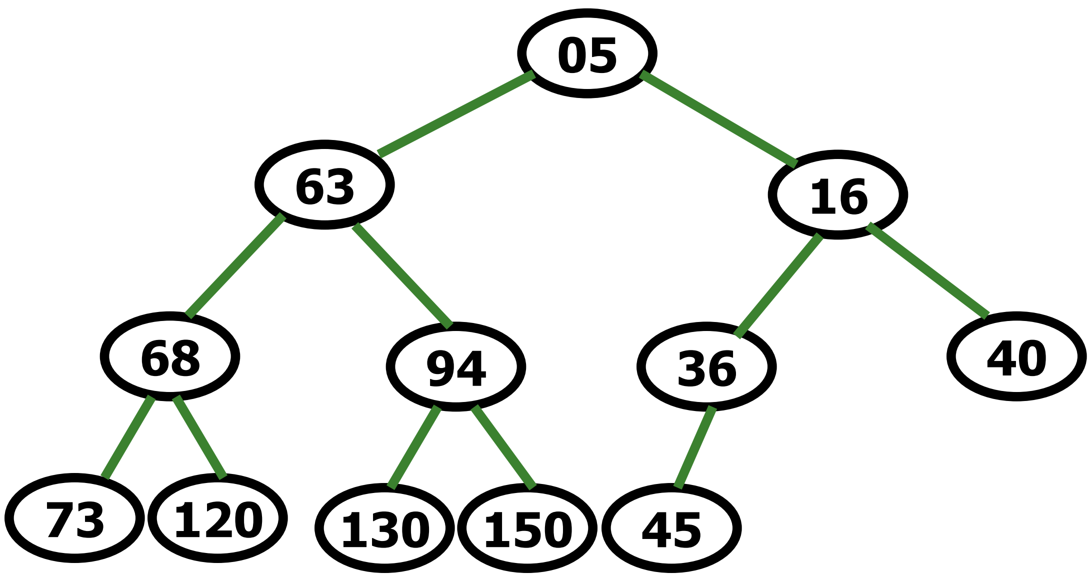
```
template <class T> T HeapT<T>:: RemoveRoot() {

    if (CurrentSize == 0) exit(1);

    Item tmpItem = heapArray[0];
    heapArray[0] = heapArray[CurrentSize - 1];
    heapArray[CurrentSize - 1] = tmpItem;

    CurrentSize --;

    SiftDown(0);

    return tmpItem;
}
```

根结点删除示意



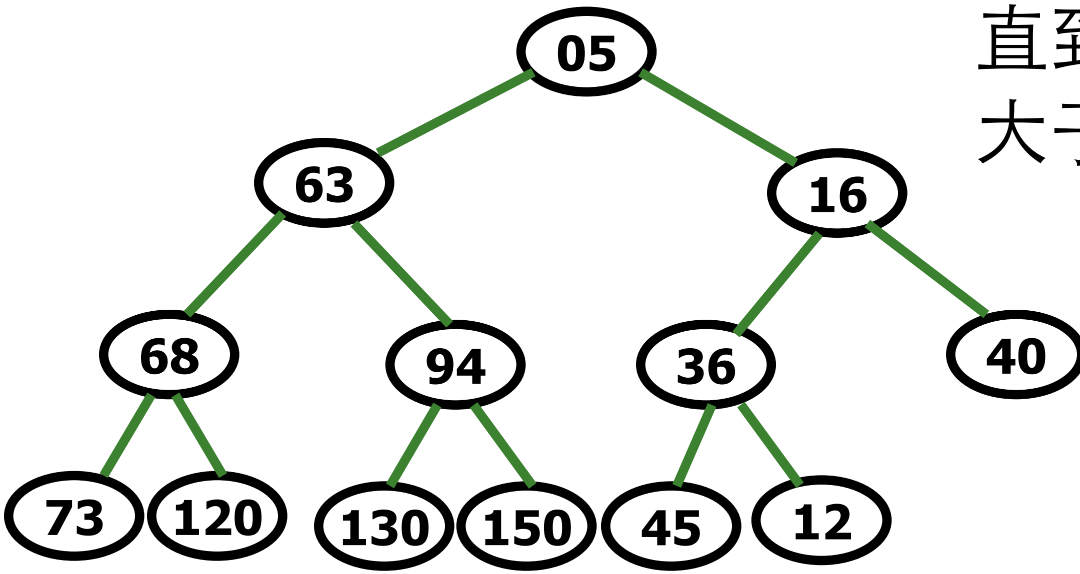
删除堆中任一元素

- 如何删除根以外的元素？是否可以沿用删除根结点的方法?
- E.g., 删除68



堆中插入新元素

- 插入元素12



- 插在堆的最后，然后逐步向上与其父结点进行比较，若不满足堆的性质则**往上拉**（*SiftUp*），直到其父结点的值不再大于它

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 =h eapArray[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
}
```

堆运算分析

- 建堆算法的时间复杂度是 O(n)
  - 线性时间内把一个无序的序列转化成堆序
- 堆的深度为log n
  - 插入结点、删除元素 的平均时间代价和最差时间代价都是O(log n)
- 最小堆只适合于查找最小值，查找任意值的效率不高

堆的应用

- 堆排序
- 优先队列(Priority Queue)
  - 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

- SiftDown操作时，一旦发现逆序对就交换会怎么样?
- 如何删除堆中的任一元素?
- 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）

编码

- 程序设计、数据通信等领域常需为某些字符集（或一般意义上的集合）进行编码：即，按某种规则用一个单独的代码来标识字符集（集合）中的每一个字符（元素）
  - 定长编码 （fixed-length coding scheme）
  - 变长编码（variable-length coding scheme）

固定长度编码

- 若所有字符对应的代码都等长，则表示 n 个不同代码需要$\log_2 n$位，称为固定长度编码
  - ASCII码就是一种固定长度编码（7位）
- 在每个字符的使用频率相同情况下，固定长度编码是空间效率最高的方法
- 具有**简单、解码容易**的优点

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

  可利用字母的出现频率来编码，使得经常出现的字母的编码较短，不常出现的字母编码较长
- 不等长编码是文件压缩技术的核心
  - 数据压缩既能节省磁盘空间，又能提高传输速度（**外存时空权衡的规则**）
  - Huffman编码是最简单的文件压缩技术，展示了不等长编码方法的基本思想

前缀编码

- 不等长编码要注意的问题：任何一个字符的编码都不能是另外一个字符编码的前缀；否则解码不唯一
  - 例如，对于字符集{Z, K, F, C, U, D, L, E}，若编码为 Z(0), K(1), F(00), C(01), U(10), D(11), L(000), E(001)，

    则不可行，因为在这种编码中，

    代码 “000110” 可以翻译为

    “ZZZDZ”， 或 “LDZ”，或 “FCU”

前缀编码

- 一个编码集合中，任何一个字符的编码都不是另一个字符编码的前缀，这种编码叫作前缀编码
- **前缀特性**保证了代码串被解码时，不会出现歧义
  - 例如，对于上面 8 个字符， 分别编码为

    Z(111100), K(111101), F(11111), C(1110),

    U(100), D(101), L(110), E(0)

    即为一种前缀编码

    代码串“000110” 可以解码为唯一的字符串“ EEEL”

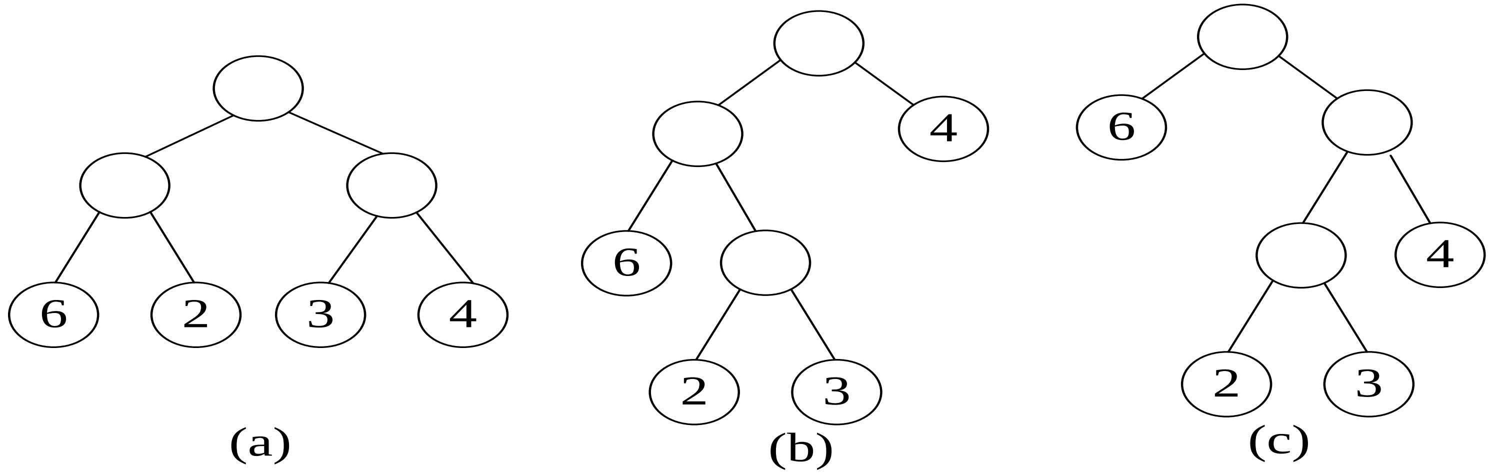
二叉树 vs 前缀编码

- 可用二叉树来设计和表示前缀编码：
  - 约定叶结点表示字符；
  - 从根结点到叶的路径中，左分支表示‘0’，右分支表示‘1’，从**根结点到叶结点上的路径分支**所组成的字符串作为该叶结点字符的**编码**

  这样的编码一定是**前缀编码** （why?)

- 如何保证这样的编码树所得到的编码总长度最小?
  - Huffman算法解决了这个问题

编码总长示例



三棵具有4个外部结点的二叉树，各外部结点的权值分别为6，2，3，4，(a)、（b）、（c)三中形态的带权外部路径长度分别为：

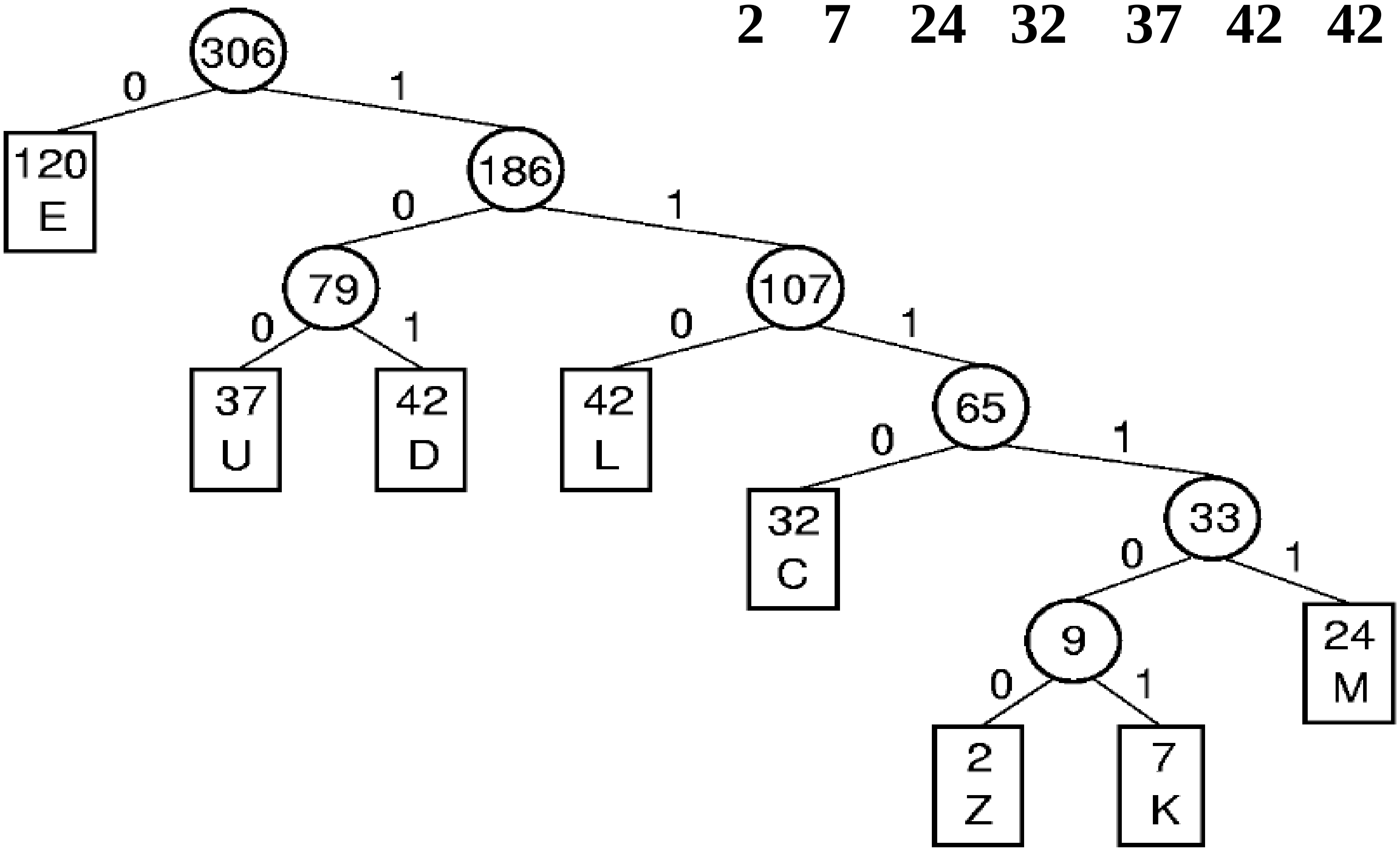
(a) $6\times2+2\times2+3\times2+4\times2=30$

(b) $6\times2+2\times3+3\times3+4\times1=31$

(c) $6\times1+2\times3+3\times3+4\times2=29$

编码总长示例

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



Huffman树

- 设 $D = \{d_0, \ldots, d_{n-1}\}$，
  $W = \{w_0, \ldots, w_{n-1}\}$
  D为待编码的字符集，W为D中各个字符出现的频率，要对D中字符进行二进制编码，使得：
  - 通信编码总长最短
  - $\forall$ i, $\forall$ j, 若$d_i \neq d_j$，则d_i的编码不可能是d_j的编码的前缀
- 利用Huffman算法编码：

  将d_0，d_1，…，d_{n-1}作 **外部结点**，w_0，w_1，…，w_{n-1}看作外部结点的权，构造具有最小带权外部路径长度的扩充二叉树

Huffman树

- 即，给出一个具有n个外部结点的扩充二叉树
  - 每个外部结点d_i 有一个w_i与之对应，作为该外部结点的权
  - 这个扩充二叉树的外部结点带权外部路径长度总和

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

最小

（注意不管内部结点，也不必有序）

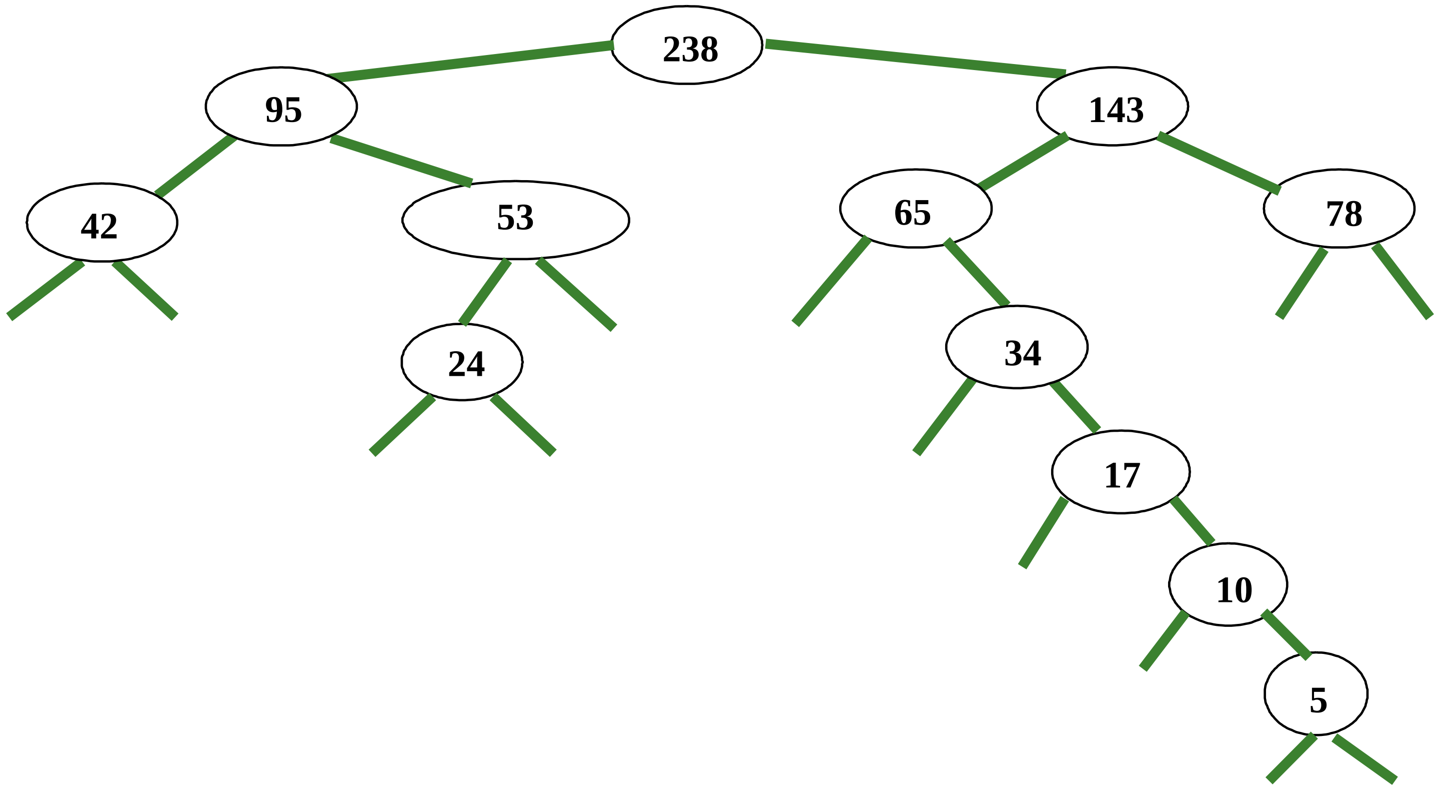
  - 权越大的叶结点离根越近；权较小的叶结点，离根较远

建立Huffman树

步骤：

1. 按照“权”(诸如，频率)将字符组成有序序列；
2. 取走前两个字符（“权”最小的两个字符），将其标记为Huffman树的叶结点，并将这两个叶结点标为一个（新）**分支结点**的两个子结点，该分支结点的权为两叶结点的**权之和**。将所得子树的“权”放回序列中适当位置，保持“权”的顺序；
3. 重复上述步骤直至序列中只剩一个元素，则Huffman树建立完毕

Huffman树建立示意

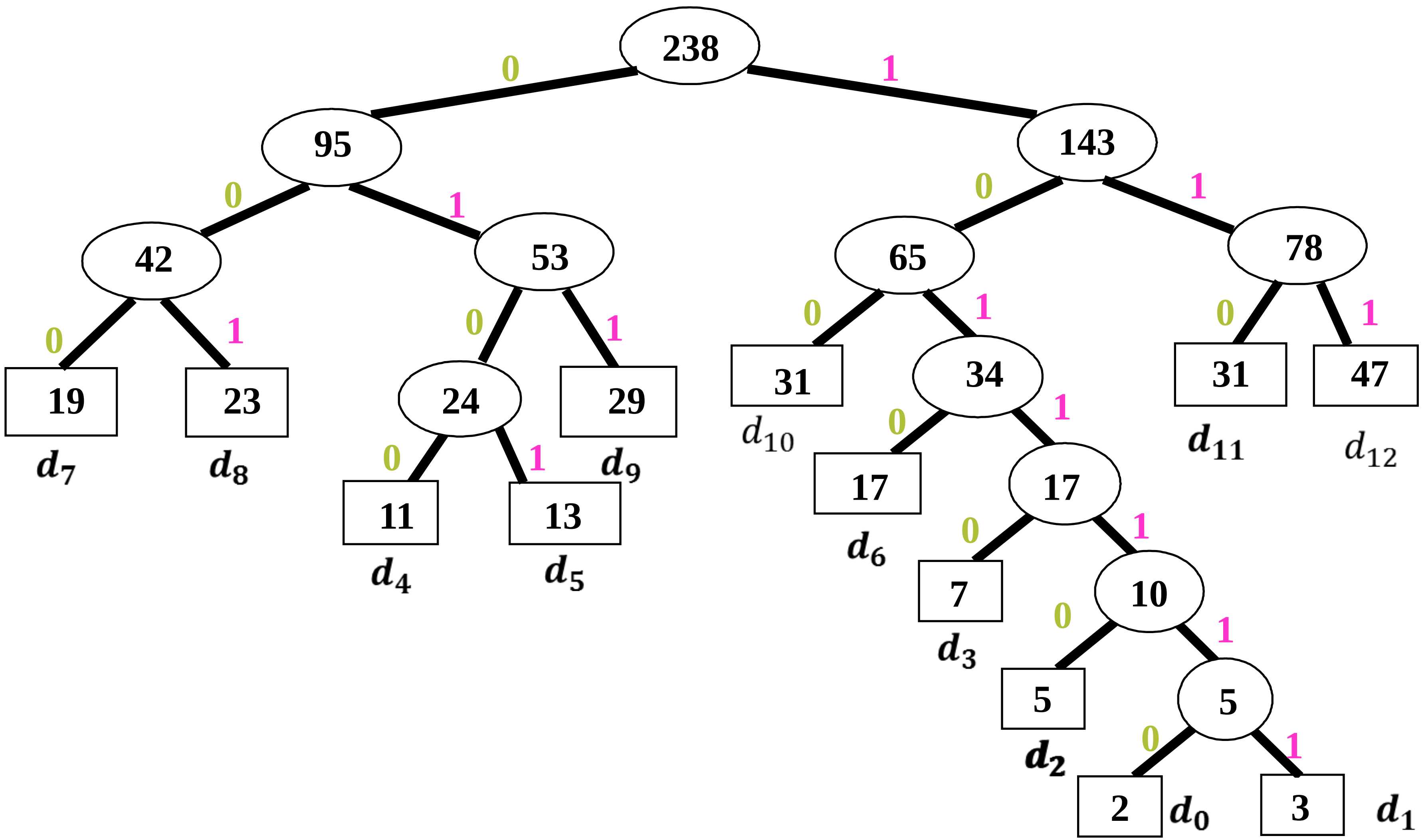


2	3	5	7	11	13	17	19	23	29	31	37	41

Huffman树

- 从根到每个叶结点的路径上编号连接起来即该叶结点代表的字符的编码
  - 从每个结点到其左子结点的边编号为0
  - 从每个结点到 其右子结点的边编号为1

Huffman编码示例



Huffman编码示例

- 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

- 出现频率越大的字符其编码越短，以提高检索速度

Huffman树的类定义

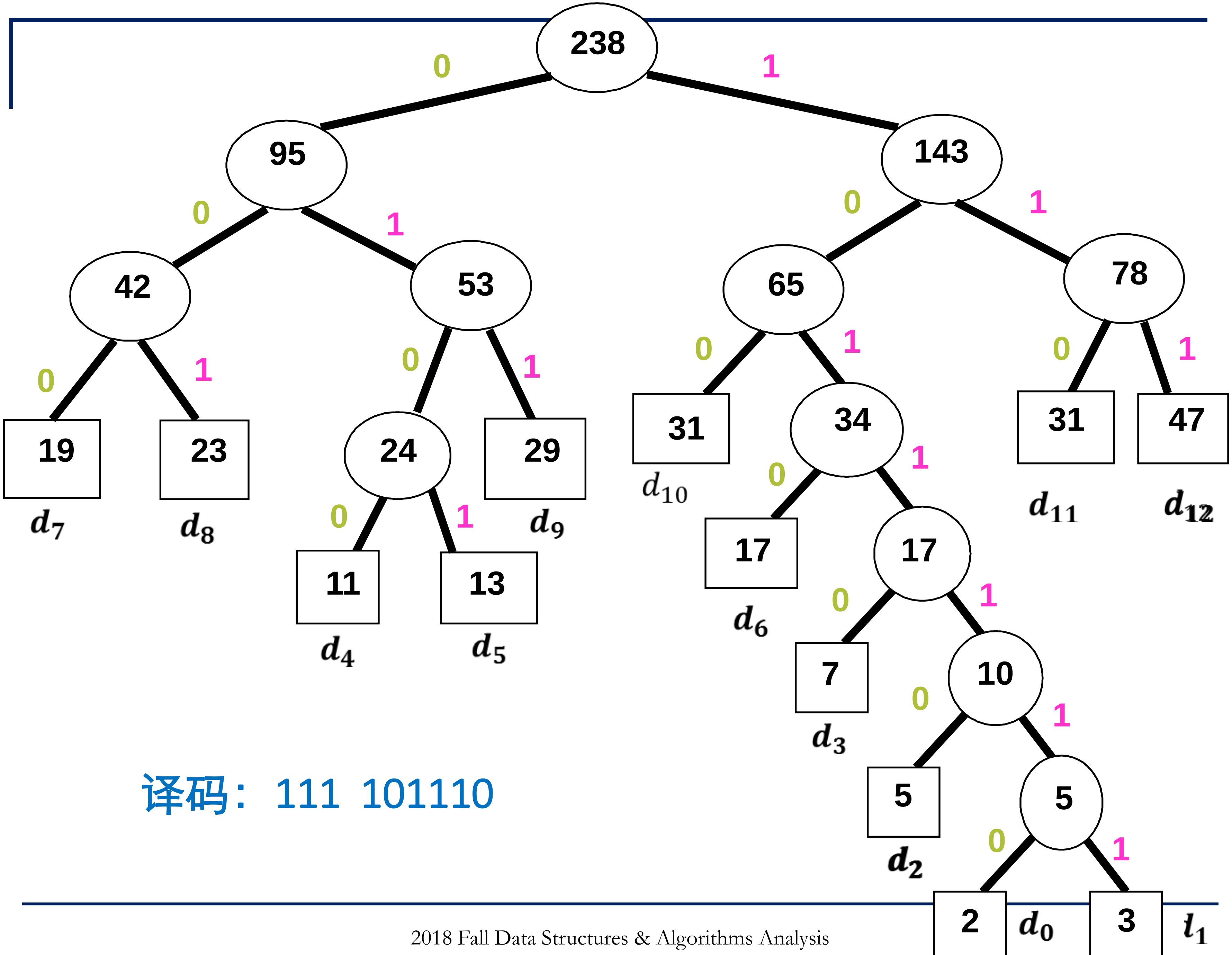
```
template<class T>
class HuffmanTree  {
private:
    HuffmanTreeNode <T> *root;                        // Huffman树的根结点
    // 把以ht1和ht2为根的两棵Huffman树合并成一棵以parent为根的二叉树
    void MergeTree(HuffmanTreeNode<T> &ht1, HuffmanTreeNode<T>
      &ht2,  HuffmanTreeNode<T> *parent);
     // 删除Huffman树或其子树
    void DeleteTree(HuffmanTreeNode<T> *root);

public:
    // 构造Huffman树，参数weight为权值数组，n为数组长度
    HuffmanTree(T weight[],int n);
    virtual ~HuffmanTree(){DeleteTree(root);};          // 析构函数
}
```

Huffman树建立算法

```
template<class T>
HuffmanTree<T>::HuffmanTree(T weight[], int n)  {
      MinHeap< HuffmanTreeNode<T> > heap(n);     // 最小值堆
      HuffmanTreeNode<T> *parent, firstchild, secondchild;
      HuffmanTreeNode<T> *NodeList = new HuffmanTreeNode<T>[n];
      for (int i = 0;i < n;i++) {                // 初始化
        NodeList[i].info = weight[i];
        NodeList[i].parent = NodeList[i].left = NodeList[i].right = NULL;
             heap.Insert(NodeList[i]);           // 向堆中添加元素
      }
      for (i = 0;i < n-1;i++) {                  // 通过n-1次合并建立Huffman树
        parent = new HuffmanTreeNode<T>;         // 申请一个分支结点
        firstchild = heap.RemoveMin();           // 选择权值最小的结点
        secondchild = heap.RemoveMin();          // 选择权值次小的结点
        // 将权值最小的两棵树合并到parent树
        MergeTree(firstchild,secondchild,parent);
        heap.Insert(*parent);                    // 把parent插入到堆中去
        root = parent;                           // Huffman树的根结点赋为parent
      }
      delete [] NodeList;
}
```

Huffman编码及其解码

- 用Huffman算法构造出的扩充二叉树给出了各字符的编码，同时也用来解码/译码
- 译码与编码过程相逆
  - 从树的根结点开始
    - 沿 0 下降到左分支
    - 沿 1 下降到右分支
    - 直到一个**叶结点**，其所对应字符即为文本信息的字符
  - 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

思考

- Huffman方法的正确性证明
  - 是否前缀编码?
  - 是否最优解?
    - 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

Huffman树编码效率

- 估计Huffman编码所节省的空间
  - 字符的平均编码长度等于每个字符的编码长度c_i乘以其出现的概率 p_i ，即:

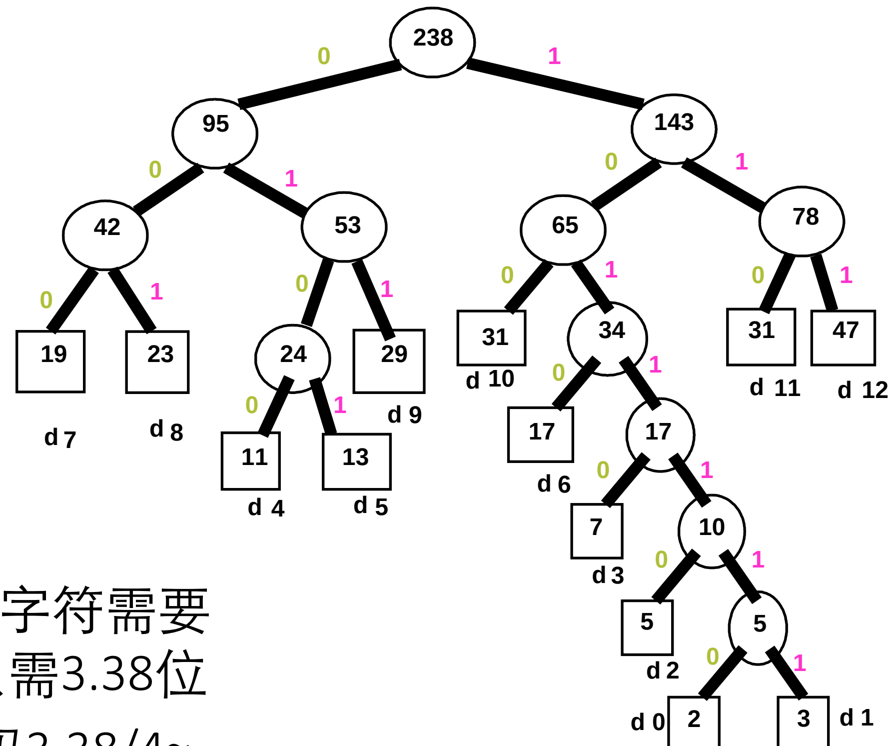
$$c_0p_0 + c_1p_1 + \ldots + c_{n-1}p_{n-1}, \quad \text{or}$$

$$(c_0f_0 + c_1f_1 + \ldots + c_{n-1}f_{n-1}) / f_T$$

其中，f_i为第i个字符的出现频率，f_T为所有字符出现的总次数

Huffman树编码效率

- 图中，平均代码长度为：
  (3*(19+23+29+31+31+47)
  + 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树编码效率

100,000 个字符组成的文件，只有 *6* 种字符出现

	a	b	c	d	e	f
出现频率%	45	13	12	16	9	5
定长码	000	001	010	011	100	101
变长码	0	101	100	111	1101	1100

100,000*3 = 300,000

(45*1+13*3+12*3+16*3+9*4+5*4)*1000 = 224,000

节省约25%的空间

本章总结

- 二叉树的主要概念与相关性质
- 二叉树的抽象数据类型、存储表示与实现效率
  - 穿线树
- 二叉树的遍历策略
- 二叉搜索树及其应用
- 堆的概念、性质与构造
- Huffman树的主要思想与具体应用

课堂练习☺

1. 假设现有元素：7，16，49，82，5，31，6，2，44。画出将每一个元素插入堆中以后的最大值堆。
2. 序列23,17,14,6,13,10,1,5,7,12是否为一个最大值堆？一个从小到大排序的数组是否最小值堆?
3. 已知某电文中共出现了10种不同的字母，每个字母出现的频率分别为A：8，B：5，C：3，D：2，E：7，F：23，G：9，H：11，I：2，J：35，现在对这段电文用三进制进行编码（即码字由0，1，2组成），问电文编码总长度至少有多少位？请画出相应的图。
4. 已知某字符串S总共有8种字符组组成，各种字符分别出现2次、1次、4次、5次、7次、3次、4次和9次，对该字符串用{0,1}进行前缀编码，问该字符串的编码至少有多少位?

K-Huffman

- 根据给出的 N 个字符的权值，对其进行 k 进制的huffman编码，如下步骤可否?
  - 出现权值最小的 k 个字符合并，权值小的放在前面，权值相同，则按先后排序，将这个组合看作一个新的组合，其权值相当于组合中所有字符权值的总和，用这个新字符代替那些组成它的字符
  - 然后再选择 k 个权值最小的字符 (包括新组合出来的字符)，再次按照上面说的次序合并
  - 以此类推，直到只剩下一个字符为止

K-Huffman练习题

- Variable Radix Huffman Encoding：
  http://acm.pku.edu.cn/JudgeOnline/problem?id=1880
  - 主要任务：根据前N个大写字母的出现次数，对其进行k进制的huffman编码