数据结构

第七讲 二叉树的应用

孙猛

http://www.math.pku.edu.cn/teachers/sunm

2017年10月19日

课程内容

- 堆与优先队列
- 哈夫曼树

堆与优先队列

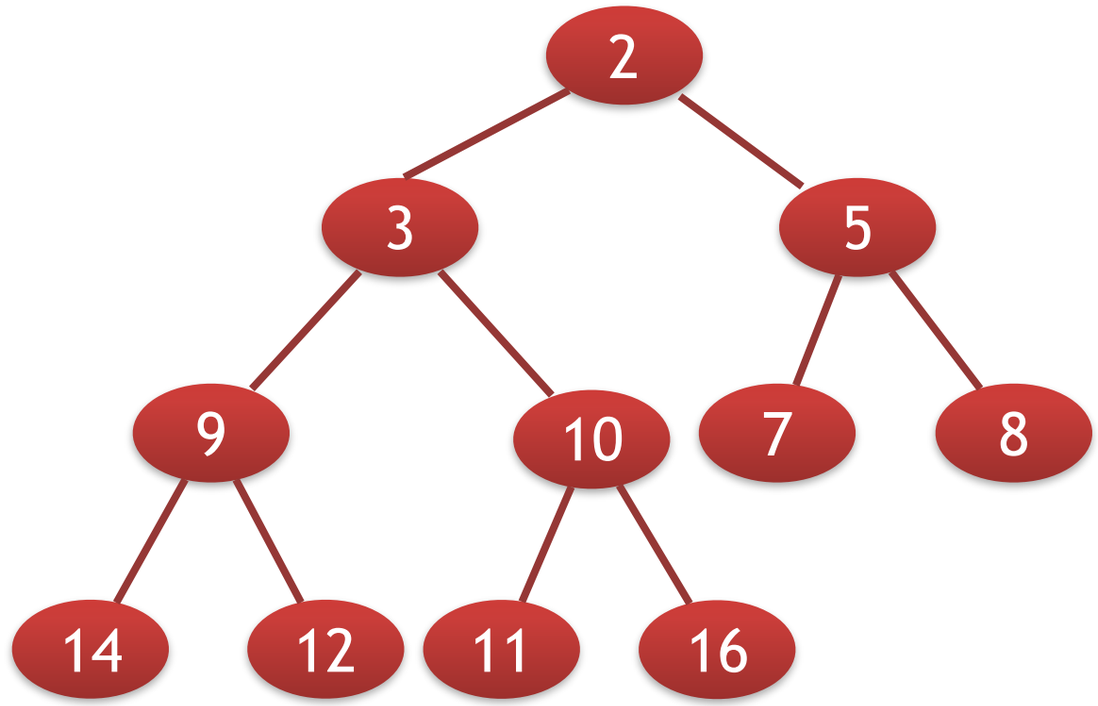
- 介绍一种特殊的完全二叉树
- 这种二叉树的顺序存储表示——堆
- 优先队列的概念及使用堆的实现方法

一种特殊的完全二叉树

每个结点的值都小于（或者都大于）它的左右子树的根结点的值。

这种二叉树的广度优先周游序列顺序表示中，用于存放结点的顺序表具有如下性质：

$$\begin{cases} k_i \le k_{2i+1} \\ k_i \le k_{2i+2} \end{cases}$$



0	1	2	3	4	5	6	7	8	9	10
2	3	5	9	10	7	8	14	12	11	16

堆的定义

- **n个元素的序列** $K=(k_0,k_1,\cdots,k_{n-1})$ **称为堆，当且仅当满足条件：**

- $$(1)\begin{cases}k_i \geq k_{2i+1}\\ k_i \geq k_{2i+2}\end{cases} \quad \textbf{或} \quad (2)\begin{cases}k_i \leq k_{2i+1}\\ k_i \leq k_{2i+2}\end{cases}$$

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

- **这一条件简称为堆序性。**

大根堆与小根堆

- 满足条件（1）的堆，在对应的完全二叉树中等价于：每个子二叉树的根均大于等于其左、右子结点；因此在这个堆中，根结点是最大结点。故则称为大根堆。

- 满足条件（2）的堆，在完全二叉树中等价于：每个子二叉树的根均小于等于其左、右子结点。因此在这个堆中，根结点是最小结点。故称为小根堆。

本节课主要讨论小根堆！

优先队列

- 优先队列是一种常见的抽象数据类型，它遵循“最小元素先出”的原则。

优先队列的基本操作有三个：

向优先队列里插入一个元素；

在优先队列中找出最小元素；

删除优先队列中最小元素。

优先队列的抽象数据类型

ADT PriorityQueue **is**

Operations

PriorityQueue createEmptyPriQueue(void)

创建一个空优先队列。

int isEmpty(PriorityQueue S)

若S为空，则返回1，否则返回0。

void add(PriorityQueue S, DataType e)

向S中添加元素e。

DataType min(PriorityQueue S)

返回S中的最小元素。

void removeMin(PriorityQueue S)

删除S中的最小元素。

end ADT PriorityQueue

优先队列

- 优先关系代表了数据的某种性质，如用于描述
  - 各项工作的计划开始时间（实际中，模拟中都可能使用）
  - 一个大项目中各种工作任务的急迫程度（或截止期）
  - 银行客户的诚信评估，用于决定优先贷款，等等
- 优先队列的操作也很简单，应包括
  - 创建，判断空（还可以有清空内容、确定当前元素个数等）
  - 插入元素，访问和删除优先队列里（当时最优先）的元素

优先队列的实现

- 最常用来表示优先队列的方法是（小根）堆。由于堆与完全二叉树的内在联系，下面表示优先队列的定义与二叉树的顺序表示基本一样：

```
struct PriorityQueue {
  int MAXNUM;                    /*元素个数的上限 */
  int n;                         /*实际元素个数*/
  DataType *pq;                  /*存放元素的数组的指针*/
};                               /*优先队列类型*/

typedef struct PriorityQueue * PPriorityQueue;
                                 /*指向优先队列的指针类型*/
```

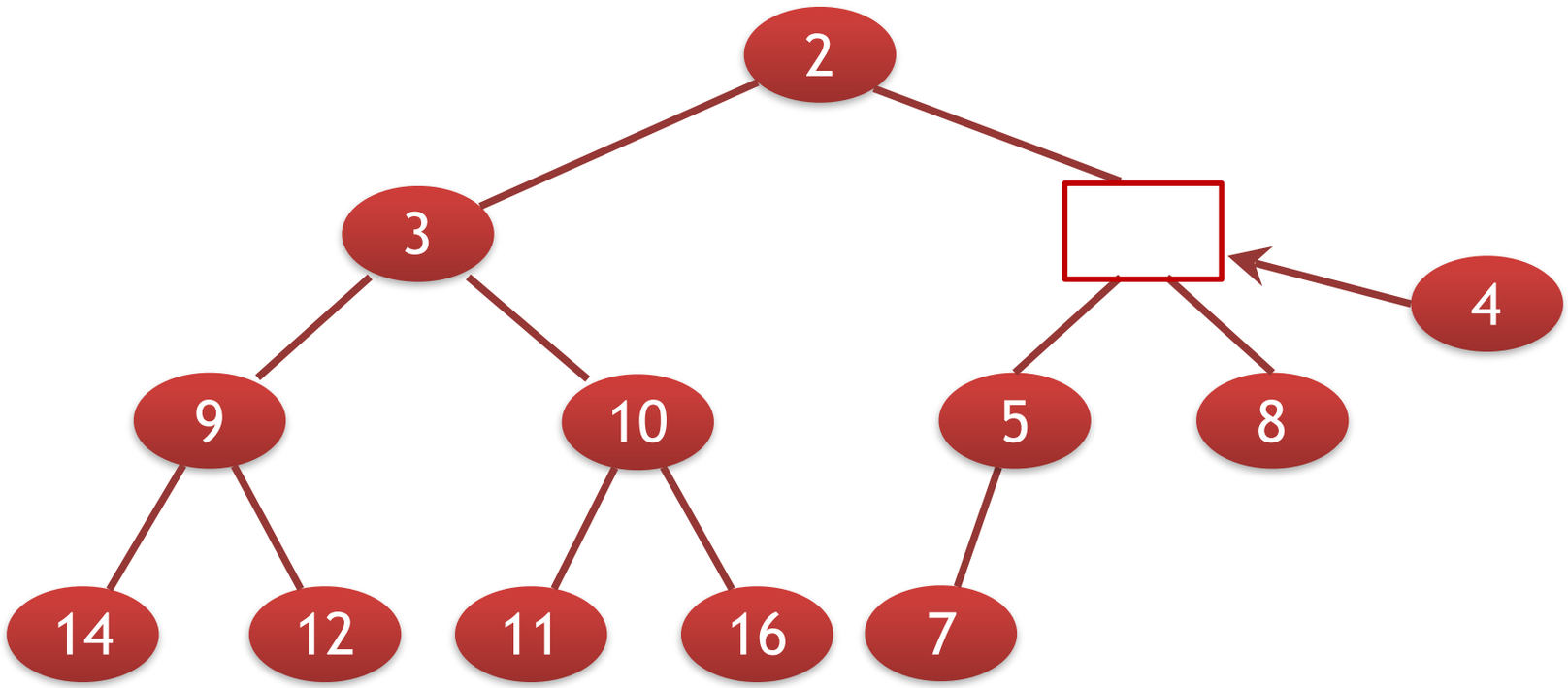
操作实现

- **插入**
  要把一个新元素加入优先队列。

  根据堆的表示特点，显然必须有某个元素要放到紧接着原有数组元素后面的位置，并且还需要保持堆序性。

  一个实现方法是：假设把新元素先放在已有元素后，然后通过反复比较，必要时交换该结点与对应的父结点，直到堆序性重新被满足为止。

向优先队列中插入4

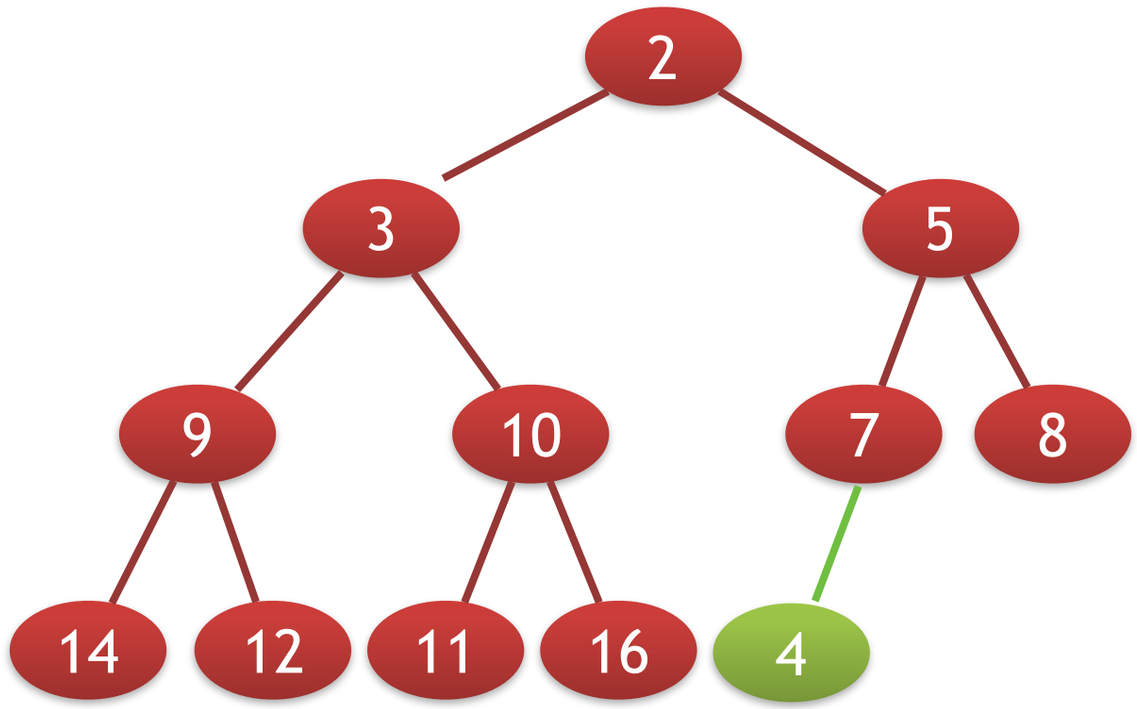


一种特殊的完全二叉树

每个结点的值都小于（或者都大于）它的左右子树的根结点的值。

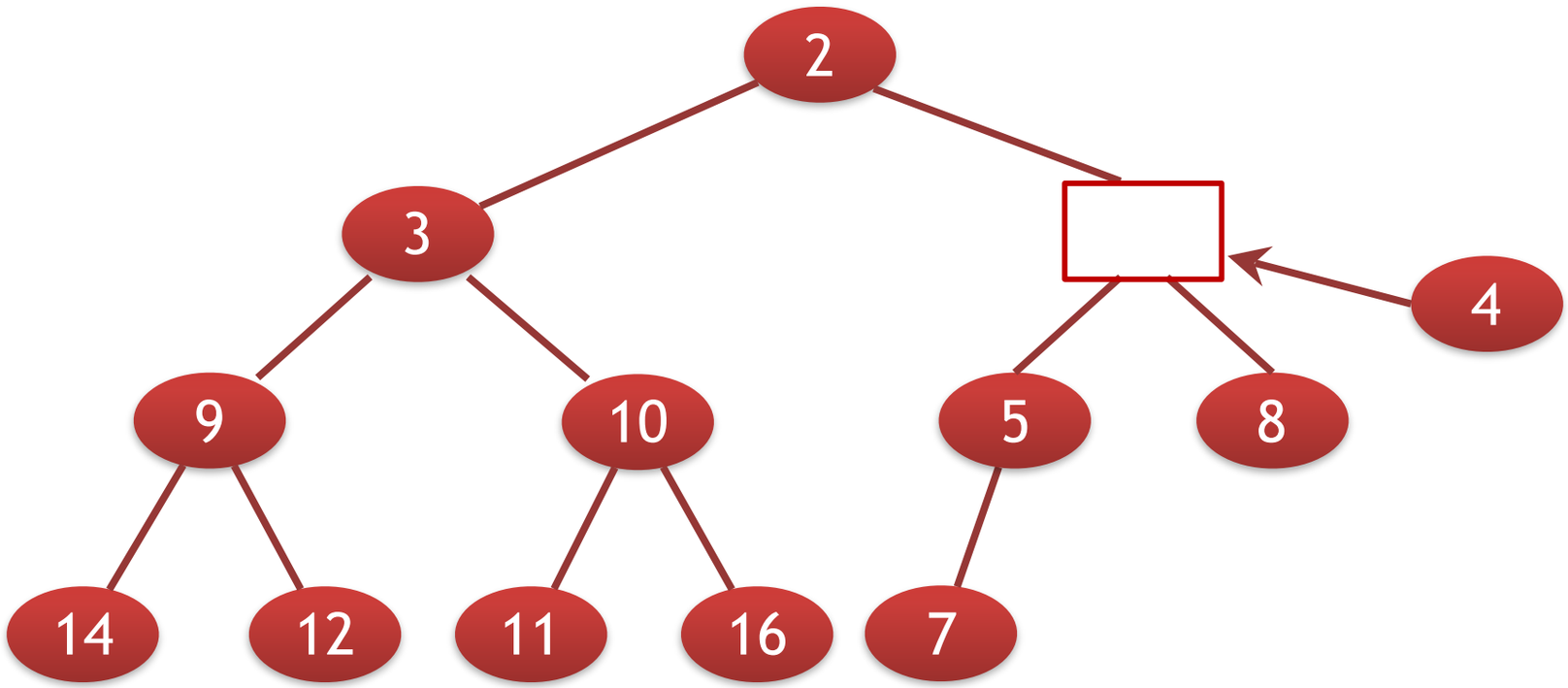
这种二叉树的广度优先周游序列顺序表示中，用于存放结点的顺序表具有如下性质：

$$\begin{cases} k_i \le k_{2i+1} \\ k_i \le k_{2i+2} \end{cases}$$



0	1	2	3	4	5	6	7	8	9	10	11
2	3	5	9	10	7	8	14	12	11	16	4

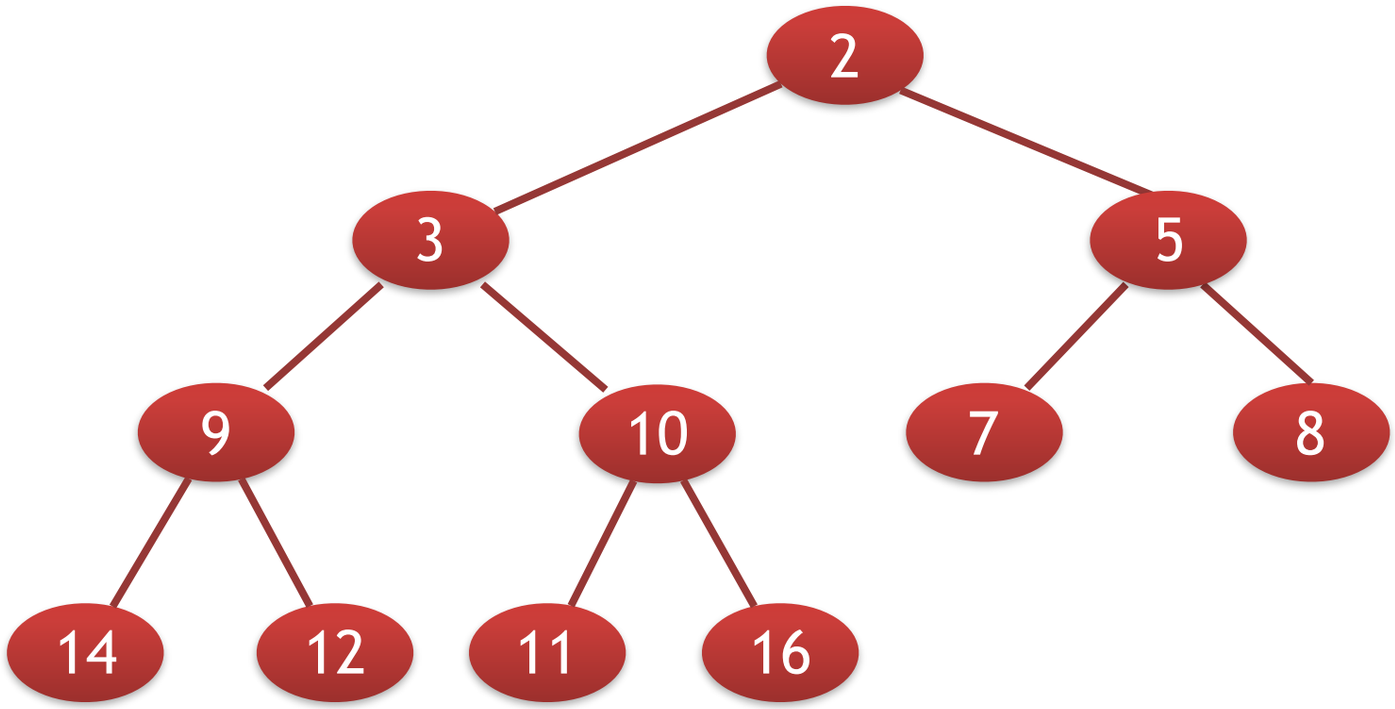
向优先队列中插入4



向优先队列中插入x(保持堆序性)

```
void add_heap(PPriorityQueue papq, DataType x) {
    int i;
    if (papq->n >= papq->MAXNUM){
      printf("Full!\n");
      return;
    }
    for (i = paqu->n; i > 0 && papq->pq[(i - 1) / 2] > x; i = (i - 1) / 2)
       papq->pq[i] = papq->pq[(i - 1) / 2];  /*空位向根移动找插入位置*/
    papq->pq[i] = x;                                  /*将x插入*/
    papq->n++;
}
```

从优先队列中删除2



最小结点的删除

- 在被删后，根结点形成一个空位，这时先考虑能否把处在堆中最后位置的结点填入这里。
- 由于这样做可能破坏堆序性，所以选择这个元素与根的两个子结点三者中最小的结点填入，选择的结果可能使得原来的空位向叶结点方向传递。
- 如此反复交换，最终到堆中最后结点小于等于空位的两个子结点时，将最后结点填入这个空位。

最小结点的删除

```
void removeMin_heap(PPriorityQueue papq) {
    int s, i, child;     DataType  temp ;
    if (isEmpty_heap(papq)) {printf("Empty!\n"); return;}
    s = --papq->n;                                          /*先删除*/
    temp = papq->pq[s];                                     /*把最后元素移到temp*/
     i =0;        child = 1;
    while (child < s) {                                     /*找最后元素存放的位置*/
        if (child < s - 1 && papq->pq[child] > papq->pq[child + 1])
          child++;                                          /*选择比较小的子结点*/
        if (temp > papq->pq[child])                         /*空位向叶结点移动*/
         { papq->pq[i] = papq->pq[child]; i = child; child = 2 * i + 1; }
        else break;                                         /*已经找到适当位置*/
     }
    papq->pq[i] = temp;                                     /*把最后元素填入*/
}
```

算法代价分析

- 在删除操作过程中，由于对每层最多只需要做2次比较，而且循环是从树根到树叶进行的，所以这个删除程序的复杂性也是O(log n)。
- 判断优先队列是否为空和取优先队列中的最小元素都非常容易实现。时间代价均为O(1)。

优先队列的应用

- **离散事件的模拟**

  利用优先队列设计一个对机场的模拟系统。该机场有两条跑道，飞机从空中飞来申请着陆，同时地上的飞机申请起飞。

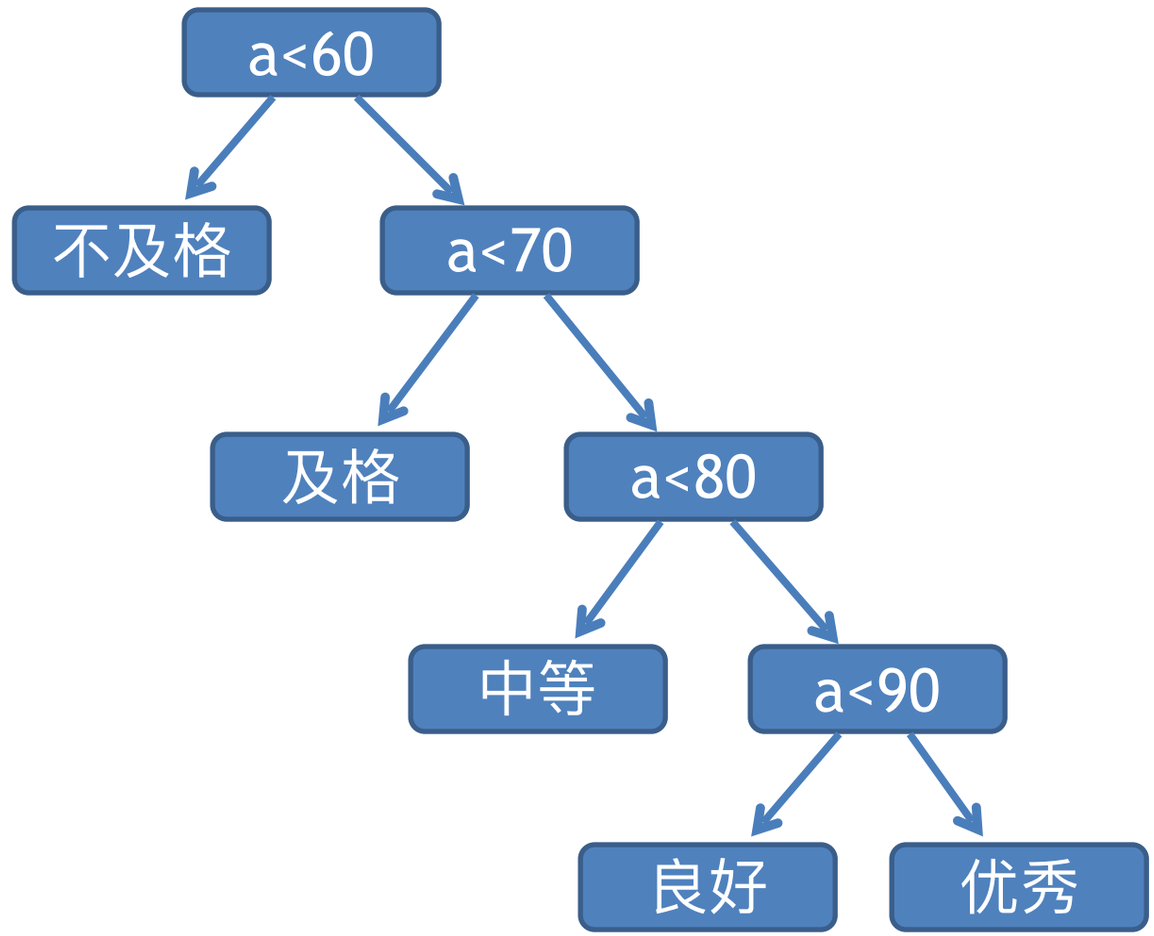
  经常出现在操作系统的各种调度算法中!

- **（堆）排序**

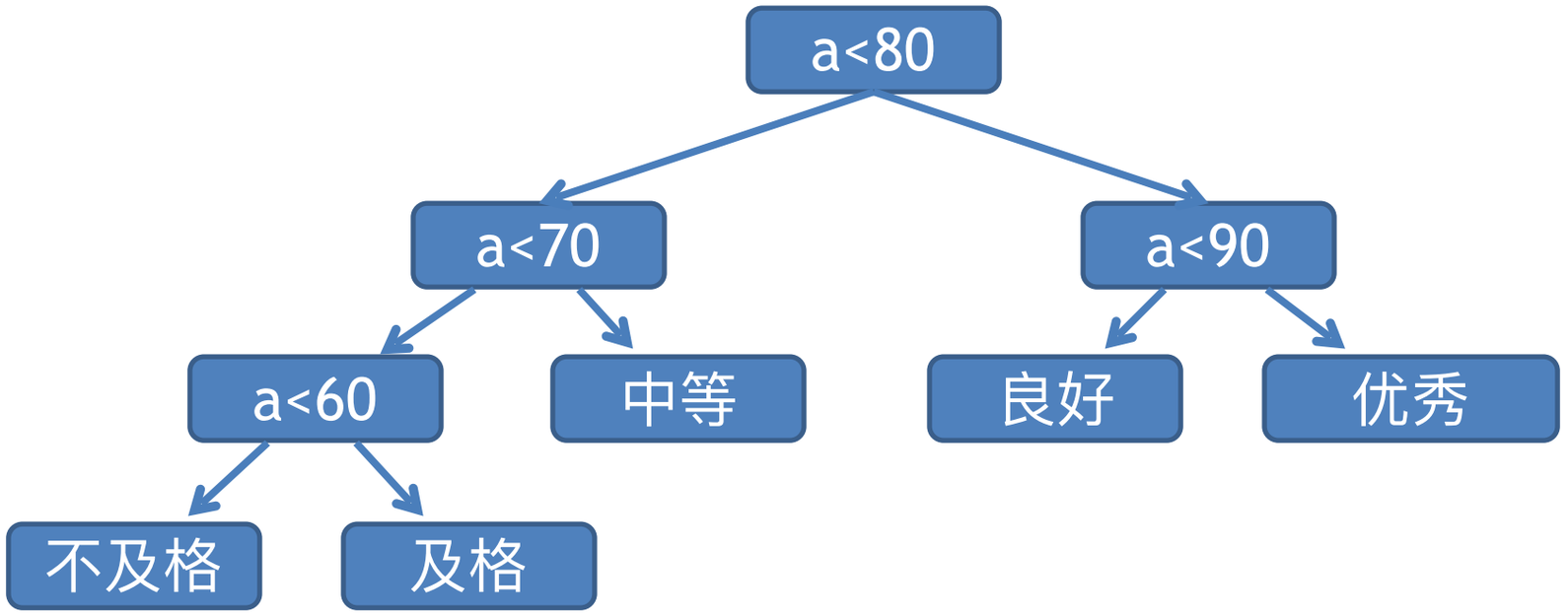
哈夫曼算法及其应用

- 一种特殊的扩充二叉树
- 带权的外部路径长度
- 哈夫曼树
- 哈夫曼算法
- 哈夫曼编码

```
if a<60
    b="不及格"
else:
    if a<70
        b="及格"
    else:
        if a<80
            b="中等"
        else:
            if a<90
                b="良好"
            else:
                b="优秀"
```



分数	0-59	60-69	70-79	80-89	90-100
所占比例	5%	15%	40%	30%	10%



带权的外部路径长度

- 在扩充二叉树中，若l_i为从根到第i个外部结点的路径长度，m为外部结点的个数。则外部路径长度

$$E=\sum_{i=1}^{m} l_i 。$$

- 如果每个外部结点都带有权值，且第i个外部结点的权值为w_i，则

$$WPL=\sum_{i=1}^{m} w_i l_i ,$$

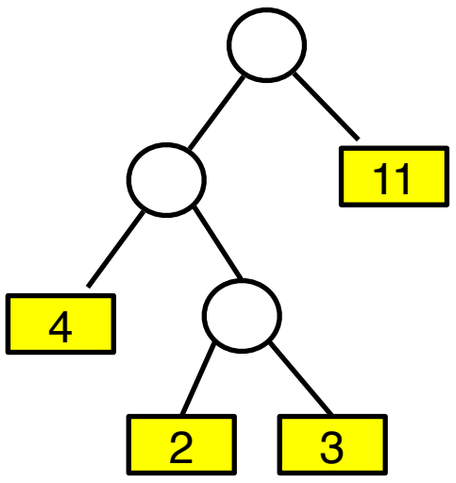
称做扩充二叉树的**带权的外部路径长度**。

哈夫曼树（最优二叉树）

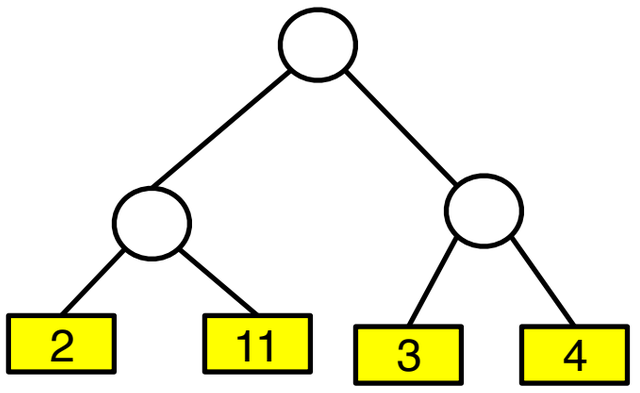
给定一组(无序)实数$\{w_1, w_2, .., w_m\}$，扩充二叉树T称为**哈夫曼树**或**最优二叉树**，如果T满足：

(1) 有m个外部结点，且分别以w_i为(i=1,…,m)其权值；

(2)T是满足条件(1)的扩充二叉树中带权的路径长度WPL最小的。



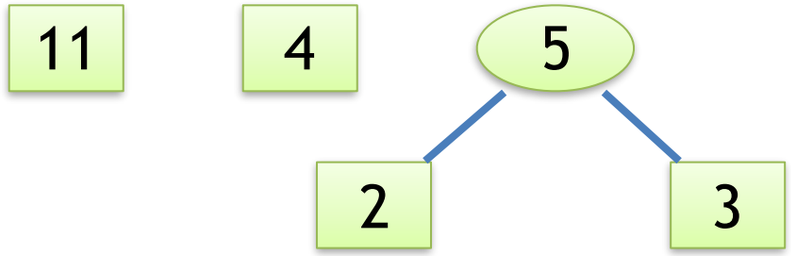
WPL= 1×11 + 2×4 + 3×2 + 3×3 = 34

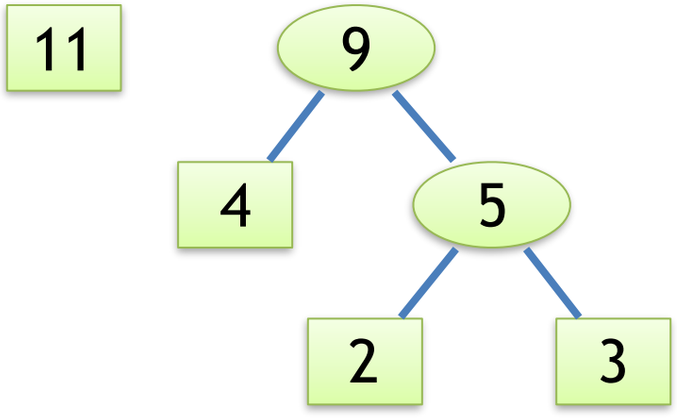


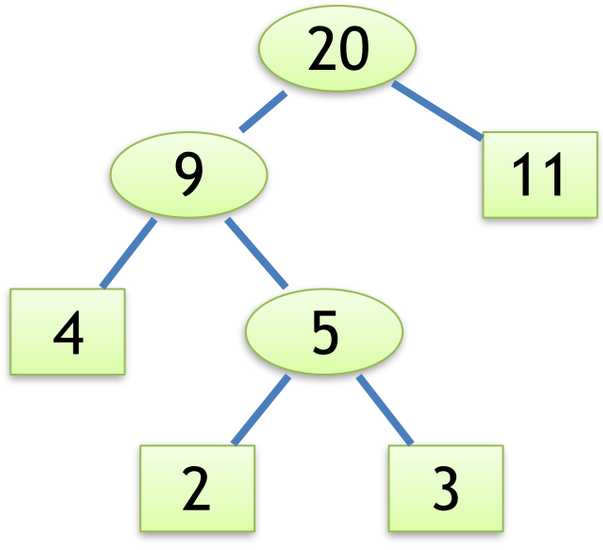
WPL=40

构造哈夫曼树示例

- w＝ {11，4，3，2}







构造哈夫曼树的基本思想

(1) 由给定的m个权值w={w_1, w_2,.., w_m}，构造包含m棵二叉树的集合F = {T_1,T_2,…,T_m}，其中二叉树 T_i 只含权为 w_i 的根结点；

(2) 从F中选取两棵根结点权最小和次最小的树作为左、右子树，构造一棵新二叉树，其根结点的权值为两棵子树的根结点权值之和；

(3) 从F删除所选的两棵树，把新构造的二叉树加入F；

(4) 重复(2)和(3)，直到F中只含一棵树为止。

数据结构设计

在这里我们介绍一种存储表示，该存储结构是在二叉树的llink和rlink基础上增加一个父结点的指针，并且所有结点顺序存放在一个顺序表中。在顺序表中，每个结点的结构由四部分组成：

数据结构设计

```
struct HtNode  {                    /* 哈夫曼树结点的结构 */
    int ww;
    int parent,llink,rlink;
};
struct HtTree{                       /* 哈夫曼树结构 */
    int    m;                        /* 外部结点的个数 */
    int    root;                     /* 哈夫曼树根在数组中的下标 */
    struct HtNode  *ht;              /*存放2*m-1个结点的数组 */
};
typedef  struct HtTree  *PHtTree; /* 哈夫曼树类型的指针类型 */
```

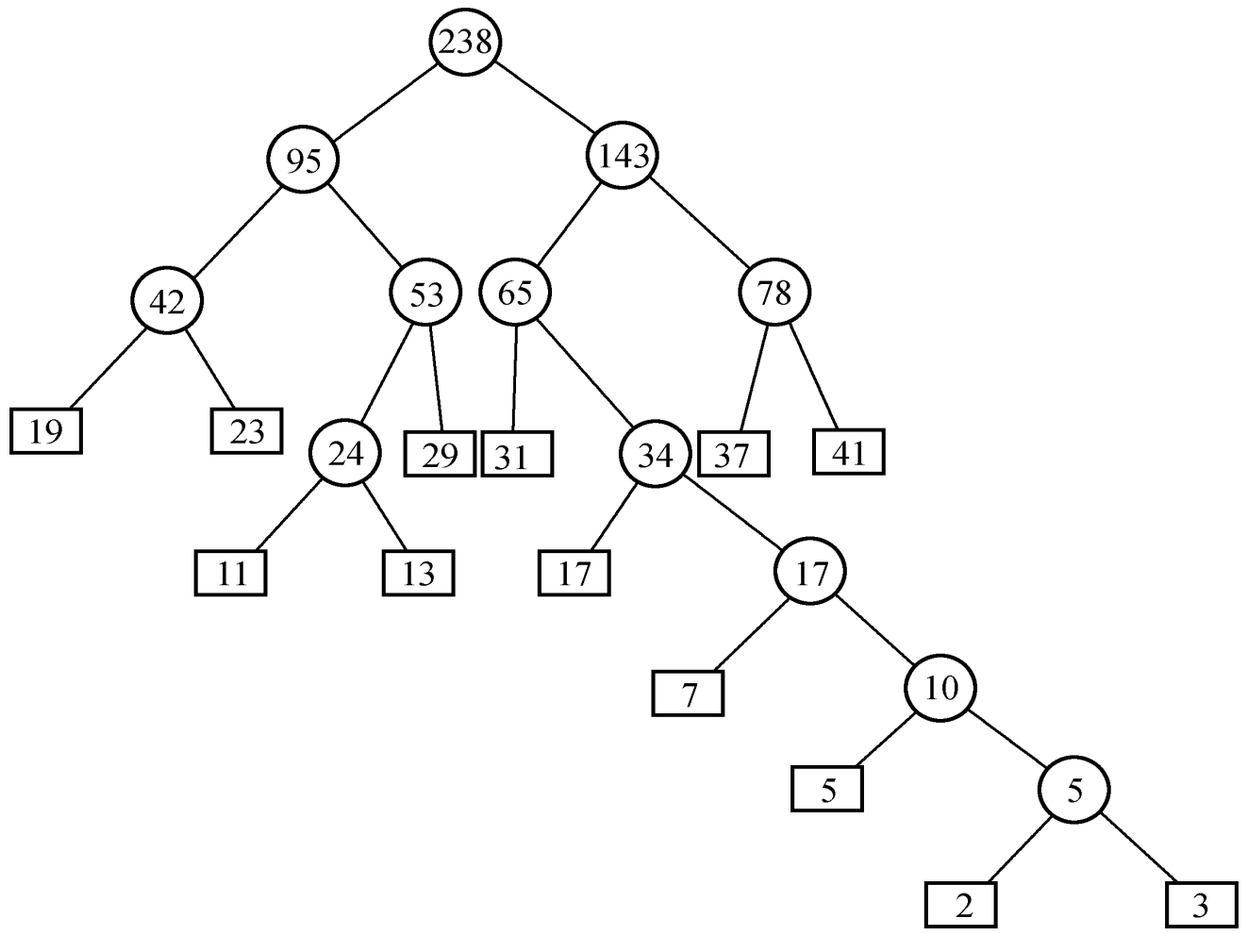
哈夫曼算法

```
PHtTree huffman(int m, int *w) {
    PHtTree  pht;       int i,j,x1,x2,m1,m2;
    pht = (PHtTree)malloc(sizeof (struct HtTree));     /* 分配空间 */
    if (pht==NULL)   { printf("Out of space!! \n");      return pht;}
    pht->ht=( struct HtNode) malloc(sizeof (struct HtNode)*(2*m-1));
    if (pht==NULL)   { printf("Out of space!! \n");      return pht;}
    for( i=0; i<2*m - 1; i++ ) {/* 置ht数组初态 */
     pht->ht[i].llink = -1;  pht->ht[i].rlink = -1;  pht->ht[i].parent = -1;
     if (i<m) pht->ht[i].ww = w[i]; else pht->ht[i].ww = -1;
     }
    for( i=0; i < m - 1; i++ )          {/* 每循环一次构造一个内部结点 */
      m1 = MAXINT;    m2 = MAXINT;             /* 相关变量赋初值 */
      x1 = -1;          x2 = -1;
      for(j=0;j<m+i;j++) /* 找两个最小权的无父结点的结点 (m1<m2)*/
            if (pht->ht[j].ww<m1 && pht->ht[j].parent==-1)
                    { m2 = m1;x2 = x1;  m1 = pht->ht[j].ww;  x1 = j;}
            else if (pht->ht[j].ww<m2 && pht->ht[j].parent==-1)
                     {m2 = pht->ht[j].ww; x2 = j;}
      pht->ht[x1].parent = m + i;      pht->ht[x2].parent = m + i;
      pht->ht[m+i].ww = m1 + m2;
        pht->ht[m+i].llink = x1;  pht->ht[m+i].rlink = x2;  /* 构造内部结点 */
      }
    pht->root = 2*m - 2;    return pht;
}
```

本算法具有平方复杂性。能加快其速度吗?

哈夫曼树的构造

- 对于一组权值w = {2，3，5，7，11，13，17，19，23，29，31，37，41}，按照上述算法构造出的哈夫曼树如下图所示：



哈夫曼树的应用——哈夫曼编码

- 设
  - d={d_1, d_2,.., d_n}为需要编码的字符集合，
  - w={w_1, w_2,.., w_n}为d中各字符出现的频率。
- d中字符的二进制编码称为最优前缀码，如果使得：

  (1)按照给出的编码传输文件时，通讯编码平均总长最短；

  (2)若$d_i \neq d_j$，则d_i的编码不可能是d_j的编码的开始部分(前缀)。

最优前缀码的构造:

利用哈夫曼树构造哈夫曼编码

哈夫曼编码

- 通过构造哈夫曼树，实现哈夫曼编码：

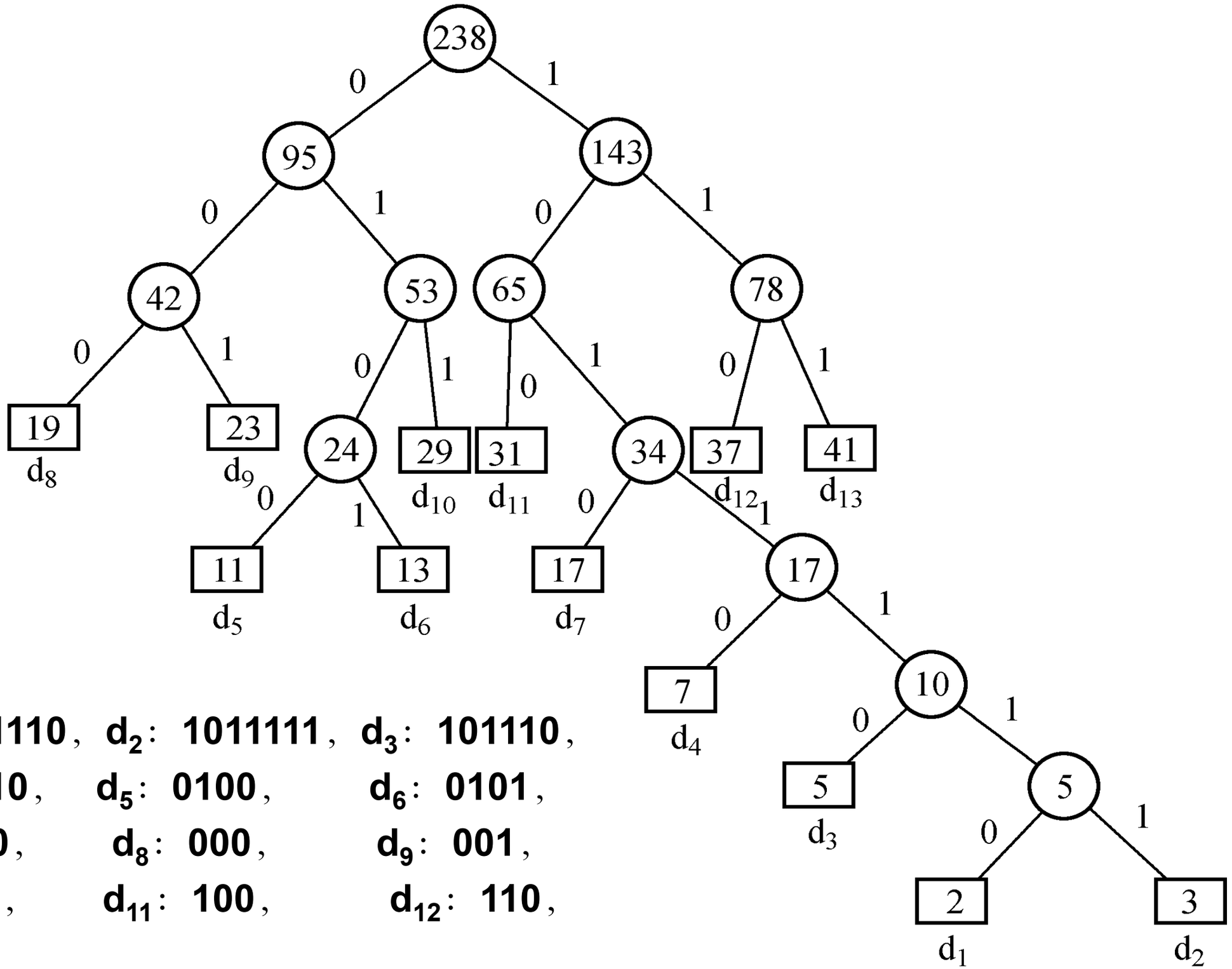
  (1)以字符 $d_1, d_2,.., d_n$为外部结点标注，把$w_1, w_2,.., w_n$分别作为这n个外部结点的权，构造一棵哈夫曼树。

  (2)在得到的哈夫曼树中，将所有从一个结点引向其左子结点的边标上二进制数字0；引向其右子结点的边标二进制数字1。

  (3)以从根结点到一个叶结点的路径上的二进制数字序列，作为这个叶结点的字符的编码，就是这个叶结点所代表字符的最优前缀编码，称为哈夫曼编码。

例子

- d = { d_1 , d_2, …, d_{13} }
  w = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41 }
- 利用哈夫曼算法构造其哈夫曼树。



d_1: 1011110, d_2: 1011111, d_3: 101110,
d_4: 10110, d_5: 0100, d_6: 0101,
d_7: 1010, d_8: 000, d_9: 001,
d_{10}: 011, d_{11}: 100, d_{12}: 110,
d_{13}: 111

解码方法

- 只要从二叉树的根结点开始，用需要解码的二进制位串，从头开始与二叉树根结点到子结点边上标的0、1相匹配，确定一条到达树叶结点的路径。一旦到达树叶结点，则译出一个字符。然后再回到根结点，从二进制位串中的下一位开始继续解码。

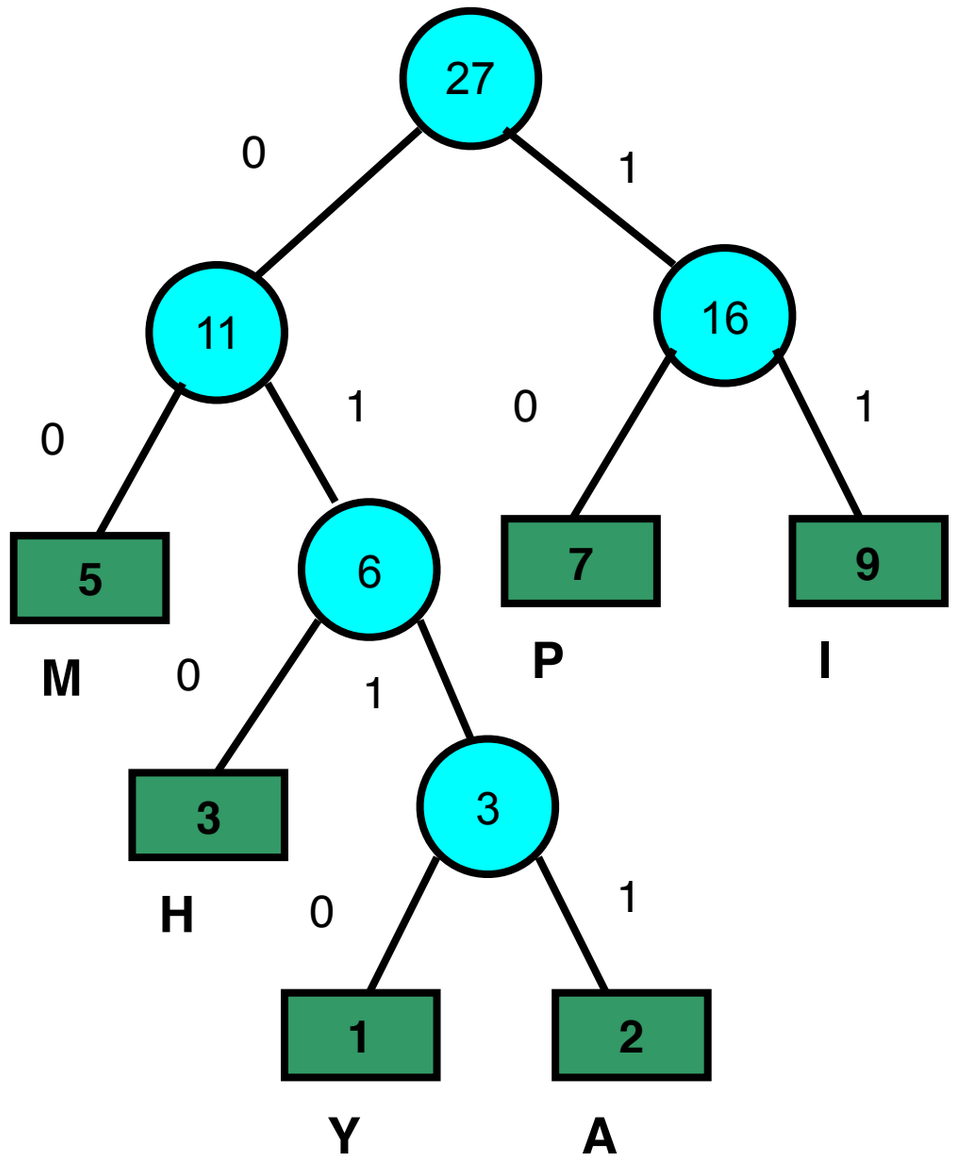
d={Y,A,H,M,P,I}　　w={1,2,3,5,7,9}

哈夫曼编码：

Y : 0110　A : 0111

H : 010　M : 00

P : 1 0　I : 11



11011100010011110100110

哈夫曼树的应用——二路归并排序

- 假设现在有m个已经排序的文件{d_1，d_2，…，d_n}，每个文件包含的记录个数对应为{w_1，w_2，…，w_n}；可以采用两两合并的方法，把所有文件的记录合到一个大文件中，使这个文件中的记录全部排序。

- 问:采用怎样的合并次序才能使得移动记录个数最少?

- 答案：按照哈夫曼树的结构从外部结点到根结点逐层进行合并，一定是一种最佳的（但并非唯一的）合并顺序。

如何证明哈夫曼树最优?

归纳法可证。具体证明过程可见：

algoviz.org/OpenDSA/Books/OpenDSA/html/HuffProof.html

本讲重点

- 堆的概念，与二叉树的联系
- 优先队列的概念，表示，实现和应用
- 哈夫曼树的概念，算法和应用

- 二叉树的应用非常广泛，这里仅仅是两个简单的例子。由于它们使用广泛，非常有名，所以特别在这里介绍。值得注意的是：在这两个应用中实际仅仅使用了二叉树的观点。（没有应用二叉树的ADT）
- 后面还会看到许多的应用二叉树的例子。