第六章 二叉树

part 2: 二叉树的应用

张史梁
slzhang.jdl@pku.edu.cn

内容提要

- 二叉树基础
  - 树与二叉树的基本概念
  - 二叉树的存储结构
  - 二叉树的周游算法
  - 建立一个二叉树
- 二叉树的应用
  - 哈夫曼树
  - 二叉检索树/排序树
- 树与树林

问题的提出

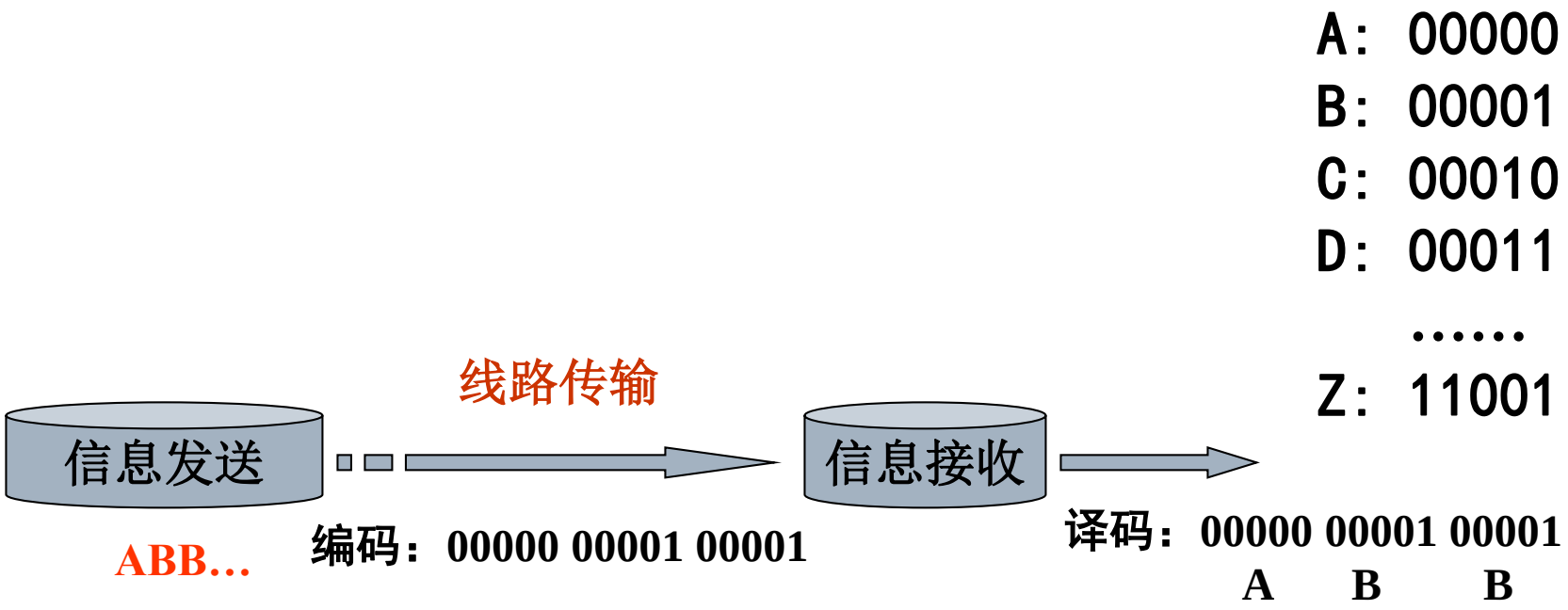
- 数据通信的二进制编码问题描述：
  - 设需要编码的字符集合$d = \{ d_1, d_2, \ldots, d_m \}$，$w = \{ w_1, w_2, \ldots, w_m \}$ 为d中各种字符出现的频率，要对d里的字符进行二进制编码，使得

① 通信编码总长最短

② 无二义性：若$d_i \neq d_j$，则d_i的编码不可能是d_j的编码的前缀。（这样就使得译码可以一个字符一个字符地进行，不需要在字符与字符之间添加分隔符）

解决方案-1

- 定长编码 – 所有的字符都具有相同的编码长度
  - 编码简单，译码更简单，用于字符等概率出现时。
  - 如26个字符：A, B, C, D, …, Z



解决方案-1

- 定长编码的优点
  - 编码简单，译码更简单，用于字符等概率出现时
- 定长编码的缺点
  - 对于各个字符出现频率不等概率情况(实际情况）下，定长编码不是最好的，
  - 如果能够使得经常出现的字符编码长度短，不经常出现的可以长一些，那么整个信息串的编码长度会减少。

解决方案-2

- 不定长编码：
- 假定字符出现频率分布如下$\{p_1, p_2, \ldots, p_m\}$，根据减少信息串编码长度的要求，频率最大的字符其编码位数应该最小。假定各个字符得到的编码位数为$\{L_1, L_2, \ldots, L_m\}$，则总的信息串的长度为：

$$AL = \sum_{i=1}^{m} P_i L_i$$

解决方案-2

- 不等长编码的难点
  - 为了译码时不出现二义问题，必须保证任何字符的编码都不是其它字符编码的前缀。
  - 如A: 0，B: 10，C: 010，这样如果接收到 01010 序列，如何译码？是ABB还是CB？ 无法确定，因此必须保证不出现上述二义问题。

- 不等长编码问题总结：
  - 最经常出现的字符编码最短；
  - 避免出现二义性问题。

两个基本概念

- “外部路径长度”E:在扩充的二叉树里从根到每个外部结点的路径长度之和。

$$E=\sum_{i=1}^{m} l_i$$

  - 其中，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。
- 设扩充二叉树具有m个带权值的外部结点，那么从根结点到各个外部结点的路径长度与相应结点权值的乘积的和，叫做扩充二叉树的带权的外部路径长度。

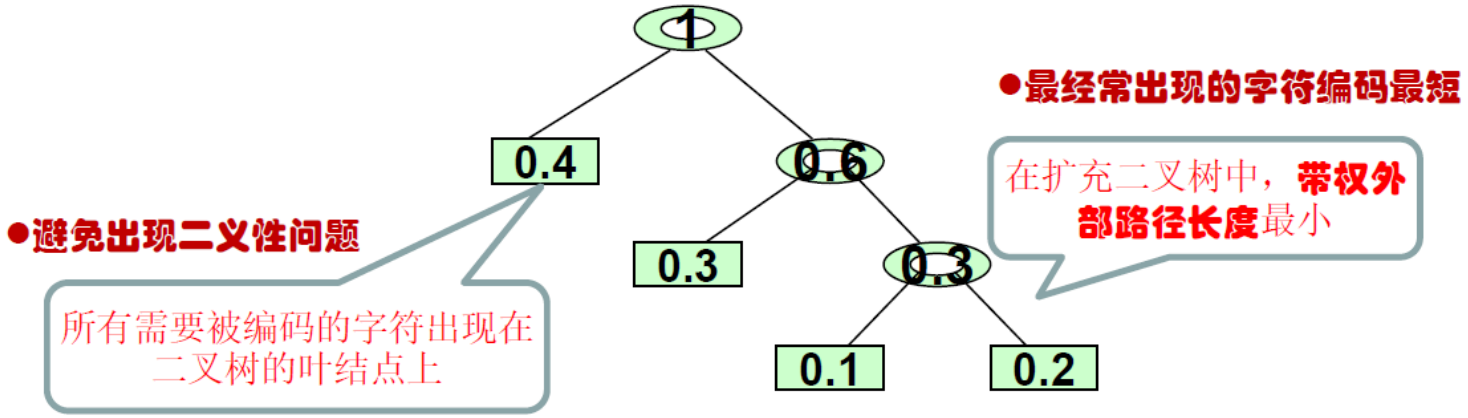
$$WPL=\sum_{i=1}^{m} w_i l_i \qquad AL=\sum_{i=1}^{m} P_i L_i$$

  - 其中，w_i是第i个外部结点的权值，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。

概述

- 构造哈夫曼树是不定长编码的一种解决方案
  - 所构造哈夫曼树是一棵二叉树
  - 利用根节点到叶结点的路径进行编码

- 例对字符集{A(0.1), B(0.2), C(0.3), D(0.4)}



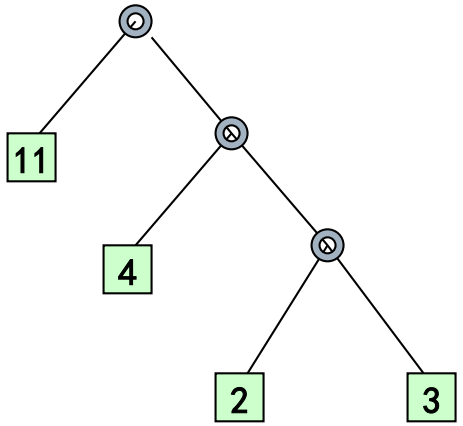
哈夫曼树

- 哈夫曼（Huffman）树定义：

  设有一组实数 $\{w_1, w_2, w_3, \ldots, w_m\}$，现要构造一棵以 w_i（$i = 1, 2, \ldots, m$）为权值

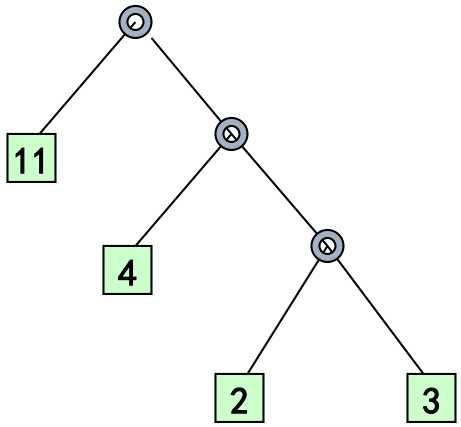
  的m个外部结点的扩充的二叉树

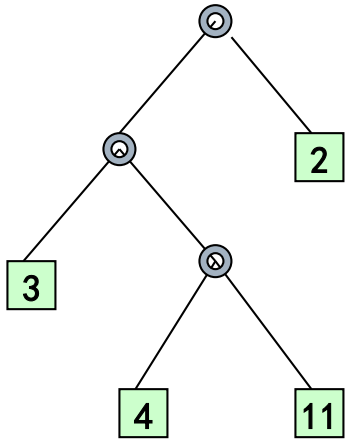
  使得带权的外部路径长度最小。

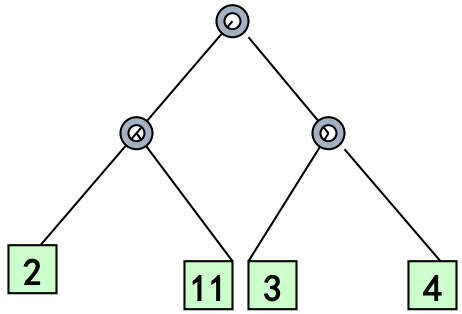


带权外部路径长度

- 设字符集{A, B, C, D}出现的频率（对应的权值）分别为{ 2, 3, 4, 11}
- 我们可以形成下列三种二叉树。它们的带权外部路径长度分别为：







1×11+2×4+3×(2+3)=34　　**2×3+3×(4+11)+1×2=53**　　**2×(2+11+3+4)=40**

- 哈夫曼树的规律：权越大的叶子离根越近，则二叉树的带权外部路径长度就越小。

哈夫曼树

- 可以看到，假定字符出现的概率为加权值，解决不定长编码问题实质上就是建立一棵哈夫曼树的过程。

- 如何构造一棵哈夫曼树呢？
  - 哈夫曼最早给出了一个带有一般规律的算法，又称为哈夫曼算法。

哈夫曼算法

- 如何构造一棵哈夫曼树呢？
  - 哈夫曼最早给出了一个带有一般规律的算法，又称为哈夫曼算法。
- 哈夫曼算法的基本思想：

① 根据给定的m个权值$\{w_1,w_2,\ldots, w_m\}$，构成m棵二叉树的集合$F=\{T_1,T_2,\ldots,T_m\}$，其中每一棵二叉树T_i中只有一个带权为w_i的根结点，其左右子树为空。

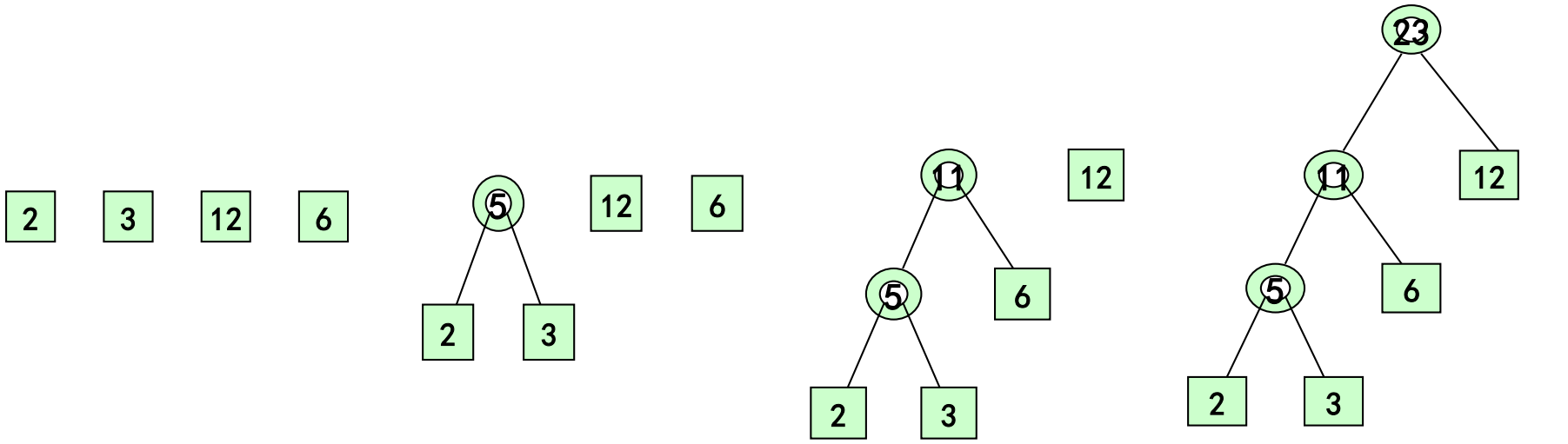
② 在F中选取两棵权值最小的树作为左右子树以构造一棵新的二叉树，且新二叉树的根结点的权值为其左右子树根结点权值之和。

③ 在F中删除这两棵树，同时将新得到的二叉树加入F。

④ 重复(2)和(3)，直到F中只含一棵树为止。

哈夫曼树构造过程

设字符集{A, B, C, D}出现的频率（对应的权值）分别为{ 2, 3, 4, 11}



4棵只有根的二叉树　　2、3合并得到3棵二叉树　　5、6合并得到2棵二叉树　　11、12合并得到1棵二叉树

左右选择不同，得到的HuffMan树形态不同，但WPL相同。

哈夫曼树的顺序表示

- 假定外部结点个数为m，则内部结点个数必为m-1，因此最后得到的HuffMan树必定有2m-1个结点。因此，用2m-1个元素的一维数组就可以存储该HuffMan树。
- 结点结构：

ww	parent	llink	rlink

  - ww: 以该结点为根的子树中所有外部结点的加权和。
  - parent: 父结点在数组中的存储位置（下标），根无父，设为-1。
  - llink: 左子节点位置；对于外部结点，设为-1。
  - rlink: 右子节点位置；对于外部结点，设为-1。

哈夫曼算法实现

- 每个元素表示一个结点，前m个存储外部结点，后m-1个用于内部结点。
- struct HtNode /*哈夫曼树结点的结构*/

```
{   int ww;
    int parent;
    int llink, rlink;   };
```

- struct HtTree /*哈夫曼树类型*/

```
{  struct HtNode ht[MAXNODE];
    int root;        };      // 哈夫曼树根在数组中的位置
```

- typedef struct HtTree *PHtTree; //指向哈夫曼树的指针

哈夫曼算法 -1

```
① /*结构定义*/
② #define MAXINT 2147483647
③ #define MAXNUM 50   /*数组w中最多容纳的元素个数,m<=MAXNUM*/
④ #define MAXNODE 100

⑤   /*哈夫曼树中的最大结点数,注意2*m-1<MAXNODE*/
⑥ struct HtNode /* 哈夫曼树结点的结构 */
⑦ {   int ww; /* 权值 */
⑧     int parent;   /* 父结点位置 */
⑨     int llink, rlink; /* 左右子女的位置 */
⑩ };
⑪ struct HtTree
⑫ {   struct HtNode ht[MAXNODE];
⑬     int root;    /* 哈夫曼树根在数组中的位置 */
⑭ };
⑮ typedef struct HtTree *PHtTree;   /*哈夫曼树类型的指针类型*/
```

哈夫曼算法 -2

```
① /* 构造具有m个叶结点的哈夫曼树*/
② /*变量w存放的是数组w中第一个元素w[0]的地址 */
③ PHtTree Huffman(int m, int *w)
④ {  PHtTree pht;
⑤    int i, j;
⑥    int x1, x2;   /* 存放权值最小的两个结点的位置 */
⑦    pht = (PHtTree)malloc(sizeof (struct HtTree)); /* 创建
   */
⑧    if (pht==NULL)
⑨    {  printf("Out of space!! \n");
⑩       return pht;
⑪    }
⑫    for( i=0; i<2*m - 1; i++ )   /* 置初态 */
⑬    {  pht->ht[i].llink = -1;
⑭       pht->ht[i].rlink = -1;
⑮       pht->ht[i].parent = -1;
⑯       if (i<m)   pht->ht[i].ww = w[i];
⑰       else       pht->ht[i].ww = -1;
⑱    }
```

```
19.  for( i=0; i < m - 1; i++ )   /* 每次构造一个内部结点 */
20.    {
21.       x1 = x2 = -1;
22.       Select(pht, m+i, &x1, &x2); /* 选最/次小值作为
     其左/右子女 */
23.       pht->ht[x1].parent = m + i;  /* 构造一个内部结
     点 */   // 设置两个子节点的父节点序号
24.       pht->ht[x2].parent = m + i;
25.       pht->ht[m+i].ww = pHT->ht[x1].ww + pHT-
     >ht[x2].ww;   // 设置父节点的权值ww
26.       pht->ht[m+i].llink = x1;  // 设置父节点
27.       pht->ht[m+i].rlink = x2;  //左右子节点序号
28.       pht->root = m+i;  //每次更新root，以最后一次为准
29.    }
30.    return pHT;
31. }
```

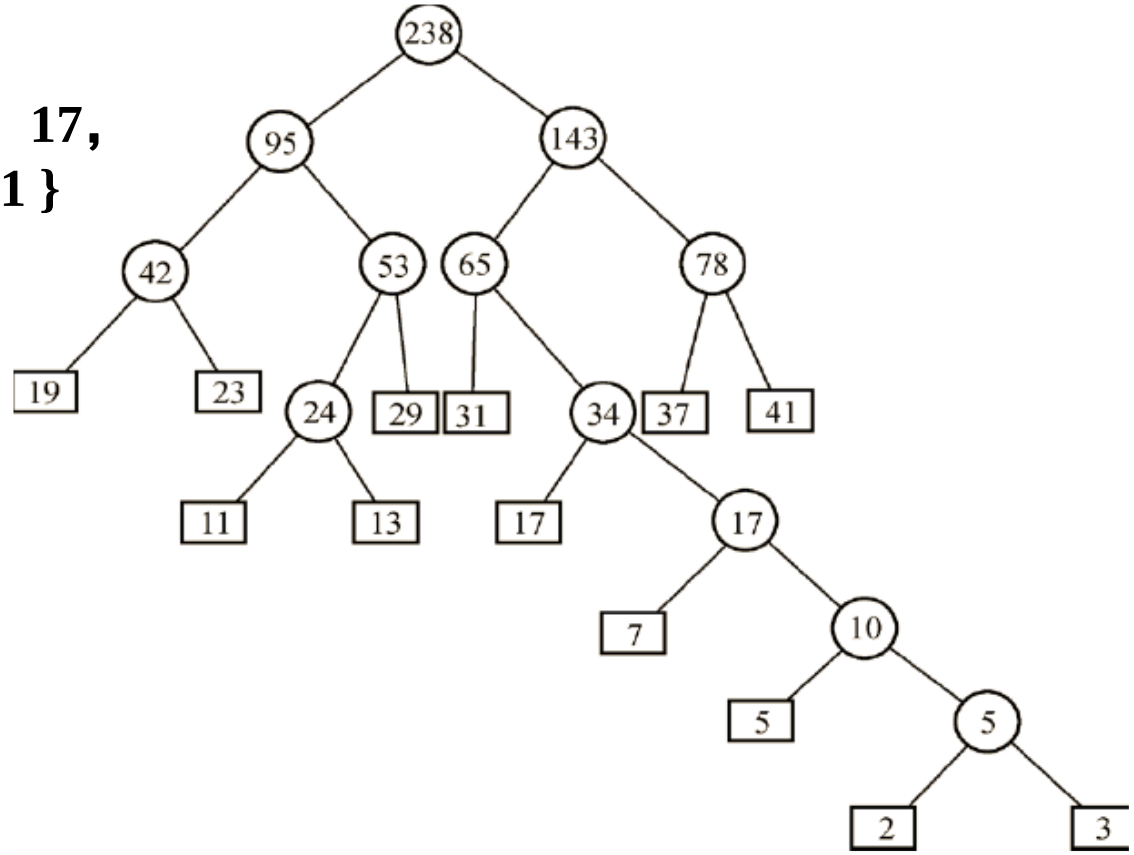
哈夫曼算法 -3

```
① /* 从哈夫曼树中pos前选取两个无父结点并且权值最小的结点，并把结点位置存放在x1和
   x2指向的空间中 */
② void Select(PHtTree pht, int pos, int *x1, int *x2)
③ {    int m1 = MAXINT, m2 = MAXINT;    /* 相关变量赋初值 */
④     /* m1存放最小的，m2存放次小的 */
⑤      for(j=0; j<pos; j++) /* 找两个最小权的无父结点的结点 */
⑥     {
⑦        if (pht->ht[j].ww<m1 && pht->ht[j].parent==-1)
⑧        {   m2 = m1; *x2 =* x1; /* 保留以前的最小到次小 */
⑨           m1 = pht->ht[j].ww; /* 改变新的最小 */
⑩           * x1 = j;                    // 记录最小节点位置
⑪        }
⑫        /* 如果大于最小的，它是否小于次小的？  */
⑬        else if(pht->ht[j].ww<m2 && pht->ht[j].parent==-1)
⑭        {  m2 = pht->ht[j].ww;   /* 改变次小的 */
⑮           *x2 = j;                    // 记录次小节点位置
⑯        }
⑰     }
⑱ }
```

哈夫曼算法实现

哈夫曼算法

w = {2，3，5，7，11，13，17，19，23，29，31，37，41}



教学网提供代码

ht的初态

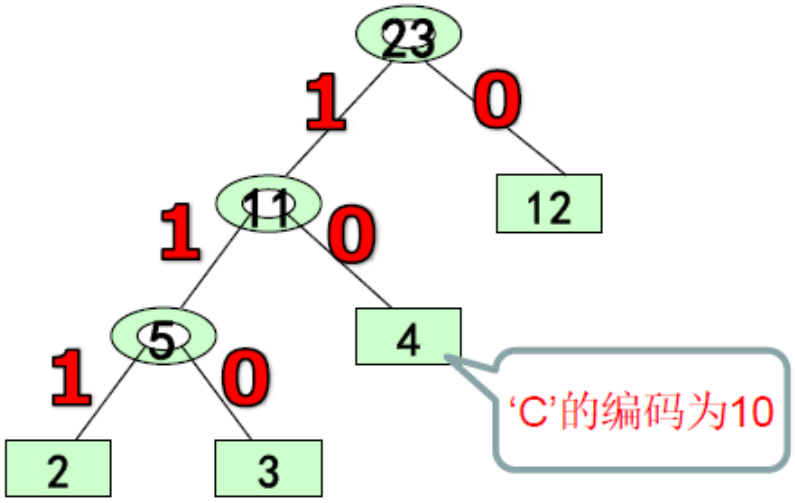
	ww	parent	llink	rlink
0	2	-1	-1	-1
1	3	-1	-1	-1
2	5	-1	-1	-1
3	7	-1	-1	-1
4	11	-1	-1	-1
5	13	-1	-1	-1
6	17	-1	-1	-1
7	19	-1	-1	-1
8	23	-1	-1	-1
9	29	-1	-1	-1
10	31	-1	-1	-1
11	37	-1	-1	-1
12	41	-1	-1	-1
13		-1	-1	-1
14		-1	-1	-1
15		-1	-1	-1
16		-1	-1	-1
17		-1	-1	-1
18		-1	-1	-1
19		-1	-1	-1
20		-1	-1	-1
21		-1	-1	-1
22		-1	-1	-1
23		-1	-1	-1
24		-1	-1	-1

ht的终态

	ww	parent	llink	rlink
0	2	13	-1	-1
1	3	13	-1	-1
2	5	14	-1	-1
3	7	15	-1	-1
4	11	16	-1	-1
5	13	16	-1	-1
6	17	17	-1	-1
7	19	18	-1	-1
8	23	18	-1	-1
9	29	19	-1	-1
10	31	20	-1	-1
11	37	21	-1	-1
12	41	21	-1	-1
13	5	14	0	1
14	10	15	2	13
15	17	17	3	14
16	24	19	4	5
17	34	20	6	15
18	42	22	7	8
19	53	22	16	9
20	65	23	10	17
21	78	23	11	12
22	95	24	18	19
23	143	24	20	21
24	238	-1	22	23

哈夫曼编码

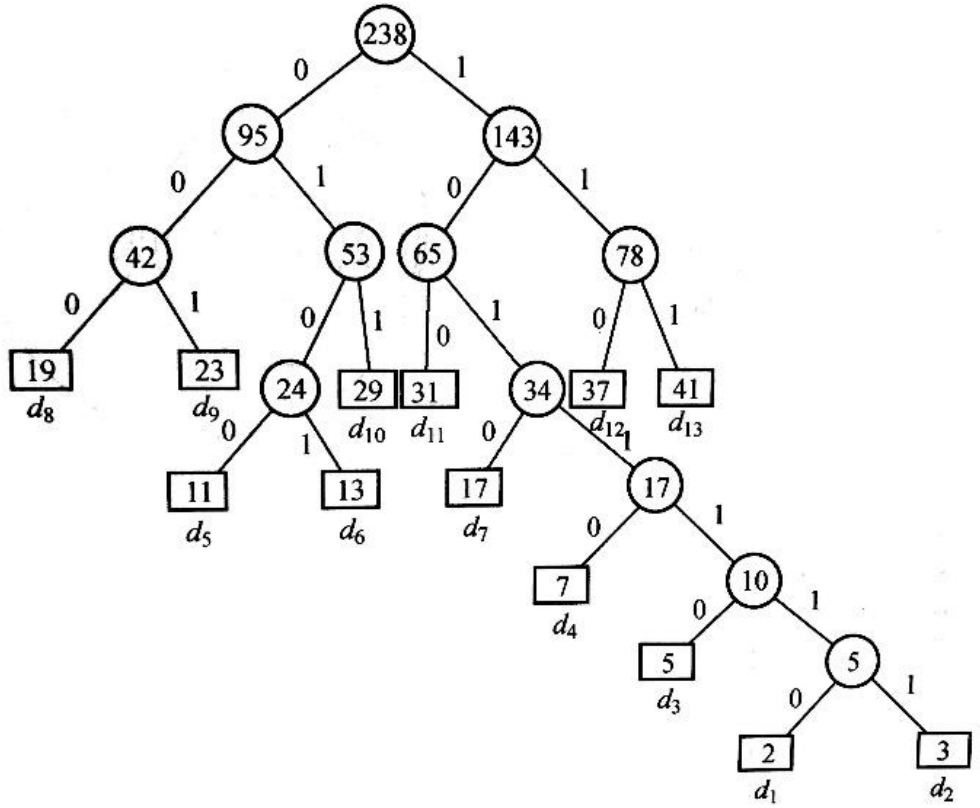
- 根据字符出现的概率，建立HuffMan树以后
  - 把从每个结点引向其左子女的边标上号码0
  - 把从每个结点引向右子女的边标上号码1
- 从根到每个叶子的路径上的号码连接起来就是这个叶子代表的字符的编码。



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 }



d_1：1011110， d_2：1011111， d_3：101110， d_4：10110

哈夫曼编码译码

- 由HuffMan编码得到的字符串的二进制序列，如何译码？
  - 从二叉树的根结点开始，对二进制序列的第一个字符开始匹配，如果为0，沿左分支走，如果为 1，沿右分支走，直到找到一个叶结点为止，则确定一条到达树叶的路径，译出一个字符。
  - 再回到树根，从二进制位串中的下一位开始继续译码

➢ 如：5个字符（A, B, C, D, E)的出现频率为｛20, 5, 30, 30, 15),可能得到的编码为：00, 010, 10, 11, 011

➢ 字符序列：　ABBDEEDCC

二进制序列：　00 010 010 11 011 011 11 10 10

译码：　　A B B D E E D C C

哈夫曼树的应用

- 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

课堂练习

- 假设用于通讯的电文由8个字符：a、b、c、d、e、f、g、i组成，它们的出现频率依次为0.07、0.19、0.02、0.06、0.32、0.03、0.21、0.10，
  - 为这8个字母设计哈夫曼编码，请画出哈夫曼树，并给出字符串“edfa”的编码。

思考

- 编制一个将百分制转换成五分制的程序，怎样才能使得程序中的比较次数最少？
- 成绩分布如下：

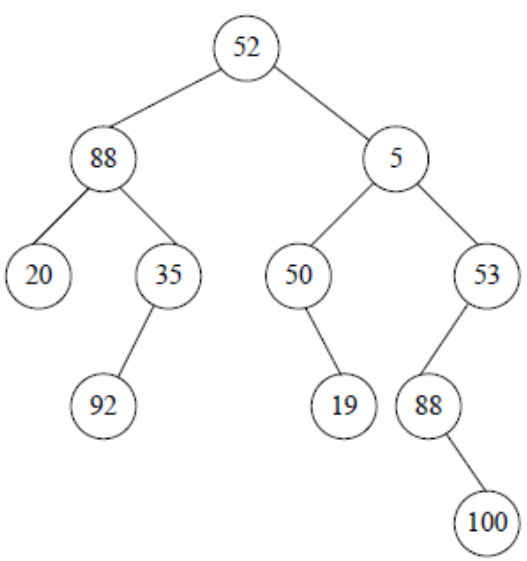
分数	0 - 59	60 - 69	70 - 79	80 - 89	90 - 100
比例数	0.05	0.15	0.40	0.30	0.10

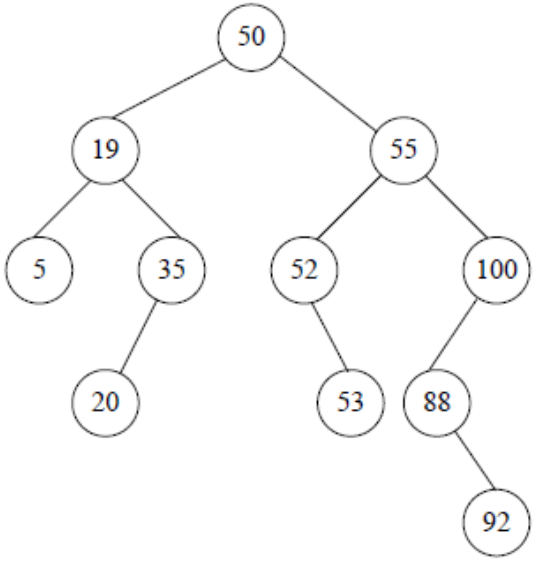
内容提要

- 二叉树基础
  - 树与二叉树的基本概念
  - 二叉树的存储结构
  - 二叉树的周游算法
  - 建立一个二叉树
- 二叉树的应用
  - 哈夫曼树
  - 二叉检索树/排序树
- 树与树林

二叉树

- 左右两棵二叉树的异同、特点？





二叉搜索树

- 二叉树的一个主要用途是提供对数据（包括索引）的快速检索，而一般的二叉树对此并不具有性能优势
- 常用名称（同义词）
  - 二叉搜索树（Binary Search Tree,简称BST）
  - 二叉查找树
  - 二叉检索树
  - 二叉排序树

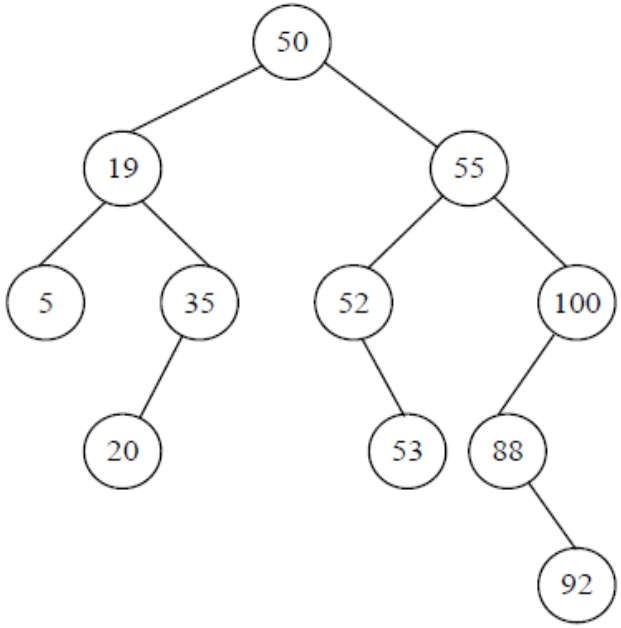
二叉搜索树：定义

假定存储在二叉树中的数据元素包含若干个域（field），其中一个称为码（key）K的域作为检索的依据，则二叉搜索树如下定义：

- 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（Binary Search Tree）
  - 树中的节点值唯一

二叉搜索树的性质

- 按照中序周游一棵二叉搜索树得到的序列将是按照码值由小到大的排列
- 树中的节点值唯一



二叉搜索树上的操作

- 检索
- 插入（生成）
- 删除

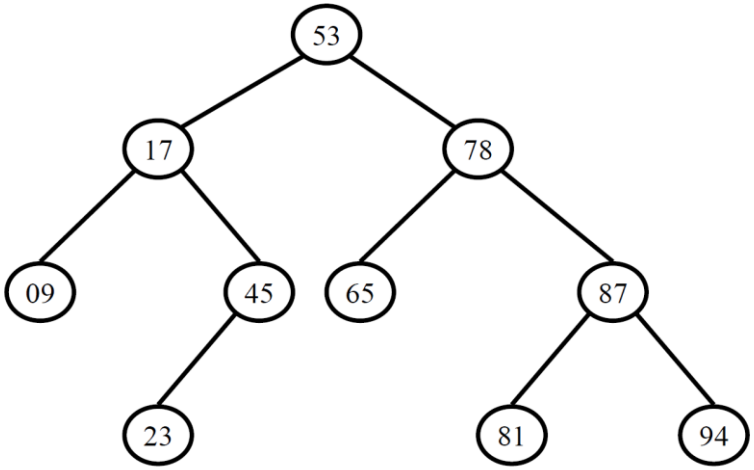
二叉搜索树上的检索

- 从根结点开始，在二叉搜索树中检索值**K**。
  - 如果根结点储存的值等于K，则检索结束
  - 如果K小于根结点的值，则只需检索左子树
  - 如果K大于根结点的值，就只需检索右子树

- 一直持续到**K**被找到或者遇上了一个树叶
  - 如果遇上树叶仍没有发现K，那么K就不在该二叉搜索树中

检索示例

- 查找码为53的结点
- 查找码为17的结点
- 查找码为87的结点
- 查找码为90的结点



比较次数？

时间复杂度？

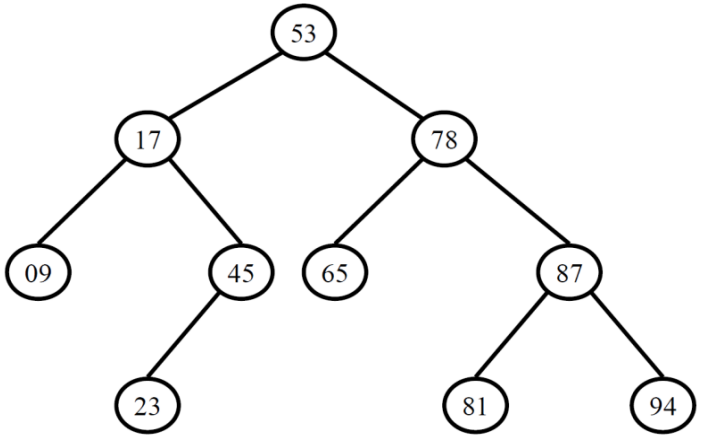
二叉搜索树的效率高就在于只需检索二个子树之一

二叉搜索树上的插入

- 设待插入的结点码为K:
  - 从根结点开始，若根结点为空，则将码K的结点作为根插入，操作结束
  - 如果K小于根结点的值，则将其按照二叉搜索树的要求插入左子树
  - 如果K大于根结点的值，就将其按照二叉搜索树的要求插入右子树
  - 如果相等，直接返回
- 保证结点插入后仍符合二叉搜索树的定义
- 成功的插入要先经历一次失败查找，再插入

插入示例

- 插入39
- 插入18



二叉搜索树的插入

- 代价分析
  - 在执行插入操作时，也不必像在有序线性表中插入元素那样要移动大量的数据，而只需改动某个结点的空指针插入一个叶结点即可
  - 与查找结点的操作一样，插入一个新结点操作的时间复杂度是根到插入位置的路径长度，因此在树形比较平衡时二叉搜索树的效率相当高

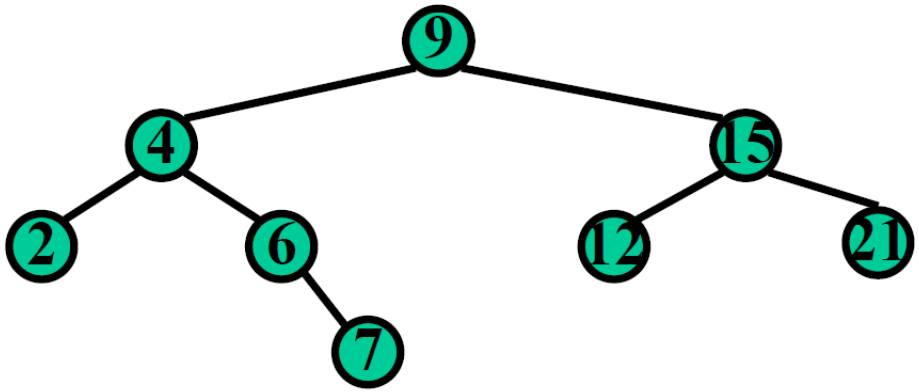
二叉搜索树的建立

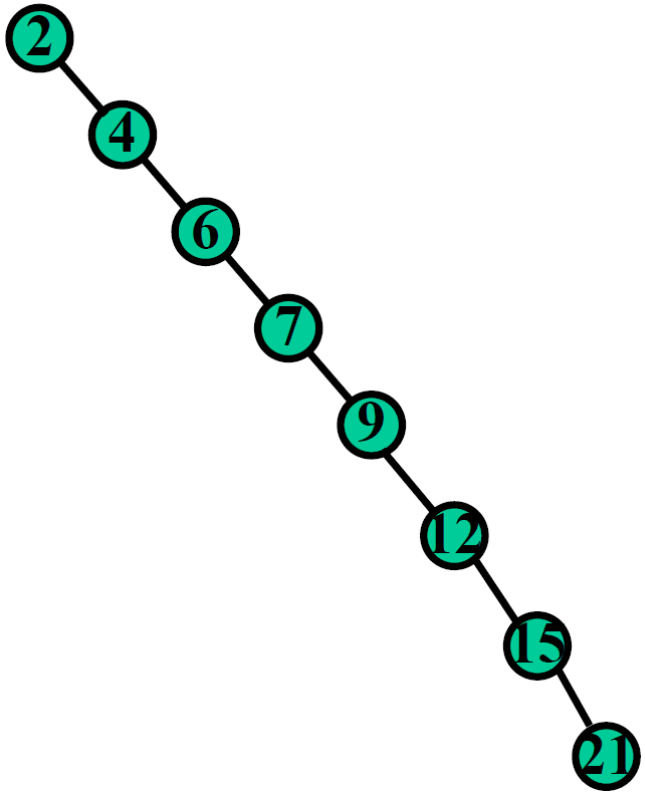
- 对于给定的关键码集合，为建立二叉搜索树，可以从一个空的二叉搜索树开始，将关键码一个个插进去

- 将关键码集合组织成二叉搜索树，实际上起了对集合里的关键码进行排序的作用，按中序周游二叉搜索树，就能得到排好的关键码序列

二叉树的平衡问题

- 输入：9,4,2,6,7,15,12,21
- 输入：2,4,6,7,9,12,15,21





二叉检索树的效率衡量

- 检索、插入、删除等操作的效率均依赖于二叉检索树的高度h，时间代价为O(h)
- 最佳：高度（尽可能）最小
- 最差：退化成线性结构

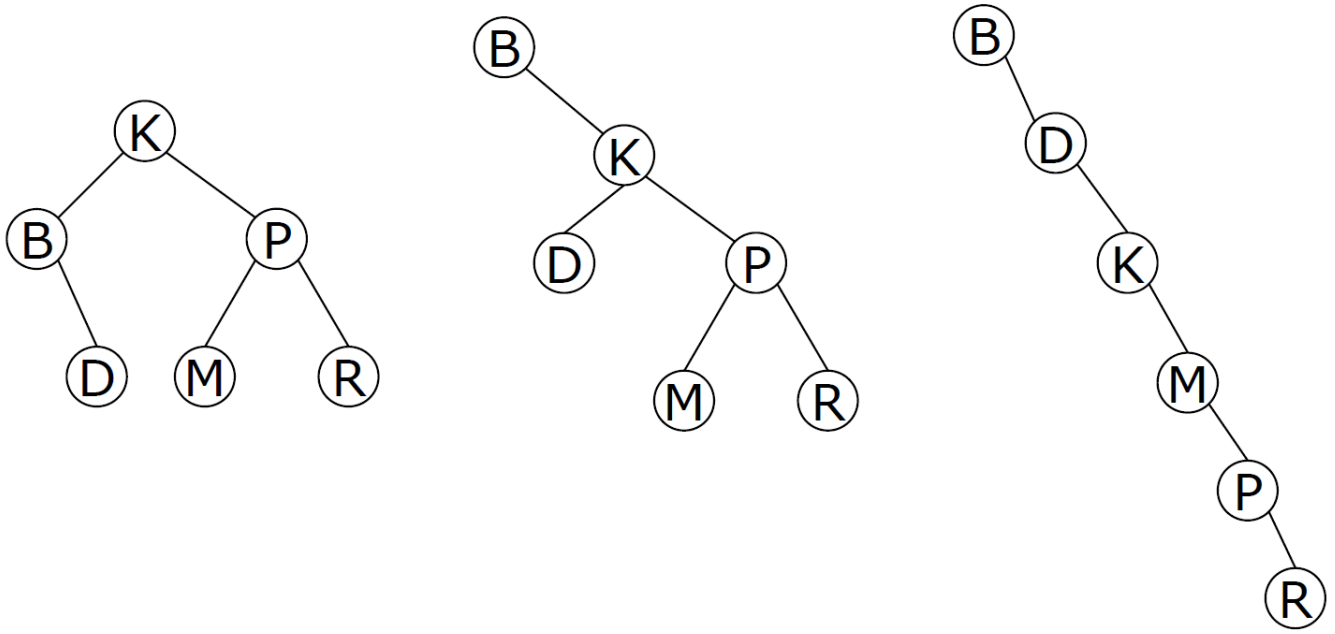
何为一棵最佳二叉检索树？

如何构筑一棵最佳二叉检索树？

如何保持二叉搜索树的最佳特性？

二叉搜索树

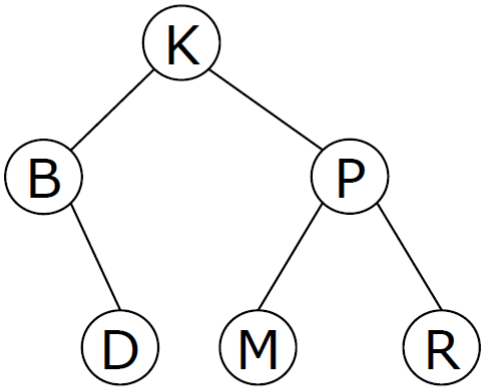
- 检索的效率树的形状、特点



最佳二叉搜索树

- 检索概率相同时，先对序列进行排序
- B D K M P R，然后用二分法依次插入这些关键码

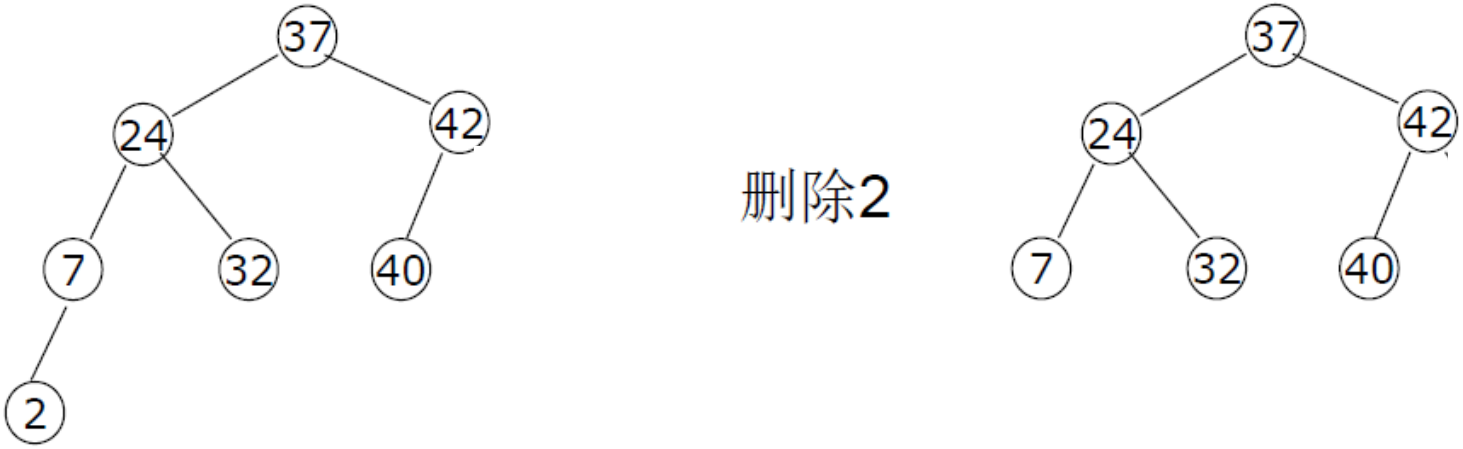
```
void balance(int data[], int first, int last)
{
   if (first <= last)
   {
          int middle =  (first + last) /2;
          insert(data[middle]);
          balance(data, first, middle-1);
          balance(data, middle+1, last);

   }
}
```



平衡二叉排序树：树中每个节点的左右子树高度之差绝对值不超过1

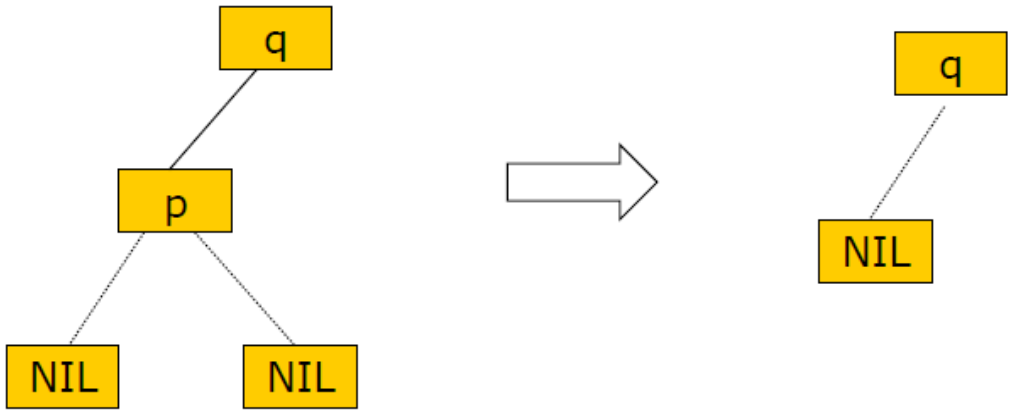
二叉搜索树上的删除

- 在二叉检索树删除一个结点，相当于删除有序序列中的一个记录，要求删除后仍能保持二叉检索树的排序特性，并且树高变化较小



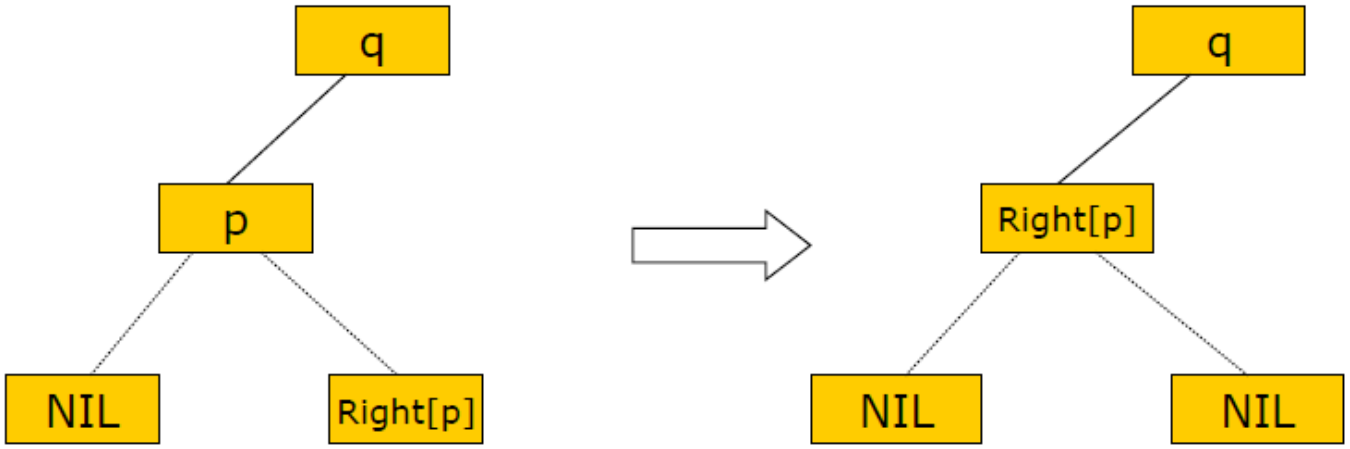
删除示例1

- 情况1. 叶结点可以直接删除，其父结点的相应指针置为空

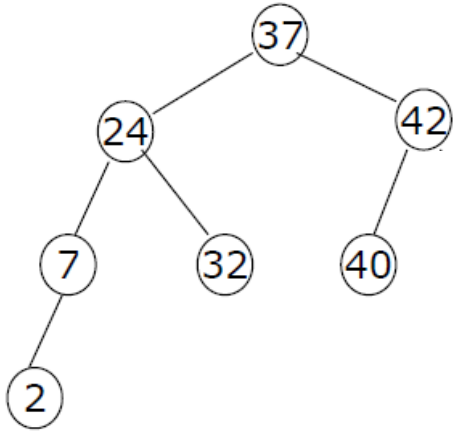


删除示例2

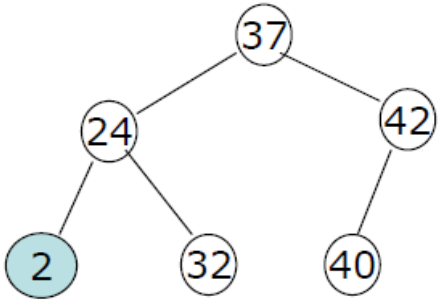
- 情况2. 只有一个子女的结点**p**被删除时，分以下情况：
  - p是q的左子结点，p只有左子结点或右子结点
  - p是q的右子结点，p只有左子结点或右子结点
- 可让此子女直接代替即可



删除示例2

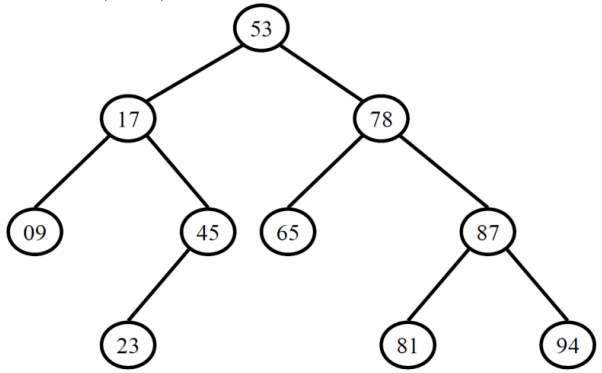


删除**7**

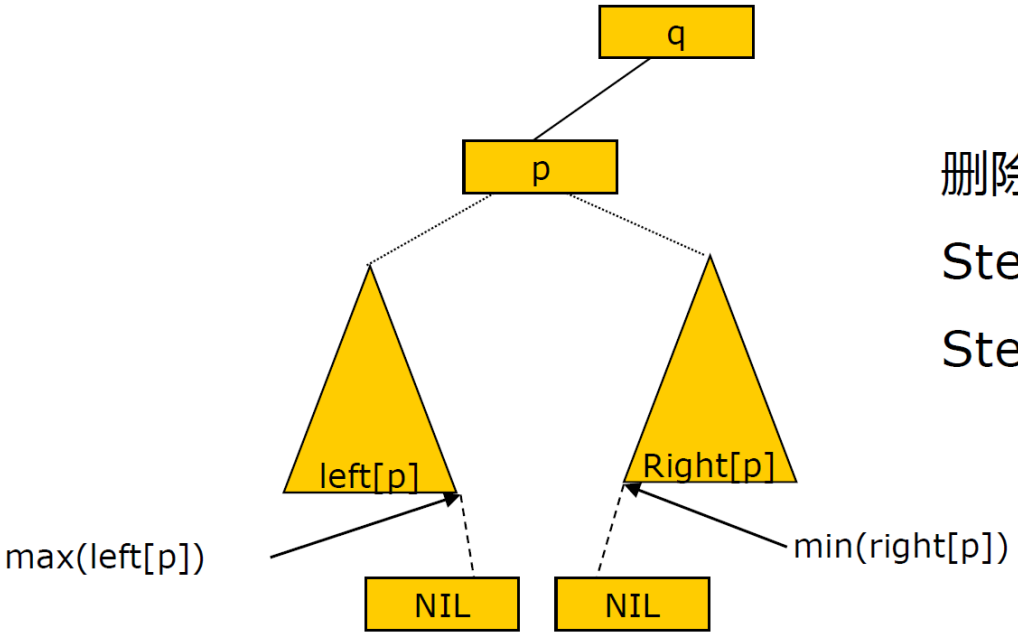


删除示例3

- 情况3. 被删除结点**p** 的左右子女皆不空。
  - 根据二叉检索树的性质，此时要寻找能代替此结点的结点必须是：比p 左子树中的所有结点都大，比p的右子树的所有结点都小（或不大于）。
- 两个选择：
  - 左子树中最大者
  - 右子树中最小者
- 而这二者都至多只有一个子结点



删除示例3

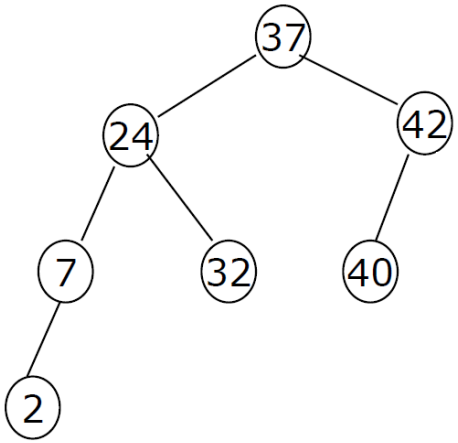


删除过程：

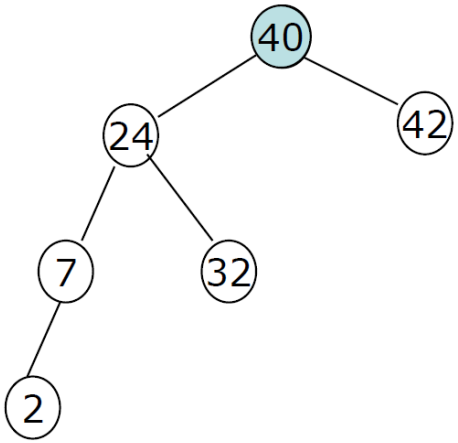
Step 1： 删除min(right[p])

Step 2: 存储min(right[p])到p中

删除示例3



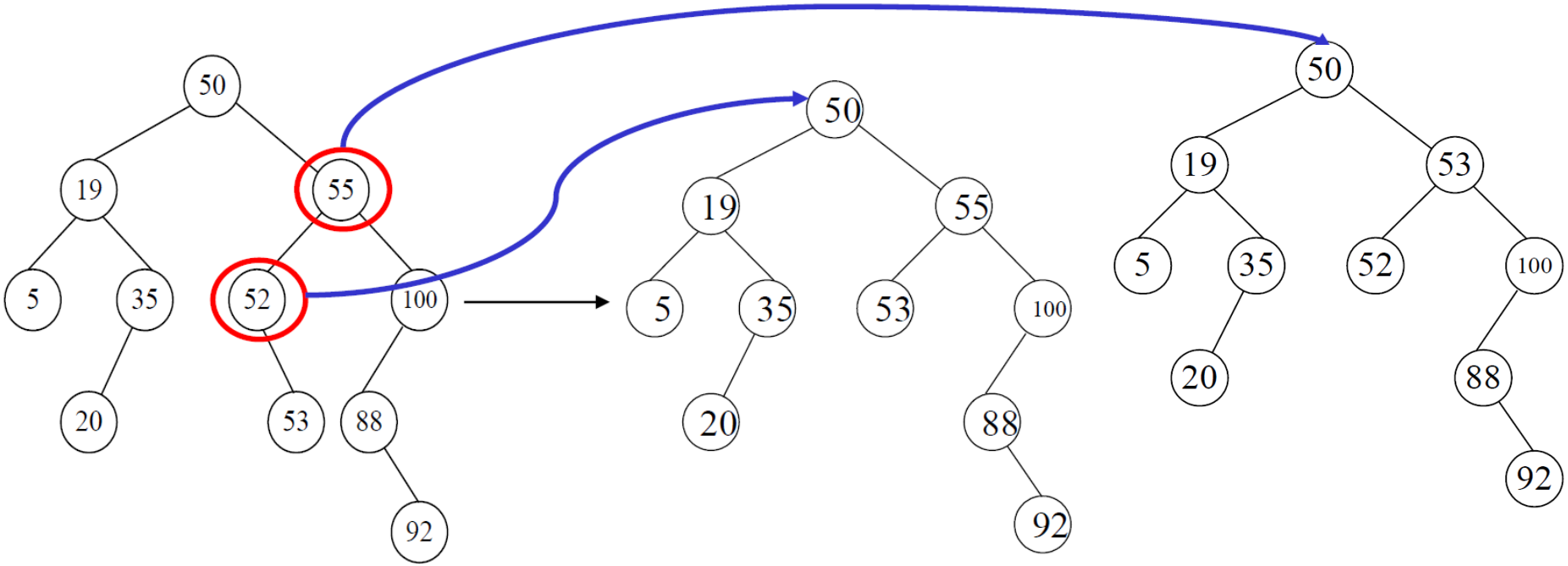
删除37



二叉搜索树的删除

- 综合3种情况，二叉搜索树结点删除算法的基本思想：
  - 若结点pointer没有左子树，则用pointer右子树的根代替被删除的结点pointer
  - 若结点pointer有左子树，则在左子树里找到按中序周游的最后一个结点temp-pointer（即左子树中的最大结点）并将其从二叉搜索树里删除
  - 由于temp-pointer没有右子树，删除该结点只需用temp-pointer的左子树代替temp-pointer，然后用temp-pointer结点代替待删除的结点pointer

删除示例


50
19
55
5
35
52
100
20
53
88
92
50
19
55
5
35
53
100
20
88
92
50
19
53
5
35
52
100
20
88
92

如何动态地保持最佳？

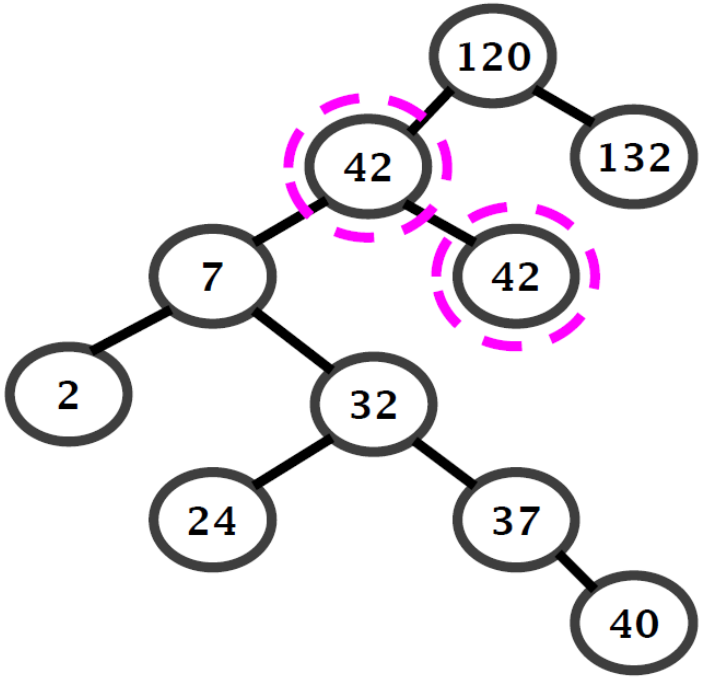
- 根据关键码集合及检索概率，可以构造出最佳二叉检索树
- 问题：静态，经过若干次插入、删除后可能会失去平衡，检索性能变坏

- 如何动态保持一棵二叉检索树的平衡，从而有较高的检索效率

平衡树技术

二叉搜索树

- 思考: 允许重复关键码吗?
  - 插入、检索、删除

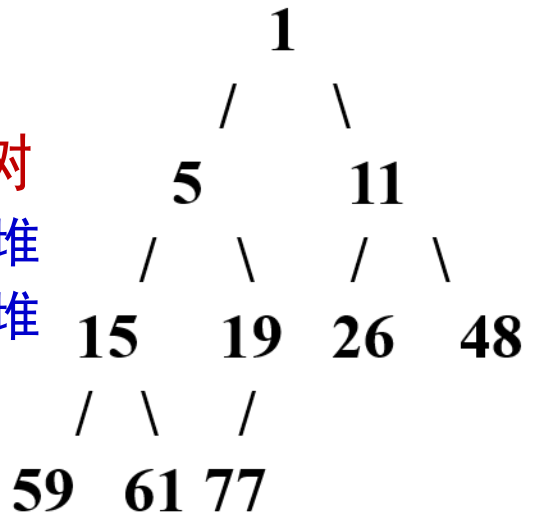


堆与优先队列

- 堆的定义：n个排序码序列$K=\{k_0,k_1,k_2,\ldots,k_{n-1}\}$当且仅当满足如下条件时，称之为堆。

$$k_i \le k_{2i+1} \quad k_i \ge k_{2i+1}$$
$$k_i \le k_{2i+2} \quad k_i \ge k_{2i+2} \quad (i=0, 1, 2, ..., n/2-1)$$

0	1	2	3	4	5	6	7	8	9
1	5	11	15	19	26	48	59	61	77

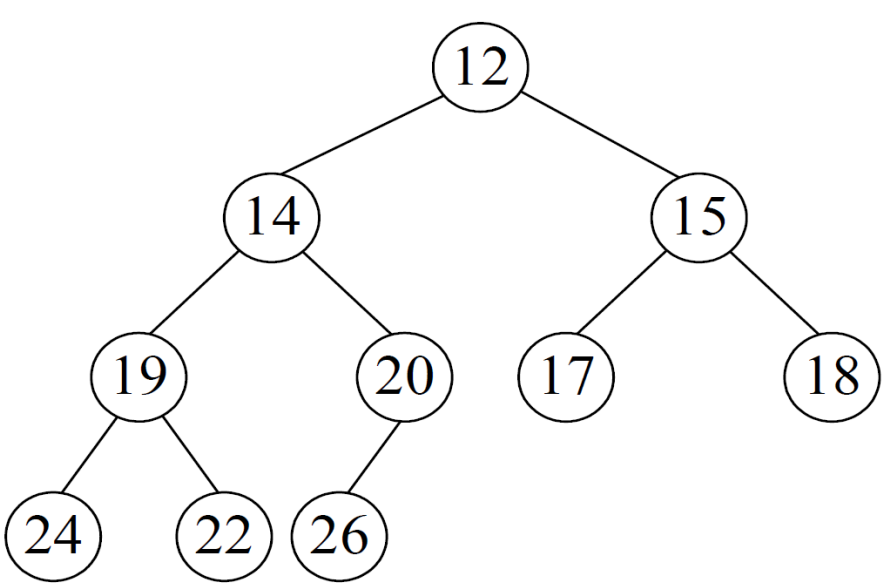


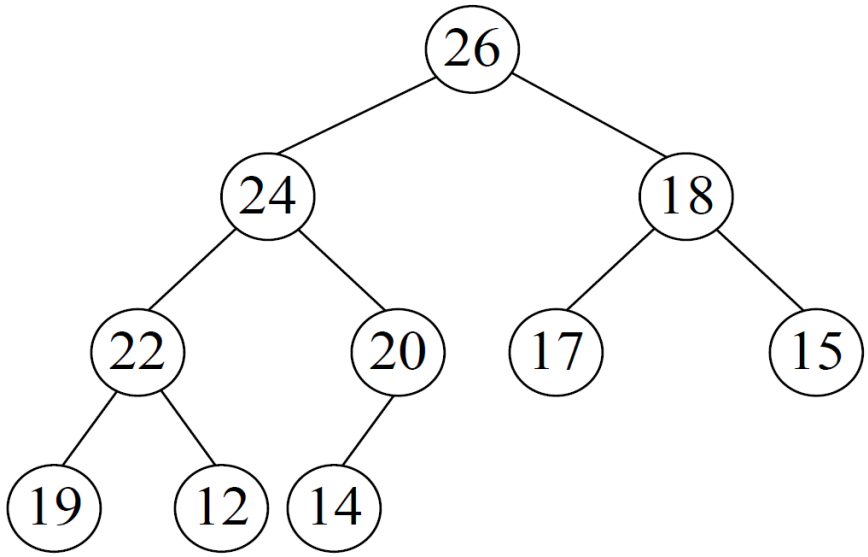
- 从定义看出：堆实质上是一棵完全二叉树
  - 如果堆中根结点的排序码最小，则称为小根堆
  - 如果堆中根结点的排序码最大，则称为大根堆

重要特性：堆中含堆，所以根节点一定是最大/最小值

小根堆与大根堆

- 关键码序列K = {12，14，15，19，20，17，18，24，22，26}所对应的小根堆与大根堆形成的完全二叉树形式为下图所示：





堆的性质

- 堆中数据**局部有序**（与**BST**树不同，全局有序）
  - 结点与其子女值之间存在大小比较关系
  - 两种堆（最大、最小）
  - 兄弟之间没有限定大小关系
- 堆不唯一
  - 从逻辑角度看，堆实际上是一种树型结构
- **堆是一个可用数组表示的完全二叉树**

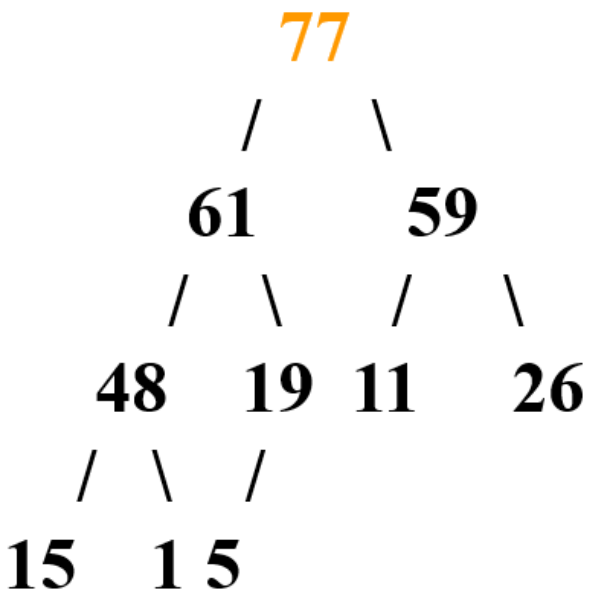
建堆的过程

- 一．将关键码放到一维数组形成完全二叉树，但并不具备最小堆的特性
  - 仅叶子结点代表的子树已经是堆
- 二．从完全二叉树的倒数第二层的i（n/2-1）位置开始，从右至左，从下至上依次调整（筛选算法）
- 三．直到树根，整棵完全二叉树就成为一个堆

建堆代码见排序算法部分

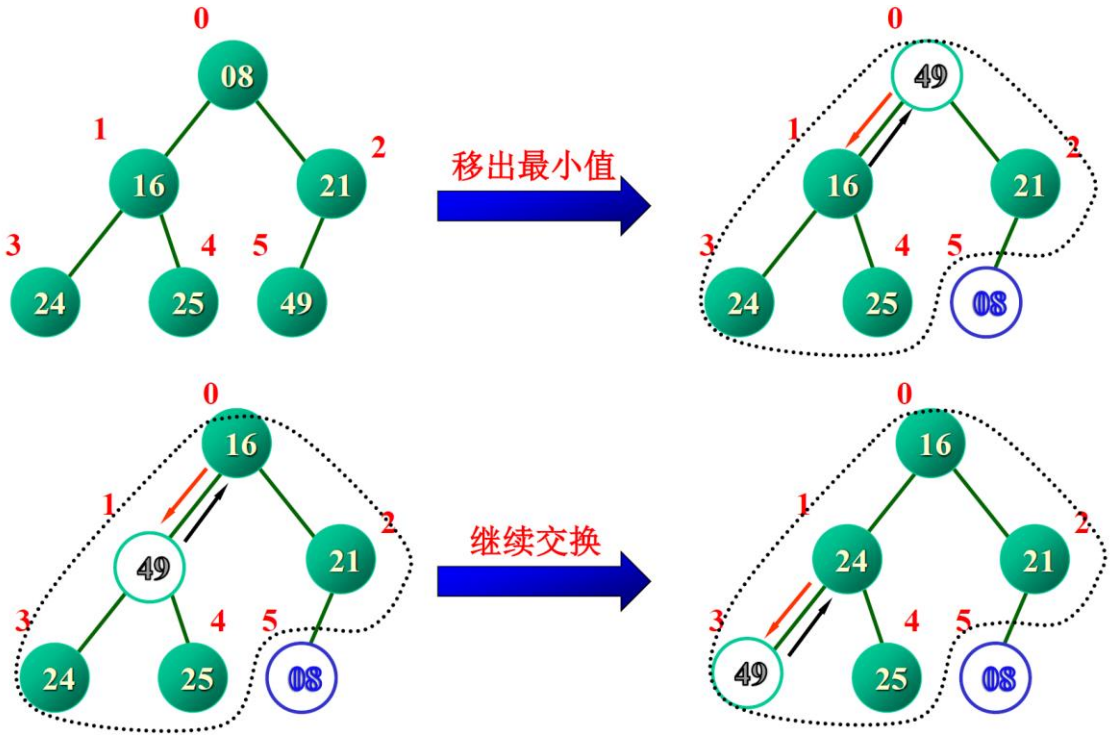
插入元素

- 先把新节点插入堆的最后位置，然后向上调整，使其成为堆



删除最大/最小值

- 移出最小值(根结点)后，剩下的n-1个结点仍要求符合堆性质
- 将堆中最后一个位置上的元素移到根节点，调整为堆



优先队列

- 优先队列(priority queue)是0个或多个元素的集合，每个元素有一个关键码值，执行查找、插入和删除操作。
- 优先队列的主要特点是从一个集合中快速地查找并移出具有最大值或最小值的元素。
  - 最小优先队列，适合查找和删除最小元素；
  - 最大优先队列中，适合查找和删除最大元素。
- 堆是优先队列的一种自然的实现方法