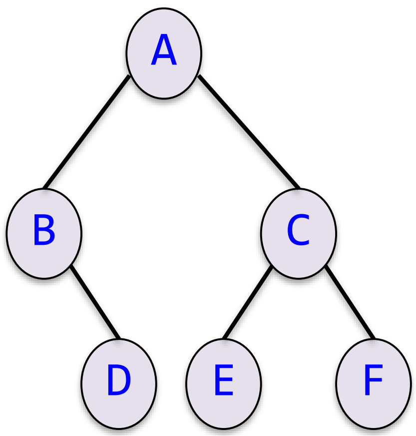
6.6 线索二叉树

6.6.1 线索二叉树的定义

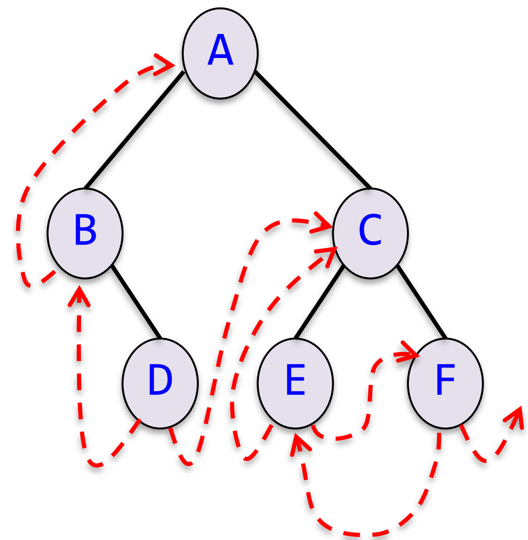
- 对于*n*个结点的二叉树，在二叉链存储结构中有***n*+1**个空链域。
- 利用这些空链域存放在某种遍历次序下该结点的前驱结点和后继结点的指针，这些指针称为线索，加上线索的二叉树称为线索二叉树。
- 线索二叉树分为先序、中序和后序线索二叉树。
- 对二叉树以某种方式遍历使其变为线索二叉树的过程称为线索化。

图中虚线为线索。



二叉树

先序线索二叉树



先序序列：**ABDCEF**

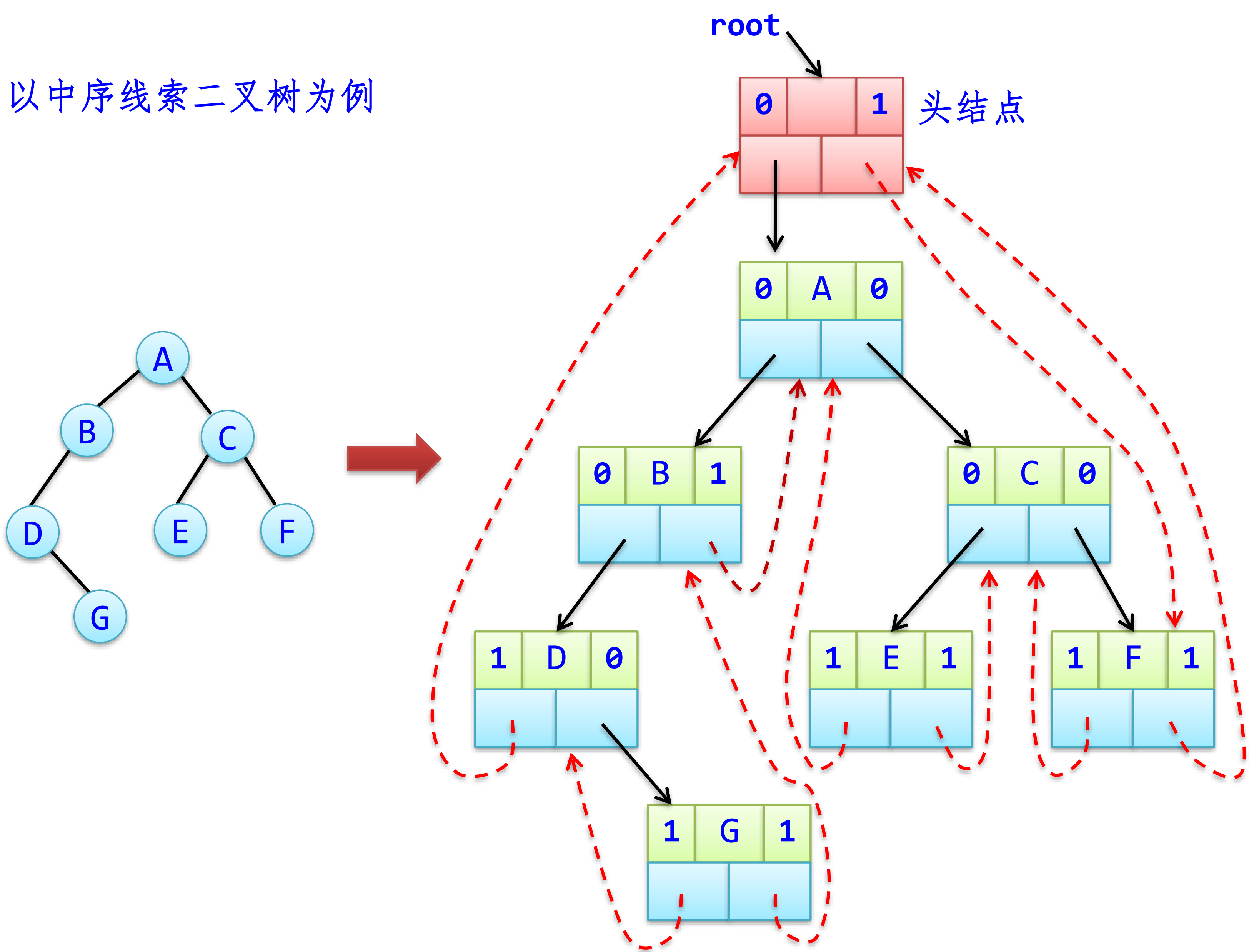
在原二叉链中增加了**ltag**和**rtag**两个标志域。

ltag= { **0** 表示**lchild**指向结点的左孩子
　　　　 1 表示**lchild**指向结点的前驱结点即为线索

rtag= { **0** 表示**rchild**指向结点的右孩子
　　　　 1 表示**rchild**指向结点的后继结点即为线索

ltag	lchild	data	rchild	rtag

以中序线索二叉树为例



6.6.2 线索化二叉树

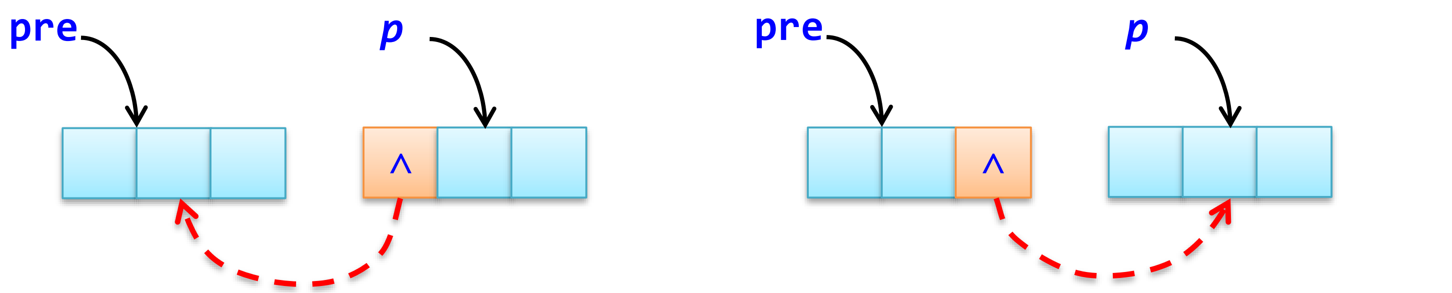
```
class ThNode:                          #线索二叉树结点类型
    def __init__(self,d=None):         #构造方法
        self.data=d                    #结点值
        self.ltag=0                    #左标志
        self.rtag=0                    #右标志
        self.lchild=None               #左指针
        self.rchild=None               #右指针
```

中序线索化二叉树类ThreadTree

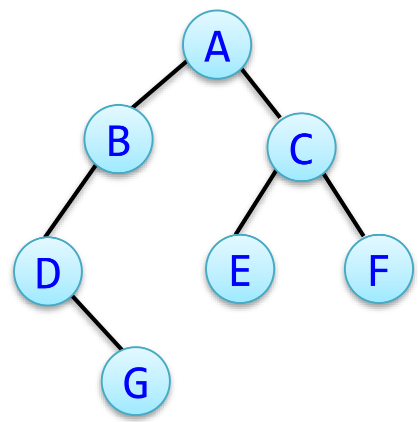
```
class ThreadTree:                   #中序线索化二叉树类
    def __init__(self,d=None):      #构造方法
        self.b=None                 #根结点指针
        self.root=None              #线索二叉树的头结点
        self.pre=None               #用于中序线索化,指向中序前驱结点
    #二叉树的基本操作(结点类型改为ThNode)
    #def SetRoot(self,r):设置根结点为r
    #def DispBTree(self):返回二叉链的括号表示串
    #中序线索二叉树的基本操作
    def CreateThread(self):         #建立以root为头结点的中序线索二叉树
    def ThInOrder(self):            #中序线索二叉树的中序遍历
```

```
def CreateThread(self):                #建立以root为头结点的中序线索二叉树
   self.root=ThNode()                  #创建头结点root
   self.root.ltag=0                    #头结点标志置初值
   self.root.rtag=1
   if self.b==None:                    #b为空树时
      self.root.lchild=self.root
      self.root.rchild=None
   else:                               #b不为空树时
      self.root.lchild=self.b
      self.pre=self.root               #pre是p的前驱结点，用于线索化
      self._Thread(self.b)             #中序遍历线索化二叉树
      self.pre.rchild=self.root        #最后处理，加入指向根结点的线索
      self.pre.rtag=1
      self.root.rchild=self.pre        #根结点右线索化
```

- 采用中序遍历进行中序线索化
- 在整个算法中*p*总是指向当前访问的结点，pre指向其前驱结点。



(a) 将结点*p*的左空指针改为线索　　(b) 将结点pre的右空指针改为线索



中序序列： D G B A E C F

```python
def _Thread(self,p):                     #对以p为根结点的二叉树进行中序线索化
    if p!=None:
        self._Thread(p.lchild)           #左子树线索化
        if p.lchild==None:               #结点p的左指针为空
            p.lchild=self.pre            #给结点p添加前驱线索
            p.ltag=1
        else: p.ltag=0
        if self.pre.rchild==None:        #结点pre的右指针为空
            self.pre.rchild=p            #给结点pre添加后继线索
            self.pre.rtag=1
        else: self.pre.rtag=0;
        self.pre=p                       #置p结点为下一次访问结点的前驱结点
        self._Thread(p.rchild)           #右子树线索化
```

建立pre和p之间的线索：相当于中序遍历中访问结点

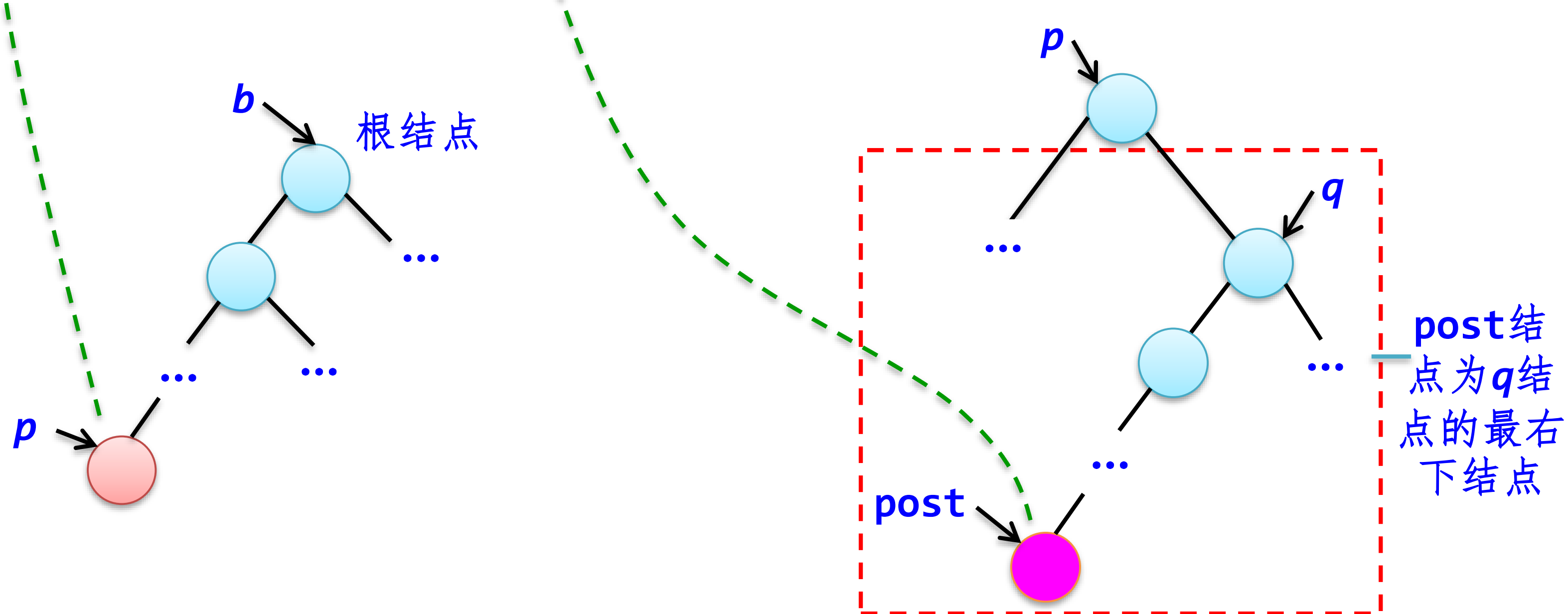
在该线索二叉树中实现中序遍历的两个步骤是：

（1）求中序序列的开始结点：实际上该结点就是根结点的最左下结点。

（2）对于一个结点*p*，求其后继结点的过程是：

① 如果*p*结点的**rchild**指针为线索，则**rchild**所指为其后继结点。

② 否则*p*结点的后继结点是其右孩子*q*的最左下结点**post**。



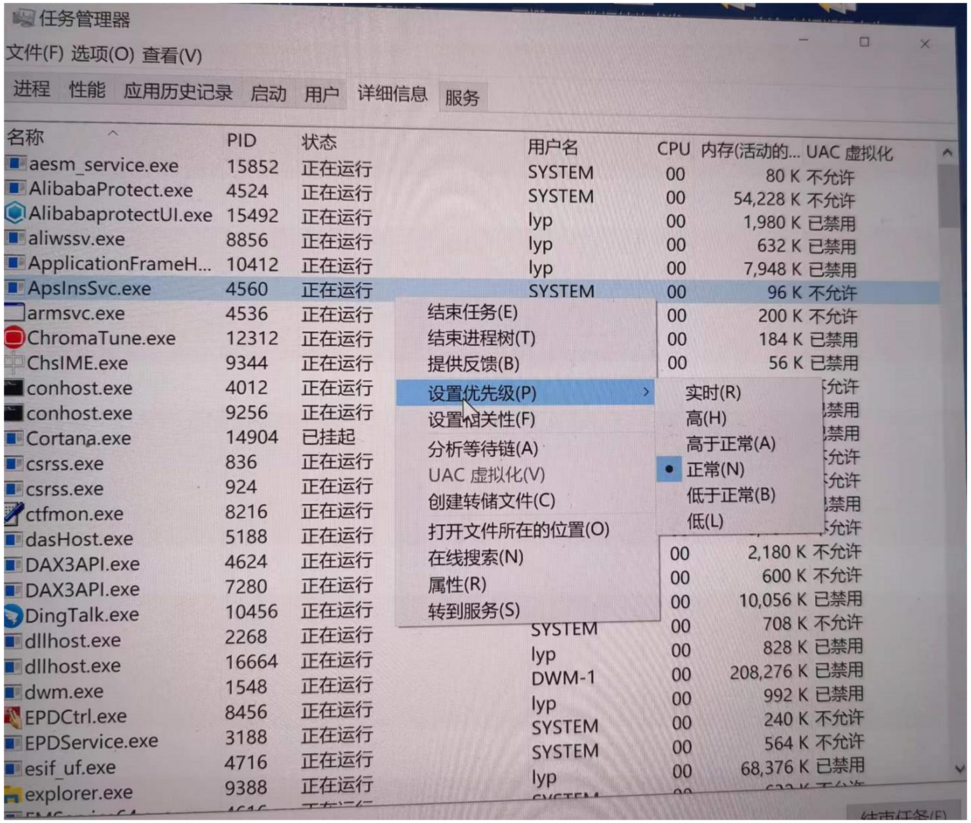
```
def ThInOrder(self):                            #中序线索二叉树的中序遍历
  p=self.root.lchild                            #p指向根结点
  while p!=self.root:
     while p!=self.root and p.ltag==0:          #找中序开始结点
        p=p.lchild
     print(p.data,end=' ')                      #访问p结点
     while p.rtag==1 and p.rchild!=self.root:
        p=p.rchild                              #如果是线索，一直找下去
        print(p.data,end=' ')                   #访问p结点
     p=p.rchild                                 #如果不再是线索，转向其右子树
```

该算法是一个非递归算法，算法的时间复杂度为O(n)，空间复杂度为O(1)

*优先队列

- 普通队列中按进队的先后顺序出队；
- 优先队列，也称为堆。按优先级越大越优先出队；
- 按照根的大小分为大根堆和小根堆，**大根堆**的元素越大越优先出队（即元素越大优先级也越大），反之，**小根堆**的元素越小越优先出队。

- 银行的服务要取号排队，VIP客户可以插到队首
- 操作系统中执行关键任务的或用户特别指定高优先级的进程，优先调度
- 任务管理器中，进程



其操作：插入；删除最大(小)值。

若采用数组或链表实现优先队列

数组：

插入---元素总是插入在尾部 $-\Theta(1)$

删除---查找最大(或最小)结点的值 $-\Theta(n)$ $+O(n)$

链表：

插入---总是插入链表的头部 $-\Theta(1)$

删除---查找最大(或最小)结点，删除 $-\Theta(n)$ $+\Theta(1)$

有序数组：

插入---找到合适的位置 $-O(\log_2 n)$

移动元素并插入 $-O(n)$

删除---删除最后一个元素 $-\Theta(1)$

有序链表:

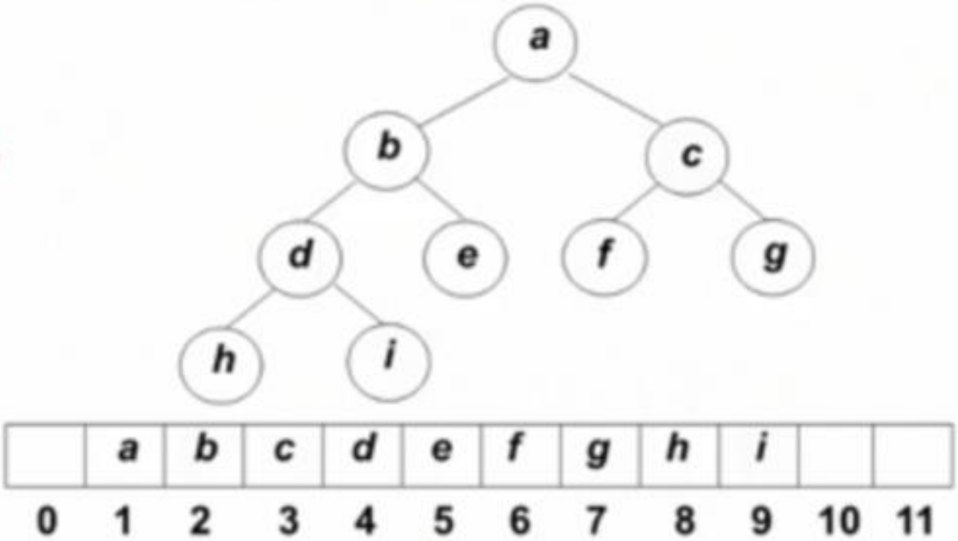
插入---找到合适的位置 $-O(n)$

插入元素 $-\Theta(1)$

删除---删除首元素或最后一个元素 $-\Theta(1)$

◆ 可否用树存储？树结构如何？

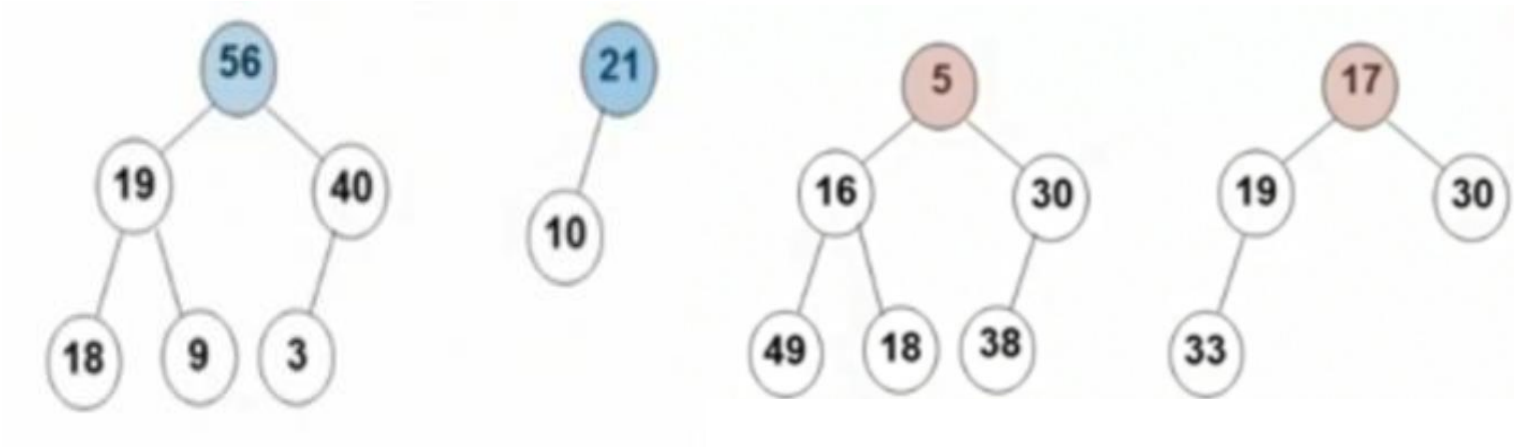
优先队列的完全二叉树表示



堆有两个特性

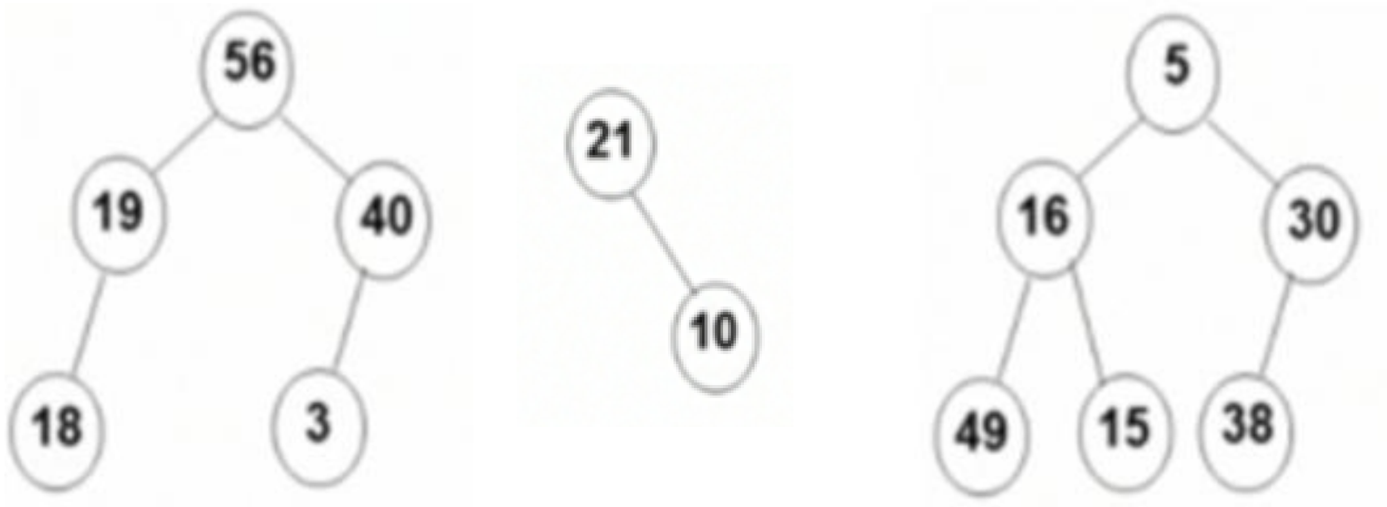
- 实现优先队列的经典方案是采用二叉堆数据结构
- 二叉堆能够将优先队列的入队和出队复杂度都保持在O(log n)
- 二叉堆的巧妙之处在于，其逻辑结构像一棵二叉树，却可以采用非嵌套的列表来实现。

例.



堆次序：从根结点到任意结点的路径上结点序列有序！

非二叉堆



下面以最小堆为例，讲解二叉堆的插入和删除最小项的操作和实现

ADT BinaryHeap 的操作定义如下:

BinaryHeap(): 创建一个空二叉堆对象;

insert(k): 将新元素加入到堆中;

findMin(): 返回堆中的最小项，最小项仍保留在堆中;

delMin(): 返回堆中的最小项，同时从堆中删除;

isEmpty(): 返回堆是否为空;

size(): 返回堆中元素的个数;

buildHeap(list): 基于一个列表创建新堆

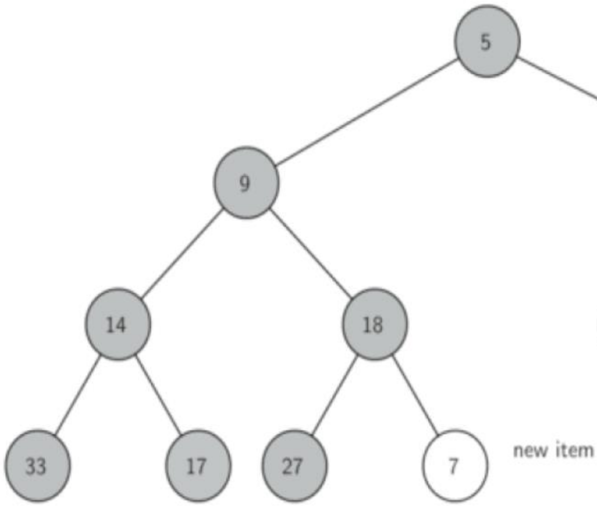
二叉堆的实现

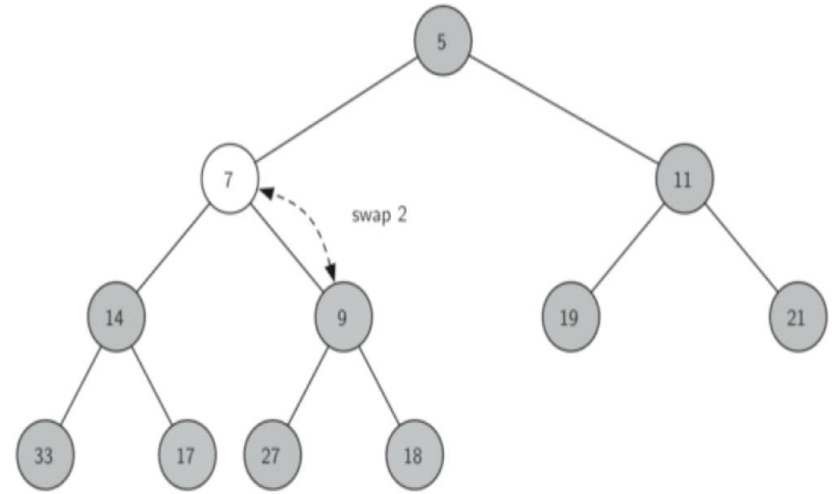
二叉堆的初始化

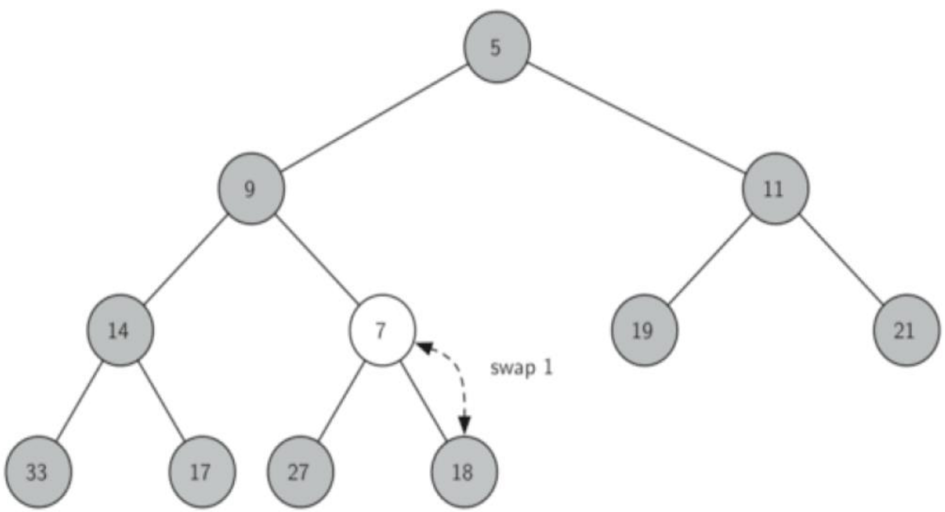
利用一个列表来保存堆数据，其中表首下标为0的项无用，但为了后面代码用到简单的整数乘除法，仍保留它。

```
class BinHeap:
    def __init__(self):
        self.heapList=[0]
        self.currentSize=0
```

插入新元素


5
9
11
14
18
19
21
33
17
27
7
new item


5
7
11
swap 2
14
9
19
21
33
17
27
18

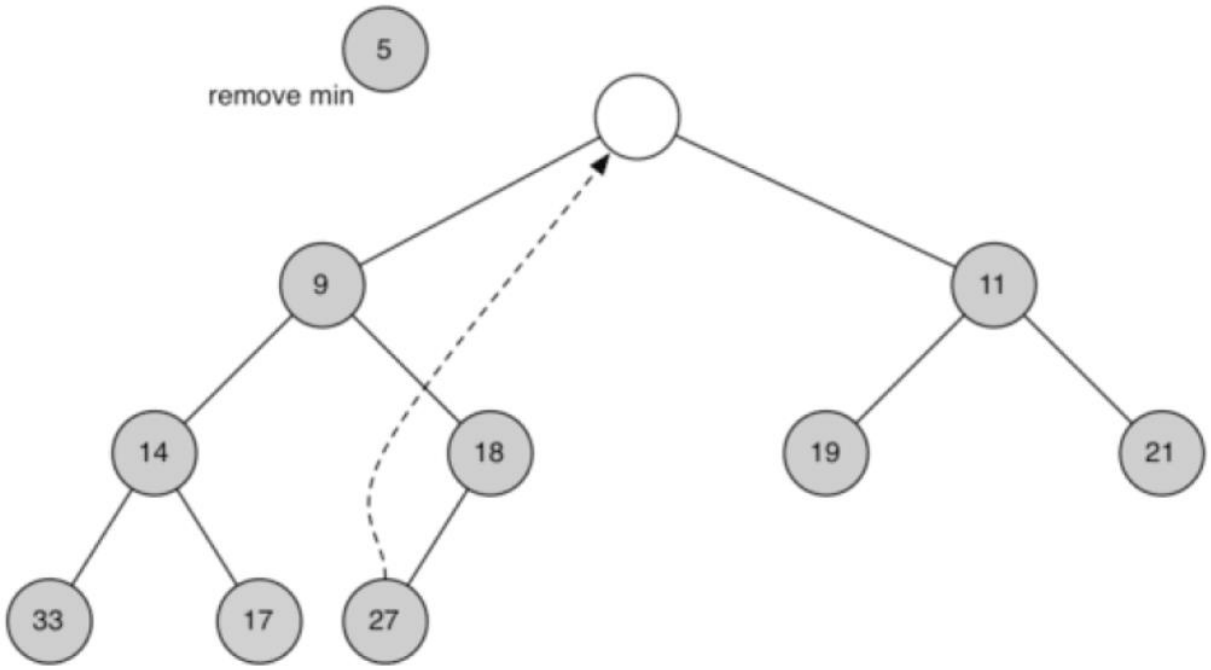

5
9
11
14
7
19
21
swap 1
33
17
27
18

```
def insert(self, k):
  self.heapList.append(k)     // 添加到末尾
  self.currentSize = self.currentSize + 1
  self.percUp(self.currentSize)   //新的元素上浮

def percUp(self, i):
  while i // 2 > 0:
   if self.heapList[i] < self.heapList[i // 2]:
     tmp = self.heapList[i // 2]                  //与父节点交换
     self.heapList[i // 2] = self.heapList[i]
     self.heapList[i] = tmp
     i = i // 2                                   // 沿路径向上
```

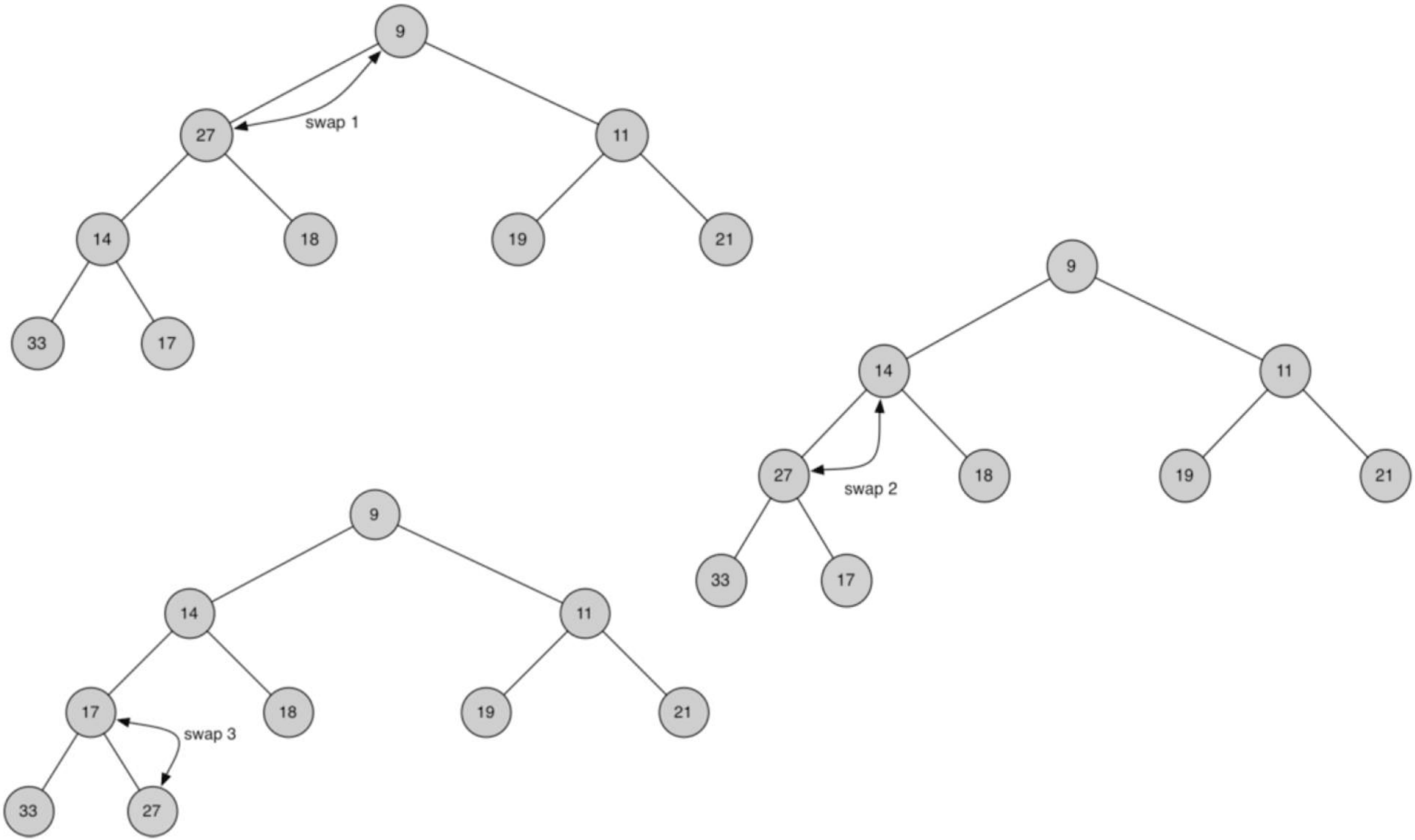
DelMin() 方法

移走整个堆中最小的元素，即根节点heapList[1]

为了保持完全二叉堆的性质，先用最后一个节点来代替根节点。



下沉的路径选择：如果比子节点大，则选择较小的子节点交换下沉



```
def delMin(self):
  retval = self.heapList[1]     //移走堆顶
  self.heapList[1] = self.heapList[self.currentSize]
  self.currentSize = self.currentSize - 1
  self.heapList.pop()
  self.percDown(1)      //新顶下沉
  return retval
```

```
def minChild(self, i):
  if 2*i+1 > self.currentSize:
     return 2*i    // 唯一子节点
  else:
     if self.heapList[2*i] < self.heapList[2*i+1]:
        return 2*i         //返回最小孩子
     else:
        return  2*i+1
```

```
def percDown(self, i):
  while (i*2) <= self.currentSize:
    mc = self.minChild(i)
    if self.heapList[i] > self.heapList[mc]:
      tmp = self.heapList[i]                 //交换下沉
      self.heapList[i] = self.heapList[mc]
      self.heapList[mc] = tmp
    i = mc    // 沿路径向下
```

最小堆的建立

- 堆的一个典型应用：堆排序

  ---需要先建堆

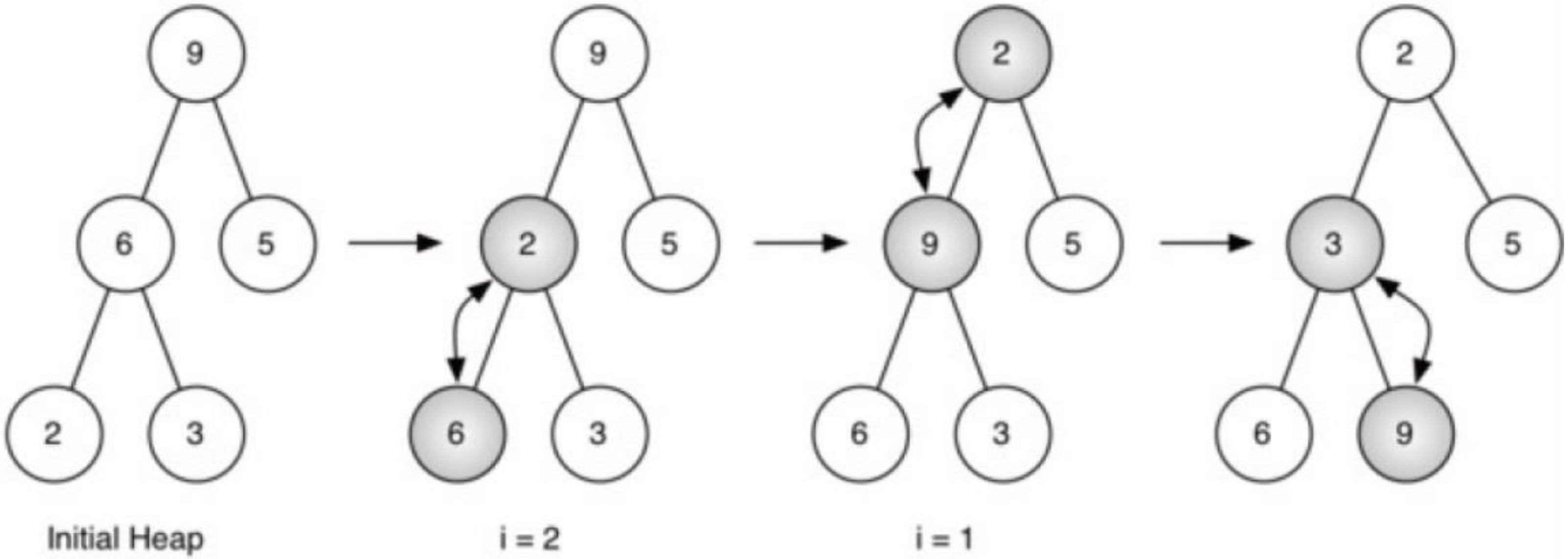
方法1. 将n个元素依次插入到一个初始为空的堆中，

其时间复杂度为O(nlogn)。

方法2. 在线性时间内建立最小堆。

(1) 按输入顺序建立一棵完全二叉树;

(2) 调整各结点位置，以满足最小堆的有序特性。


9
6
5
2
3
Initial Heap
9
2
5
6
3
i = 2
2
9
5
6
3
i = 1
2
3
5
6
9

Python中提供了heapq模块，其中包含堆的基本操作方法用于创建堆，但只能创建小根堆。其主要方法如下：

- heapq.heapify(list)：把列表list调整为堆。
- heapq.heappush(heap, item)：向堆heap中插入元素item（进队item元素），该方法会维护堆的性质。
- heapq.heappop(heap)：从堆heap中删除最小元素并且返回该元素值。
- heapq.heapreplace(heap, item)：从堆heap中删除最小元素并且返回该元素值，同时将item插入并且维护堆的性质。它优于调用函数heappop(heap)和heappush(heap, item)。
- heapq.heappushpop(heap, item)：把元素item插入到堆heap中，然后从heap中删除最小元素并且返回该元素值。它优于调用函数heappush(heap, item)和heappop(heap)。
- heapq.nlargest(n, iterable[, key])：返回迭代数据集合iterable中第*n*大的元素，可以指定比较的key。它比通常计算多个list第*n*大的元素方法更方便快捷。
- heapq.nsmallest(n, iterable[, key])：返回迭代数据集合iterable中第*n*小的元素，可以指定比较的key。它比通常计算多个list第*n*小的元素方法更方便快捷。
- heapq.merge(*iterables)：把多个堆合并，并返回一个迭代器。

例如，定义一个heapq列表，将其调整为小根堆，调用一系列heapq方法及其输出结果如下：

```
import heapq
heap=[6,5,4,1,8]                          #定义一个列表heap
heapq.heapify(heap)                       #将heap列表调整为堆
print(heap)                               #输出:[1,5,4,6,8]
heapq.heappush(heap,3)                    #进队3
print(heap)                               #输出:[1,5,3,6,8,4]
print(heapq.heappop(heap))                #输出:1
print(heap)                               #输出:[3,5,4,6,8]
print(heapq.heapreplace(heap,2))          #输出:3(出队最小元素，再插入2)
print(heap)                               #输出:[2,5,4,6,8]
print(heapq.heappushpop(heap,1))          #输出:1(插入1，再出队最小元素)
print(heap)                               #输出:[2,5,4,6,8]
```

- 由于**heapq**不支持大根堆，那么如何创建大根堆呢？
- 对于数值类型，一个最大数的相反数就是最小数，可以通过对数值取反、仍然创建小根堆的方式来获取最大数。

6.7 哈夫曼树

ASCII码/定长码

ab12: 01100001 01100010 00110001 00110010

97 98 49 50

哈夫曼码/不定长码

按字符的使用频率，使文本代码的总长度具有最小值。

哈夫曼编码和译码都基于一种二叉树，即为哈夫曼树。

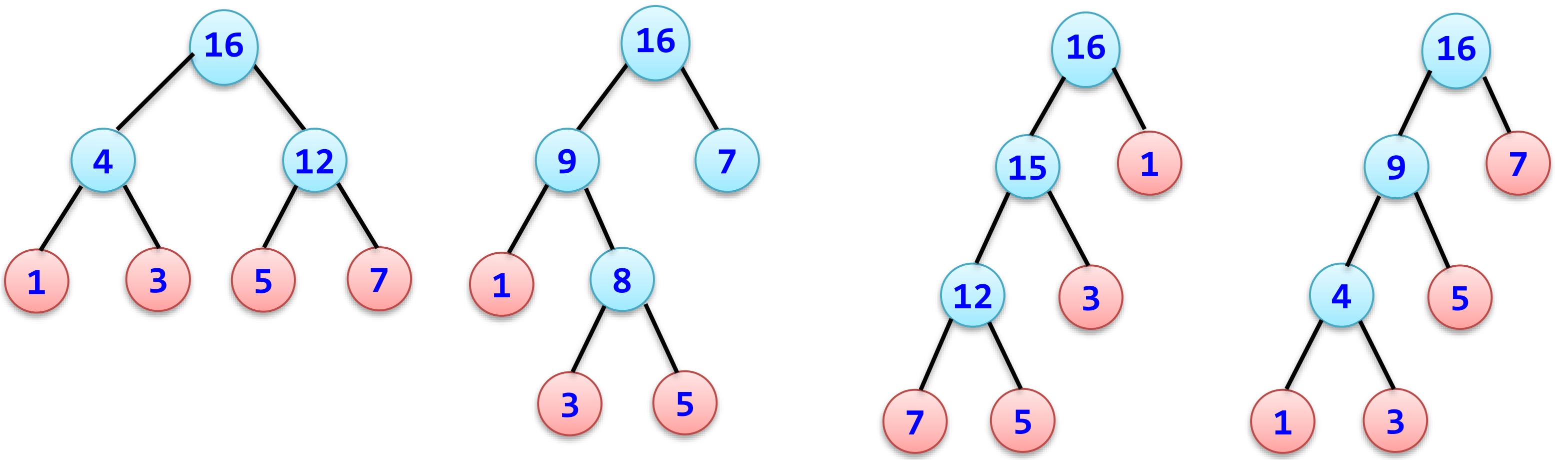
6.7.1 哈夫曼树的定义

- 应用中常给树中的结点赋上一个表示某种意义的数值–权。
- 从树根结点到某个结点之间的路径长度与该结点权的乘积称为结点的带权路径长度。
- 一棵二叉树，树中所有叶子结点的带权路径长度之和称为该树的带权路径长度，通常记为：

$$\mathrm{WPL} = \sum_{i=1}^{n_0} w_i \times l_i$$

- 在n_0个带权叶子结点构成的所有二叉树中，带权路径长度WPL最小的二叉树称为哈夫曼树（或最优二叉树）。

给定4个叶子结点，设其权值分别为1、3、5、7，可以构造出形状不同的4棵二叉树。



（a） （b） （c） （d）

（a） $WPL=1\times 2+3\times 2+5\times 2+7\times 2=32$

（b） $WPL=1\times 2+3\times 3+5\times 3+7\times 1=33$

（c） $WPL=7\times 3+5\times 3+3\times 2+1\times 1=43$

（d） $WPL=1\times 3+3\times 3+5\times 2+7\times 1=29$ √

6.7.2 哈夫曼树的构造算法

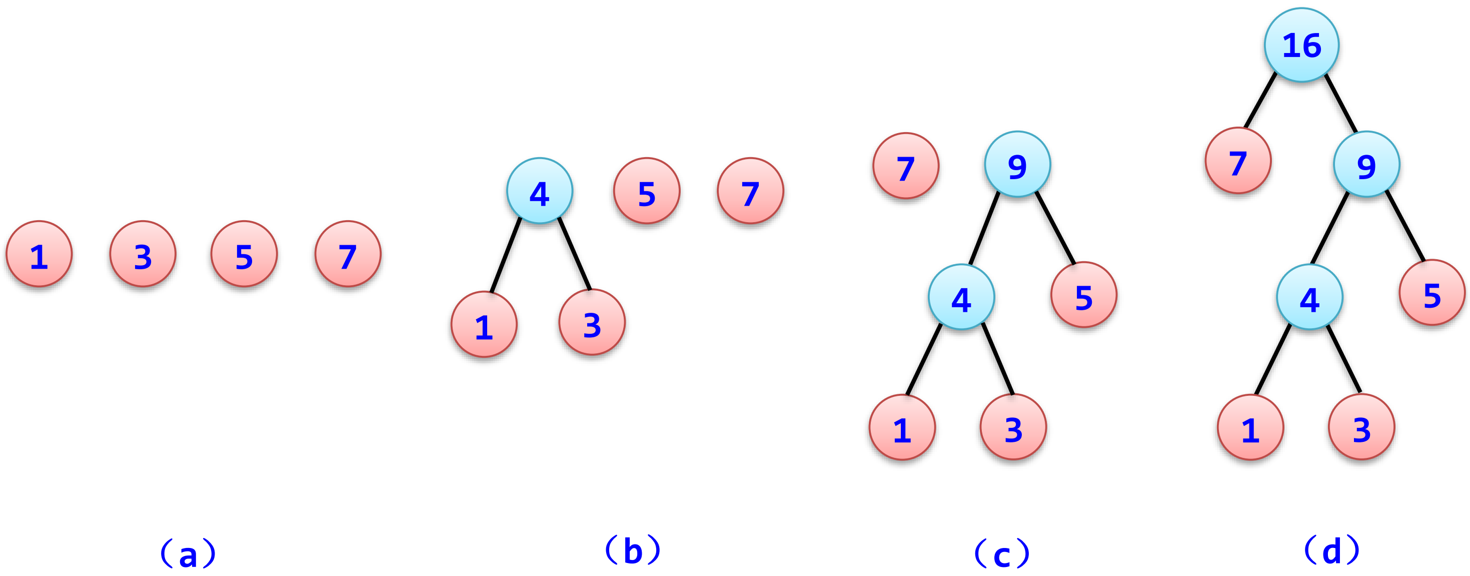
（1）根据给定的n_0个权值W=(w_1，w_2，…，w_{n0})，对应结点构成n_0棵二叉树的森林T=(T_1，T_2，…，T_{n0})，其中每棵二叉树T_i（$1\leqslant i\leqslant n_0$）中都只有一个带权值为$w_i$的根结点，其左、右子树均为空。

（2）在森林T中选取两棵根结点权值最小的子树作为左、右子树构造一棵新的二叉树，且置新的二叉树的根结点的权值为其左、右子树的根的权值之和。称为合并，每合并一次T中减少一棵二叉树。

（3）重复（2）直到T只含一棵树为止。这棵树便是哈夫曼树。

每次都是两棵子树合并 ⇨ 哈夫曼树中没有单分支结点

W=(1，3，5，7)来构造一棵哈夫曼树



定理6.3 对于具有n_0个叶子结点的哈夫曼树，共有$2n_0-1$个结点。

证明：从哈夫曼树的构造过程看出，每次合并都是将两棵二叉树合并为一个，所以哈夫曼树不存在度为1的结点，即$n_1=0$。

由二叉树的性质1可知$n_0=n_2+1$，即$n_2=n_0-1$

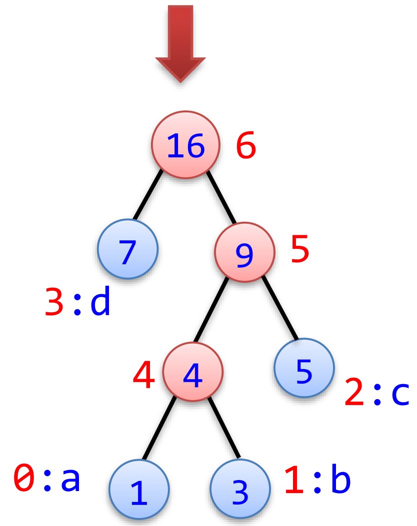
则结点总数$n=n_0+n_1+n_2= n_0+n_2=n_0+n_0-1=2n_0-1$。

构造哈夫曼树中，采用静态数组ht存储哈夫曼树，即每个数组元素存放一个结点。设计哈夫曼树中结点类如下：

```
class HTNode:                    #哈夫曼树结点类
  def __init__(self,d=" ",w=None):    #构造方法
    self.data=d                  #结点值
    self.weight=w                #权值
    self.parent=-1               #指向双亲结点
    self.lchild=-1               #指向左孩子结点
    self.rchild=-1               #指向右孩子结点
    self.flag=True               #标识是双亲的左(True)或者右(False)孩子
```

```python
def CreateHT():                          #构造哈夫曼树
  global ht,n0,D,W                       #全局变量，存放哈夫曼树等信息
  ht=[None]*(2*n0-1)                     #初始为含2n0-1个空结点
  heap=[]                                #优先队列元素为[w,i]，按w权值建立小根堆

  for i in range(n0):                    #i从0到n0-1循环建立n0个叶子结点并进队

    ht[i]=HTNode(D[i],W[i])              #建立一个叶子结点
    heapq.heappush(heap,[W[i],i])                    #将[W[i],i]进队

  for i in range(n0,2*n0-1):             #i从n0到2n0-2循环做n0-1次合并操作

    p1=heapq.heappop(heap)               #出队两个权值最小的结点p1和p2
    p2=heapq.heappop(heap)
    ht[i]=HTNode()                       #新建ht[i]结点
    ht[i].weight=ht[p1[1]].weight+ht[p2[1]].weight  #求权值和
    ht[p1[1]].parent=i                   #设置p1的双亲为ht[i]
    ht[i].lchild=p1[1]                   #将p1作为双亲ht[i]的左孩子
    ht[p1[1]].flag=True
    ht[p2[1]].parent=i                   #设置p2的双亲为ht[i]
    ht[i].rchild=p2[1]                   #将p2作为双亲ht[i]的右孩子
    ht[p2[1]].flag=False
    heapq.heappush(heap,[ht[i].weight,i])           #将新结点ht[i]进队
```

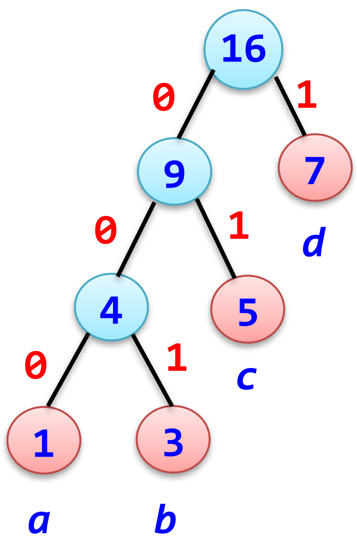
n_0=4, D=['a','b','c','d'], W=[1,3,5,7]



i（结点索引）	0	1	2	3	4	5	6
D[*i*]	a	b	c	d			
W[*i*]	1	3	5	7	4	9	16
parent	4	4	5	6	5	6	-1
lchild	-1	-1	-1	-1	0	4	3
rchild	-1	-1	-1	-1	1	2	5

6.7.3 哈夫曼编码

- 构造一棵哈夫曼树。
- 规定哈夫曼树中的左分支为0，右分支为1。
- 从根结点到每个叶子结点所经过的分支对应的0和1组成的序列便为该结点对应字符的编码。这样的编码称为哈夫曼编码。

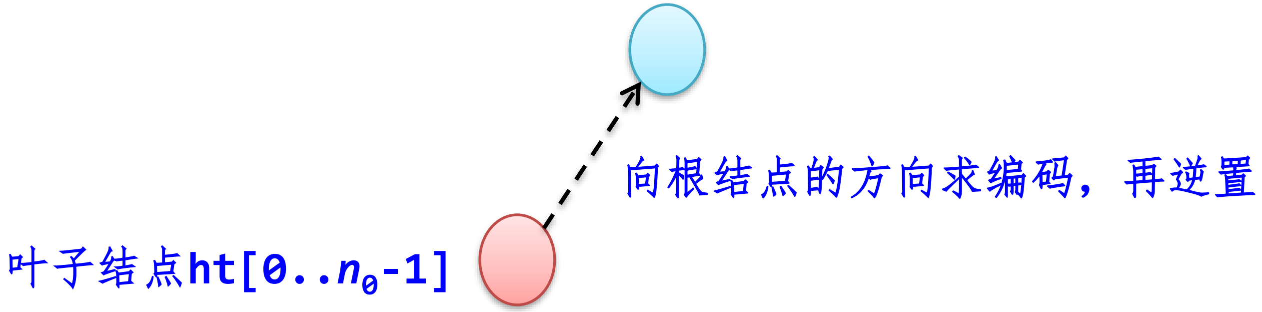


a: 000
b: 001
c: 01
d: 1

哈夫曼编码的实质就是使用频率越高的采用越短的编码。

只有ht[0..n_0-1]叶子结点才对应哈夫曼编码，用hcd[i]（$0 \leqslant i \leqslant n_0-1$）表示ht[$i$]叶子结点的哈夫曼编码。

```
def CreateHCode():                    #根据哈夫曼树求哈夫曼编码
  global n0,ht,hcd                    #hcd存放哈夫曼编码
  hcd=[]
  for i in range(n0):                 #遍历下标从0到n0-1的叶子结点
     code=[]                          #存放ht[i]结点的哈夫曼编码
     j=i                              #从ht[i]开始找双亲结点
     while ht[j].parent!=-1:
       if ht[j].flag:                 #ht[j]结点是双亲的左孩子
          code.append("0")
       else:                          #ht[j]结点是双亲的右孩子
          code.append("1")
       j=ht[j].parent
     code.reverse()                   #逆置code
     hcd.append(''.join(code))        #将code转换为字符串并添加到hcd中
```



输出所有叶子结点的哈夫曼编码的算法

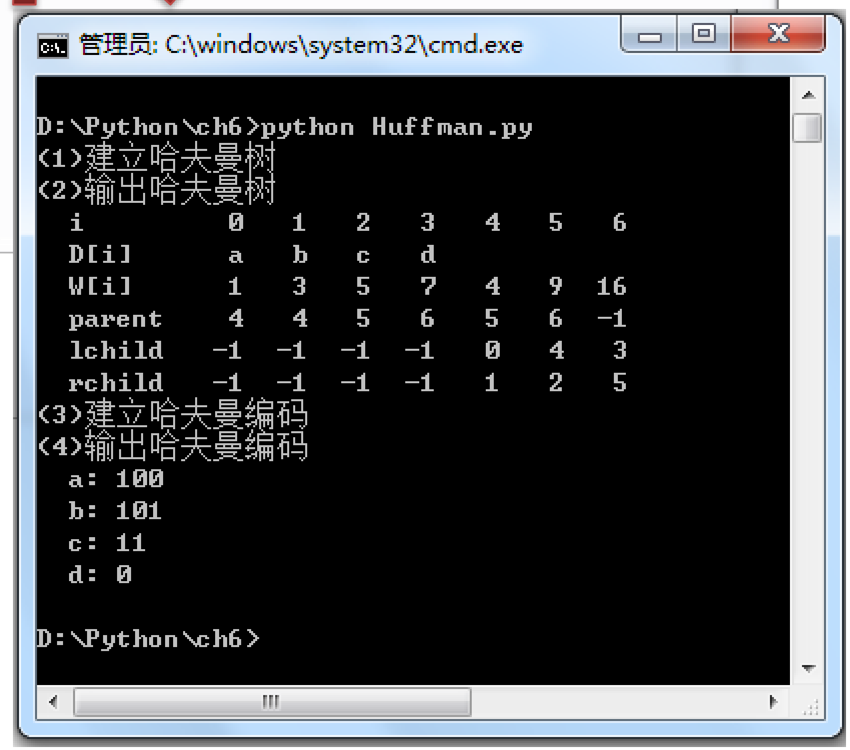
```
def DispHCode():                      #输出哈夫曼编码
   global hcd
   for i in range(len(hcd)):
     print("  "+ht[i].data+": "+hcd[i])
```

输出ht的算法

```
def DispHT():                         #输出哈夫曼树
   global n0,ht
   print("  i      ",end=' ')
   for i in range(2*n0-1):
      print("%3d" %(i),end=' ')
   print()
    …
```

```
if __name__ == '__main__':
    n0=4                                    #编码的字符个数
    D=['a','b','c','d']                     #字符列表
    W=[1,3,5,7]                             #权值列表
    print("(1)建立哈夫曼树")
    CreateHT()
    print("(2)输出哈夫曼树")
    DispHT()
    print("(3)建立哈夫曼编码")
    CreateHCode()
    print("(4)输出哈夫曼编码")
    DispHCode()
```

程序验证



提个醒

在一组字符的哈夫曼编码中，任一字符的哈夫曼编码不可能是另一字符哈夫曼编码的前缀。

2014年全国硕士研究生入学统一考试题

示例

6. 5个字符有如下4种编码方案，不是前缀编码的是（ ）。

A. 01,0000,0001,001,1

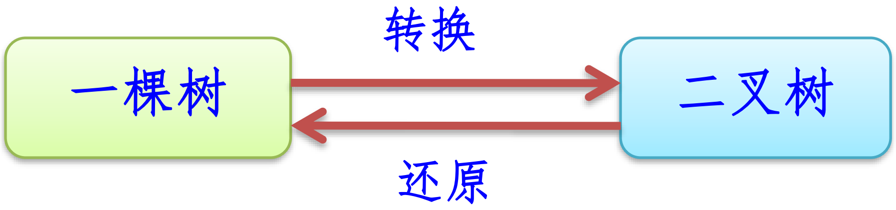
B. 011,000,001,010,1

C. 000,001,010,011,100

D. 0,100,110,1110,1100

6.8 二叉树与树、森林之间的转换

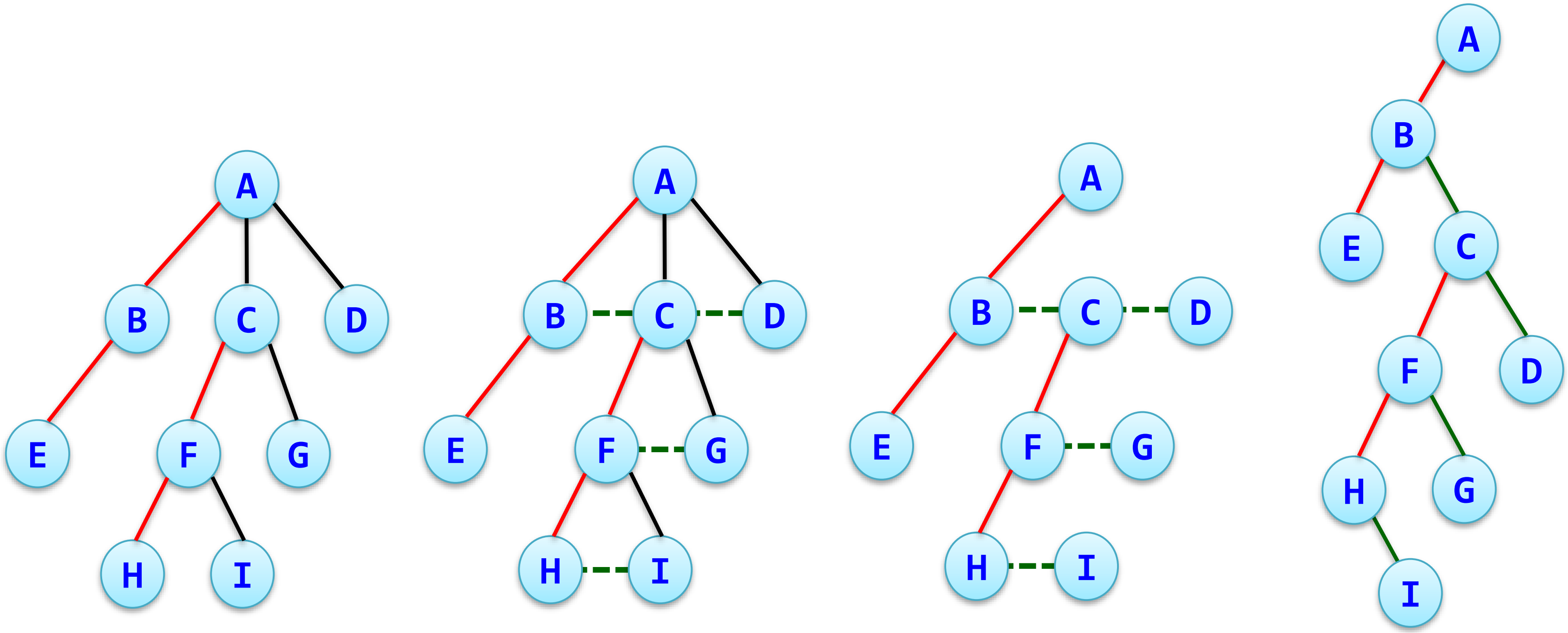
6.8.1 树到二叉树的转换及还原



1．树到二叉树的转换

一棵树可以按照以下规则转换为二叉树：

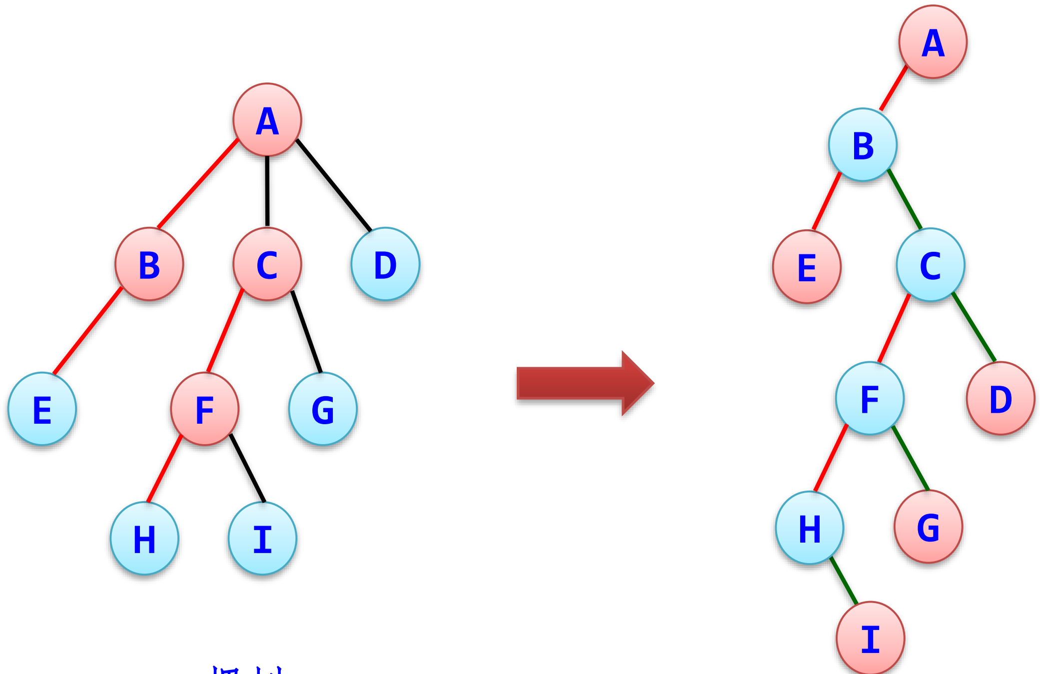
- 加线：在各兄弟结点之间加一连线，将其隐含的“兄一弟”关系以“双亲一右孩子”关系显示表示出来。
- 抹线：对任意结点，除了其最左子树之外，抹掉该结点与其他子树之间的“双亲一孩子”关系。
- 调整：以树的根结点作为二叉树的根结点，将树根与其最左子树之间的“双亲一孩子”关系改为“双亲一左孩子”关系，且将各结点按层次排列，形成二叉树。



（a）一棵树　　（b）加线　　（c）抹线　　（d）调整

转换成二叉树的特点

- 根结点只有左子树而没有右子树。
- 左分支不变（左分支为最左孩子），兄弟变成右分支（右分支实为双亲的兄弟）。



一棵树　　　　二叉树

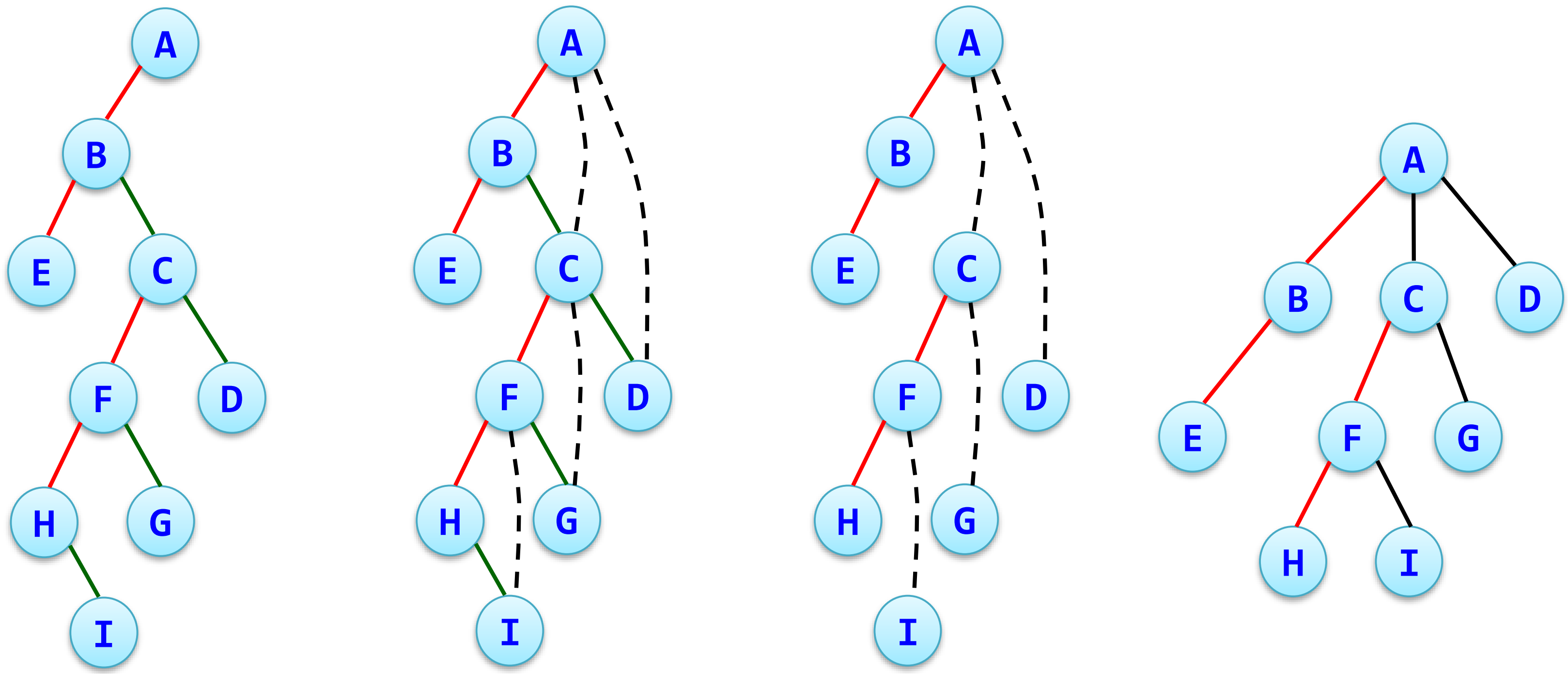
扩展

树中分支结点个数为*m*，则二叉树中无右孩子的结点个数为*m*+1。这里*m*=4。

2．一棵由树转换的二叉树还原为树

一棵二叉树（由一棵树转换的）可以按照以下规则还原为一棵树：

- 加线：在各结点的双亲与该结点右链上的每个结点之间加一连线，以“双亲—孩子”关系显示表示出来。
- 抹线：抹掉二叉树中所有双亲结点与其右孩子之间的“双亲—右孩子”关系。
- 调整：以二叉树的根结点作为树的根结点，将各结点按层次排列，形成树。

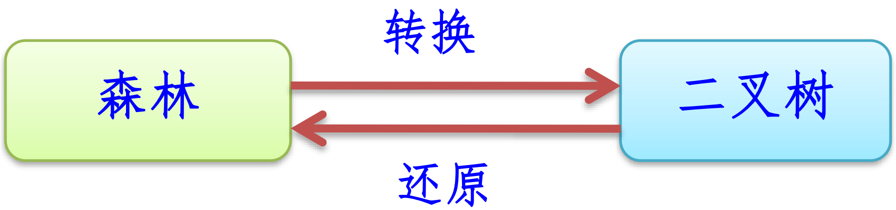


（a）一棵二叉树 （b）加线 （c）抹线 （d）调整

还原成一棵树的特点

- 根结点不变。
- 左分支不变（左分支为最左孩子），右分支变成兄弟。

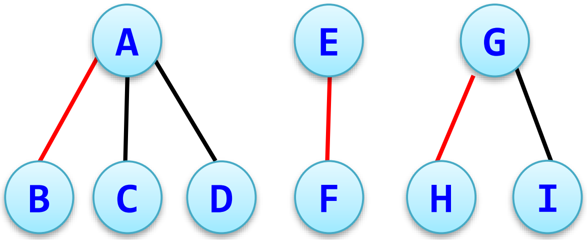
6.8.2 森林到二叉树的转换及还原



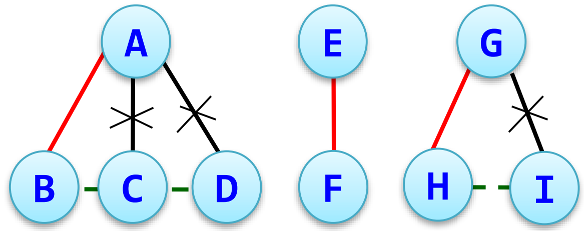
1．森林转换为二叉树

含有两棵或两棵以上树的森林可以按照以下规则转换为二叉树：

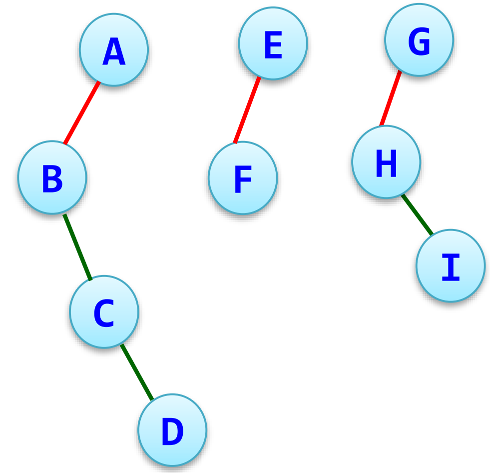
- **转换**：将森林中的每一棵树转换成二叉树，设转换成的二叉树为bt_1、bt_2、…、bt_m。
- **连接**：将各棵转换后的二叉树的根结点相连。
- **调整**：以bt_1的根结点作为整个二叉树的根结点，将bt_2的根结点作为bt_1的根结点的右孩子，将bt_3的根结点作为bt_2的根结点的右孩子，…，如此这样得到一棵二叉树，即为该森林转换得到的二叉树。



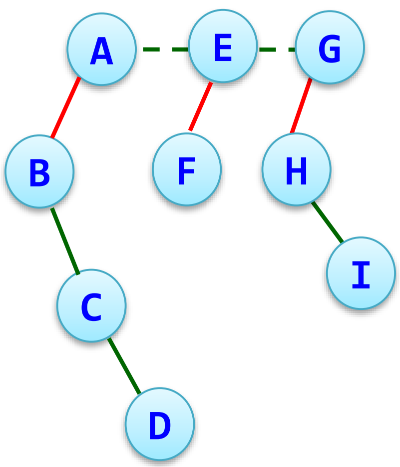
（a）森林



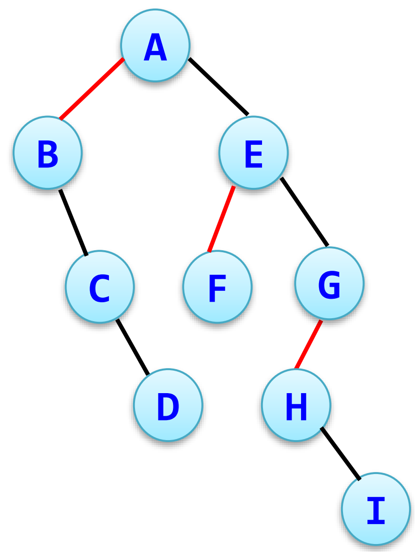
（b）转化为二叉树(1)



（c）转化为二叉树(2)



（d）连线

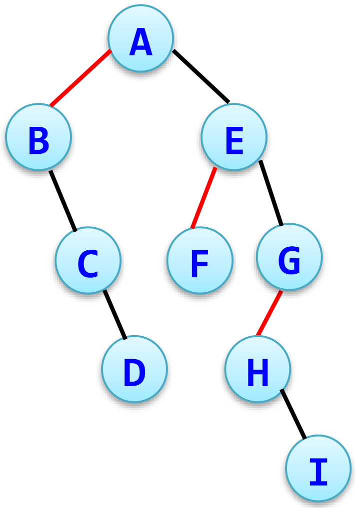


（e）转换成的二叉树

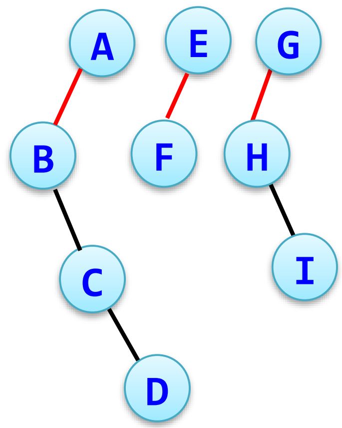
2．二叉树还原为森林

当一棵二叉树的根结点有***m*-1**个右下孩子，这样还原的森林中有***m***棵树。这样的二叉树可以按照以下规则还原其相应的森林：

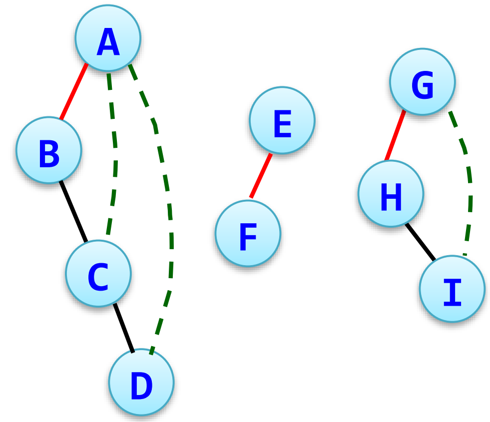
- 抹线：抹掉二叉树根结点右链上所有结点之间的“双亲—右孩子”关系，分成若干个以右链上的结点为根结点的二叉树，设这些二叉树为$\mathbf{bt}_1$、$\mathbf{bt}_2$、…、$\mathbf{bt}_m$。
- 转换：分别将$\mathbf{bt}_1$、$\mathbf{bt}_2$、…、$\mathbf{bt}_m$二叉树各自还原成一棵树。
- 调整：将转换好的树构成森林。



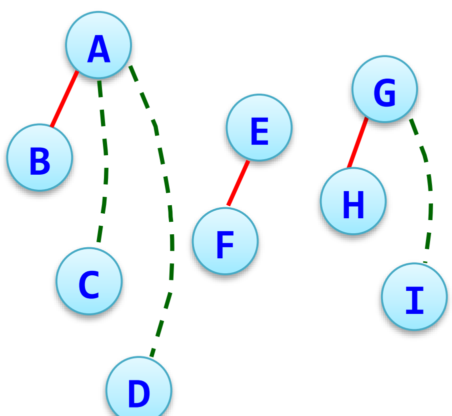
（a）一棵二叉树



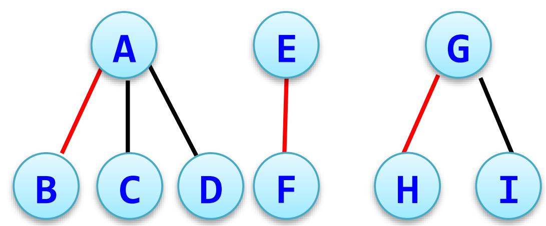
（b）抹线



（c）还原为树(1)



（d）还原为树(2)



（e）还原的森林