数据结构和算法
（Python描述）

郭 炜

微信公众号

微博：http://weibo.com/guoweiofpku

学会程序和算法，走遍天下都不怕!

讲义照片均为郭炜拍摄

哈夫曼树和堆

哈夫曼树
(最优二叉树)

木兰围场泰丰湖

最优二叉树

给定n个节点，节点i有权值W_i。要求构造一棵二叉树，叶子节点为给定的节点，且

$$\mathrm{WPL} = \sum_{i=1}^{n} W_i \times L_i$$

最小。Li 是节点i到树根的路径的长度。WPL: Weighted Path Length of Tree

最优二叉树也叫哈夫曼树

最优二叉树的构造

1) 开始n个节点位于集合S

2) 从S中取走两个权值最小的节点n1和n2，构造一棵二叉树，树根为节点r，r的两个子节点是n1和n2，且$W_r=W_{n1}+W_{n2}$，并将r加入S

3) 重复2）,直到S中只有一个节点，最优二叉树就构造完毕，根就是S中的唯一节点

证明较麻烦，略

显然，最优二叉树不唯一

哈夫曼编码树

需要对信息中用到的每个字符进行编码。

定长编码方案：每个字符编码的比特数都相同。比如ASCII编码方案。

A 000	C 010	E 100	G 110
B 001	D 011	F 101	H 111

BACADAEAFABBAAAGAH

被编码为以下54个bits:

001000010000011000100000101000001001000000000110000111

哈夫曼编码树

熵编码方案：使用频率高的字符，给予较短编码，使用频率低的字符，给予较长编码，如哈夫曼编码。

A 0	C 1010	E 1100	G 1110
B 100	D 1011	F 1101	H 1111

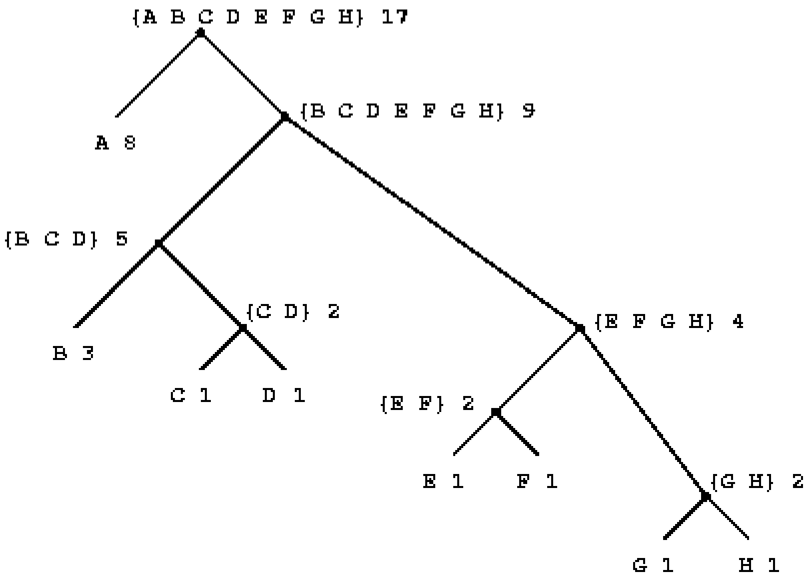
BACADAEAFABBAAAGAH

被编码为以下42个bits:

100010100101101100011010100100000111001111

哈夫曼编码树

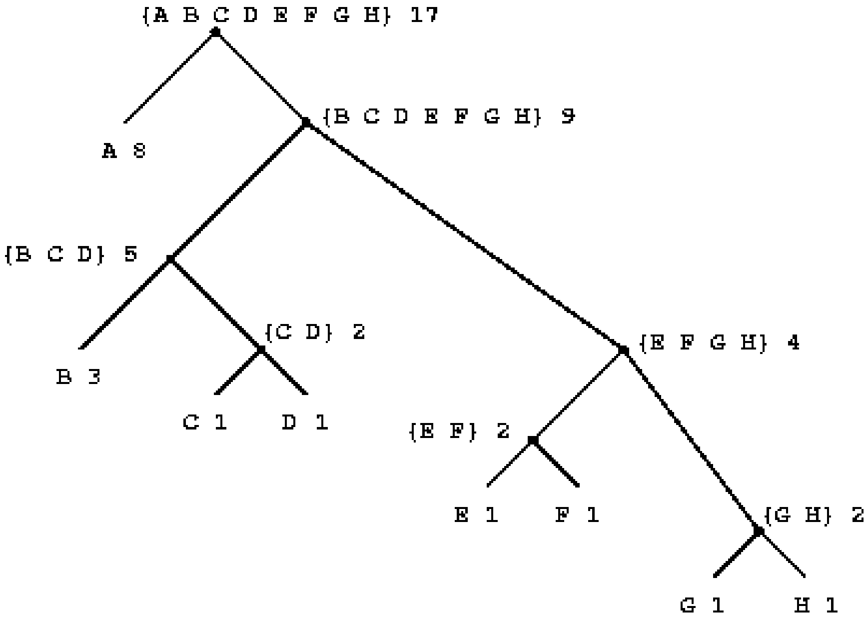
使用可变长编码，需要解决的问题是：如何区分一个编码是一个字符的完整编码，还是另一个字符的编码的前缀。解决办法之一就是采用前缀编码：任何一个字符的编码，都不会是其他字符编码的前缀。



哈夫曼编码树：

- 二叉树
- 叶子代表字符，且每个叶子节点有个权值，权值即该字符的出现频率
- 非叶子节点里存放着以它为根的子树中的所有字符，以及这些字符的权值之和
- 权值仅用来建树，对于字符串的解码和编码没有用处

哈夫曼编码树

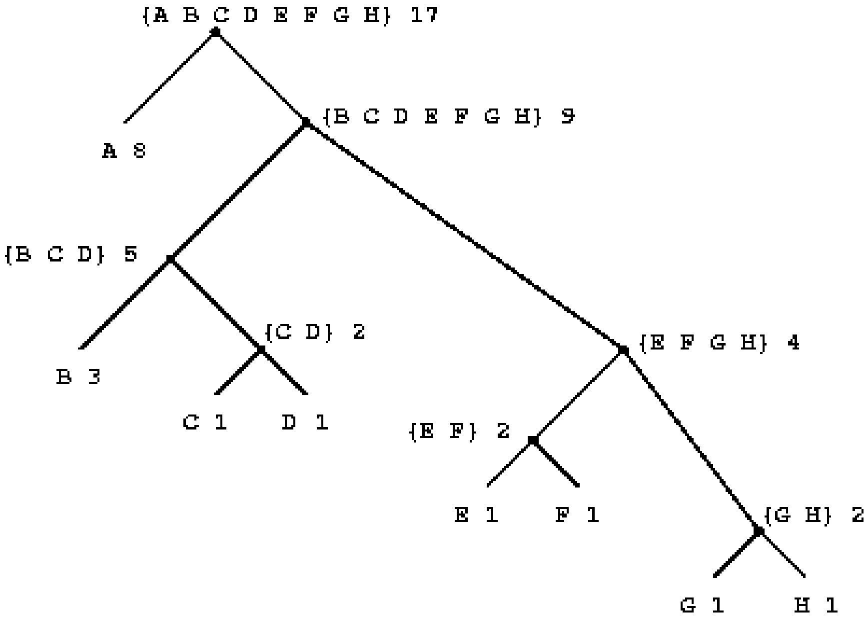


字符的编码过程：

从树根开始，每次往包含该字符的子树走。往左子树走，则编码加上比特1，往右子树走，则编码加上比特0

A 0
B 100
C 1010
G 1110
H 1111

哈夫曼编码树



字符串编码的解码过程：

从树根开始，在字符串编码中碰到一个0，就往左子树走，碰到1，就往右子树走。走到叶子，即解码出一个字符。然后回到树根重复前面的过程。

10001010
BAC

哈夫曼编码树的构造

基本思想：使用频率越高的字符，离树根越近。构造过程和最优二叉树一样

过程：

1． **开始时，若有n个字符，则就有n个节点。每个节点的权值就是字符的频率，每个节点的字符集就是一个字符。**

2． **取出权值最小的两个节点，合并为一棵子树。子树的树根的权值为两个节点的权值之和，字符集为两个节点字符集之并。在节点集合中删除取出的两个节点，加入新生成的树根。**

3． **如果节点集合中只有一个节点，则建树结束。否则，goto 2**

哈夫曼编码树的构造

```
Initial leaves
{(A 8) (B 3) (C 1) (D 1) (E 1) (F 1) (G 1) (H 1)}Merge
{(A 8) (B 3) ({C D} 2) (E 1) (F 1) (G 1) (H 1)}Merge
{(A 8) (B 3) ({C D} 2) ({E F} 2) (G 1) (H 1)}Merge
{(A 8) (B 3) ({C D} 2) ({E F} 2) ({G H} 2)}Merge
{(A 8) (B 3) ({C D} 2) ({E F G H} 4)}Merge
{(A 8) ({B C D} 5) ({E F G H} 4)} Merge
{(A 8) ({B C D E F G H} 9)}Final merge
{({A B C D E F G H} 17)}
```

哈夫曼编码树不唯一

如何快速地在节点集合取出权值最小的两个节点？不要O(n)的笨办法。
用"堆"，可以做到O(log(n))

最优二叉树例题

黄石大峡谷

Fence Repair

一块长木板，要切割成长度为$L_1,L_2...L_n$的n块板子。每切一刀的费用，等于被切的那块板子的长度。求最少费用。

Fence Repair

思路：

考虑等价的切割的逆过程，即用n块板子去粘接成最终的长板子。每粘接一次的费用等于粘成的木板长度。

将粘接过程中产生的每个木板，包括最终长木板，都看作一个节点。则粘接的过程可以描述成一棵树的建立过程。将n1,n2粘接成R ，就相当于建一棵以R为根， n1,n2为子节点的二叉树。最终长板就是最终二叉树的树根。

建树完成后，设n_i到根的路径长度为L_i，则其参加了次L_i粘接,贡献了费用$L_i \times W_i$。要使总费用最小就是 WPL = $\sum_{i=1}^{n} W_i \times Li$ 最小，即最优二叉树问题。

堆的定义、性质和用途

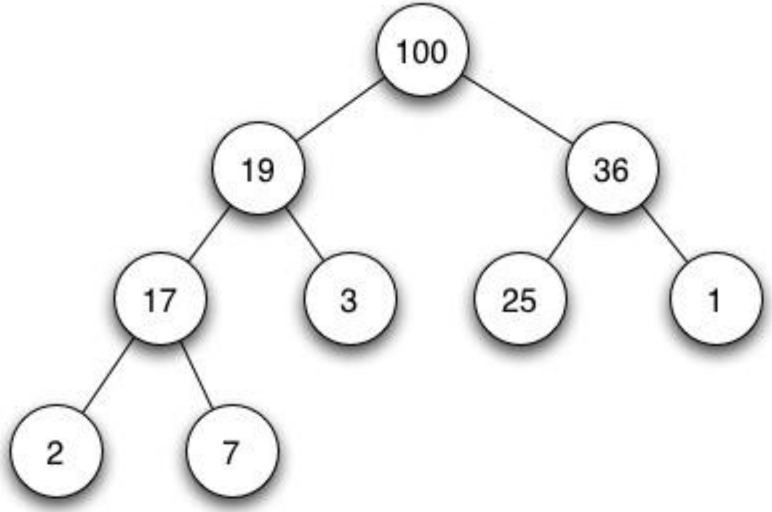
美国鹅颈湾

堆的定义

1) 堆(二叉堆)是一个完全二叉树

2) 堆中任何节点优先级都高于或等于其两个子节点（什么叫优先级高可以自己定义)

一个大就是优先级高的堆：



堆的存储

用列表存放堆。堆顶元素下标是0。下标为i的节点，其左右子节点下标分别为 i*2+1, i*2 + 2。

堆的性质

1) 堆顶元素是优先级最高的

2) 堆中的任何一棵子树都是堆

3) 往堆中添加一个元素，并维持堆性质，复杂度O(log(n))

4) 删除堆顶元素，剩余元素依然维持堆性质，复杂度O(log(n))

5) 在无序列表中原地建堆，复杂度O(n)

堆的作用

- 堆用于需要经常从一个集合中取走(即删除)优先级最高元素，而且还要经常往集合中添加元素的场合(堆可以用来实现优先队列)

- 可以用堆进行排序，复杂度O(nlog(n))，且只需要O(1)的额外空间,称为“堆排序”。递归写法需要o(log(n))额外空间，非递归写法需要O(1)额外空间。

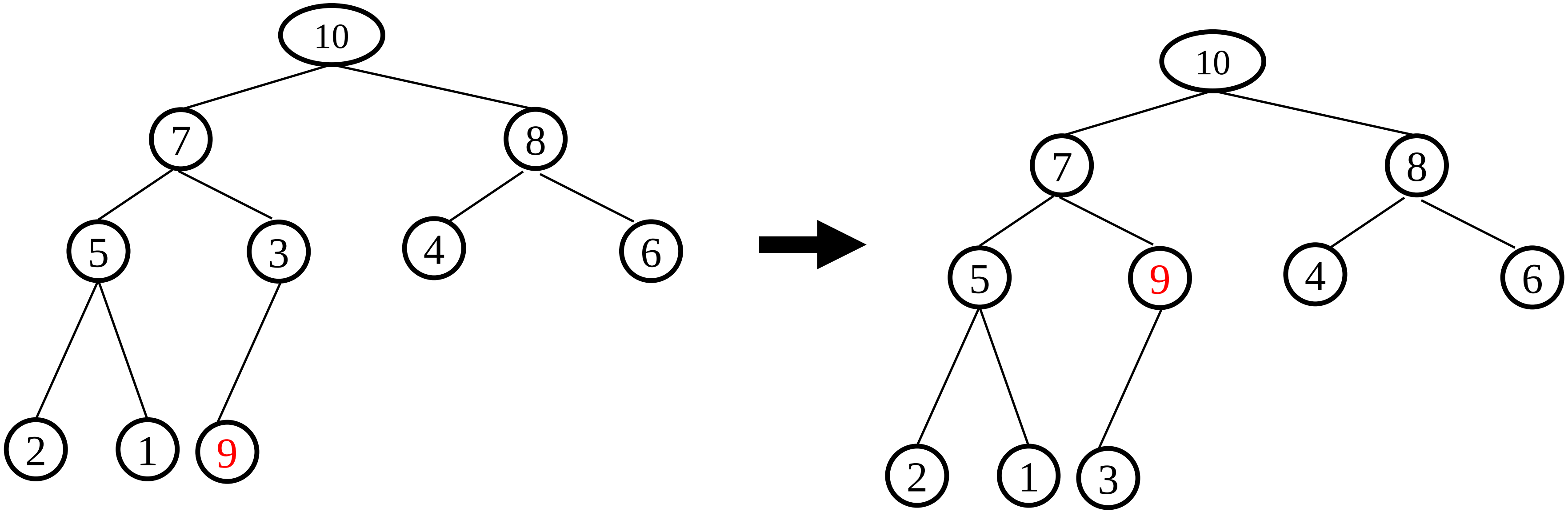
堆的操作

泰国普吉岛最南端

堆的操作：添加一个元素

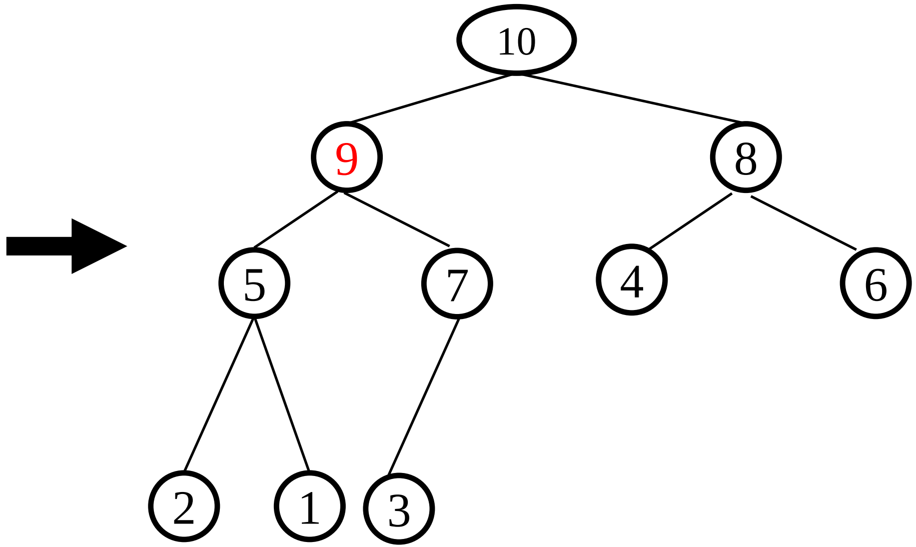
1) 假设堆存放在列表a中，长度为n
2) 添加元素x到列表a尾部，使其成为a[n]
3) 若x优先级高于其父节点，则令其和父节点交换，直到x优先级不高于其父节点，或x被交换到a[0]，变成堆顶为止。此过程称为将x"上移"
4) x停止交换后，新的堆形成，长度为n+1

堆的操作：添加一个元素



在堆中添加新元素9

堆的操作：添加一个元素



堆的操作：添加一个元素

显然，交换过程中，以x为根的子树，一直都是个堆

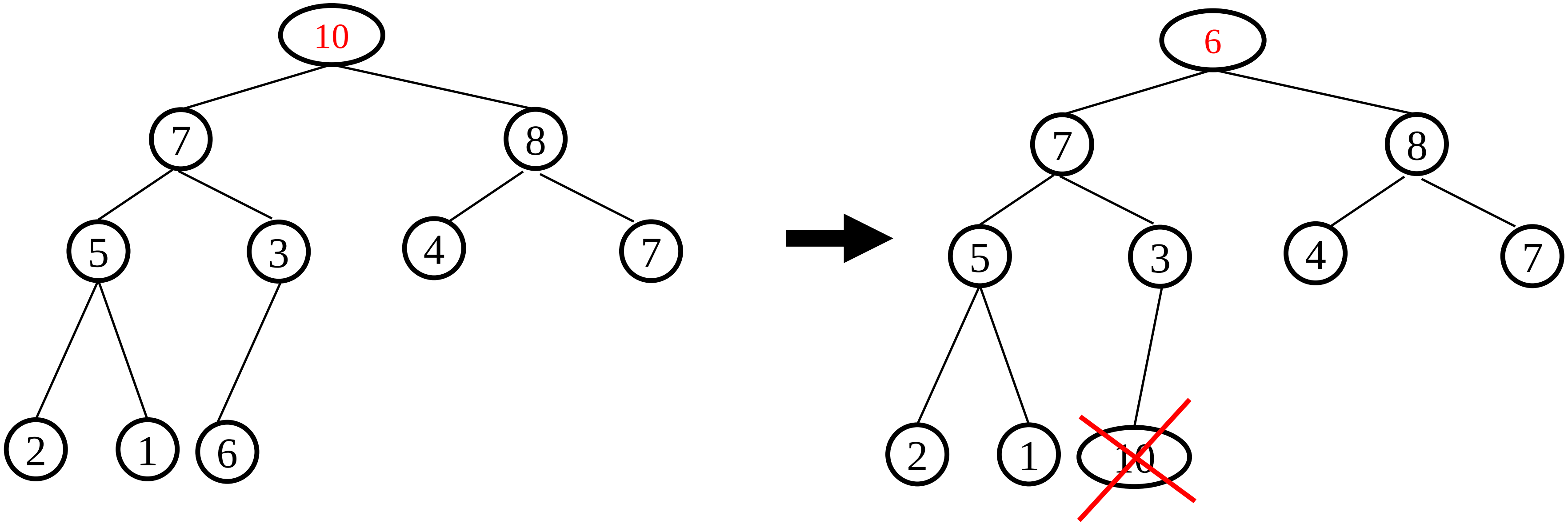
由于n个元素的完全二叉树高度为$\log_2(n+1)$向上取整，每交换一次x就上升一层，因此上移操作复杂度O(log(n))，即添加元素复杂度O(log(n))

堆的操作：删除堆顶元素

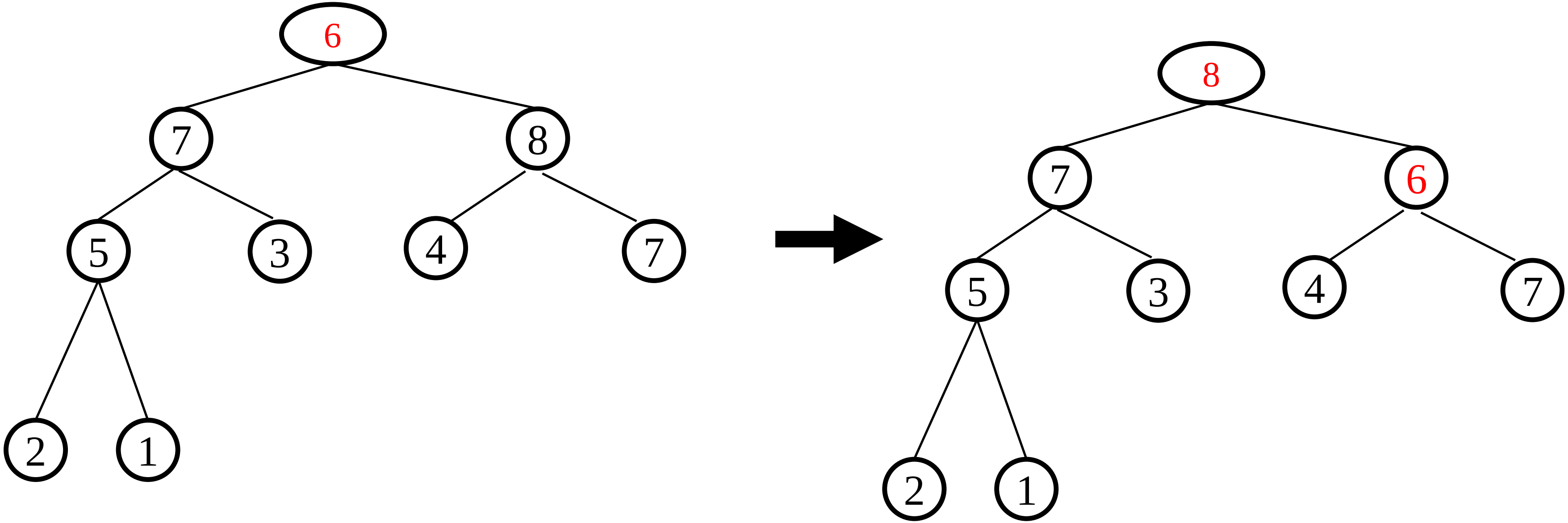
1) 假设堆存放在列表a中，长度为n
2) 将a[0]和a[n-1]交换
3) 将a[n-1]删除(pop)
4) 记此时的a[0]为x，则将x和它两个儿子中优先级较高的，且优先级高于x的那个交换，直到x变成叶子节点，或者x的儿子优先级都不高于x为止。将此整个过程称为将x"下移"
5) x停止交换后，新的堆形成，长度为n-1

下移过程复杂度为O(log(n))，因此删除堆顶元素复杂度O(log(n))

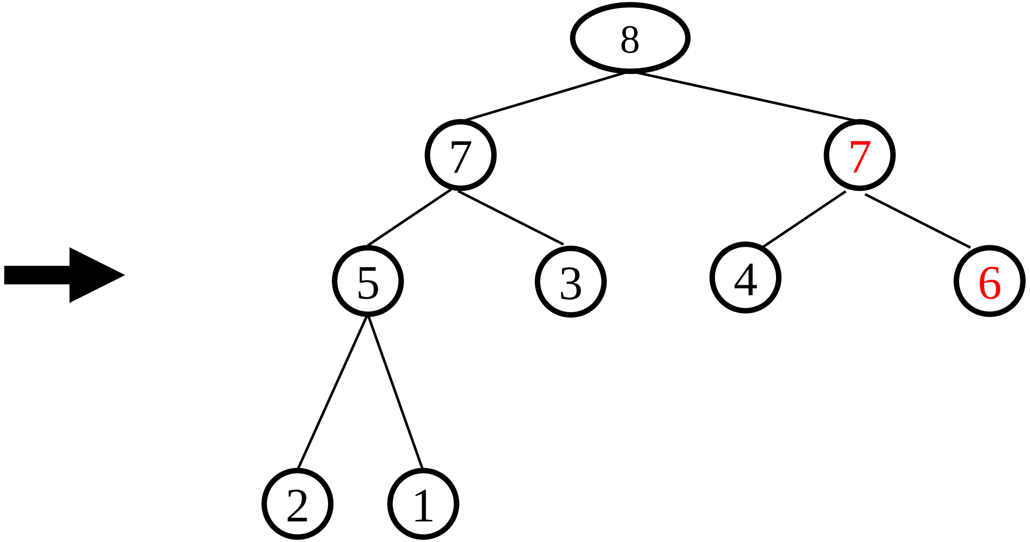
堆的操作：删除堆顶元素



堆的操作：删除堆顶元素



堆的操作：删除堆顶元素



堆的操作：删除堆顶元素

重要结论:

如果a[i]的两棵子树都是堆，则对a[i]的下移操作完成后，以新a[i]为根的子树会形成堆。

堆的操作：建堆

一个长度为n的列表a,要原地将a变成一个堆

方法:

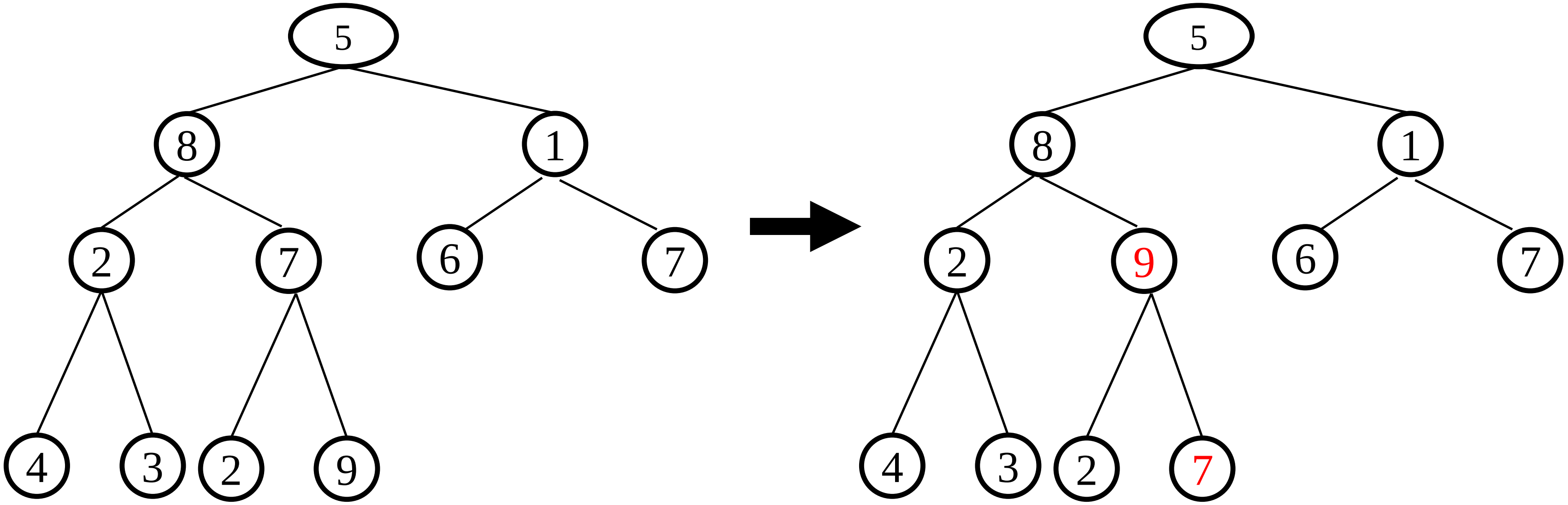
将a看作一个完全二叉树。假设有H层。根在第0层，第H-1层都是叶子

对第H-2层的每个元素执行下移操作
对第H-3层的每个元素执行下移操作
.....
对第0层的元素执行下移操作

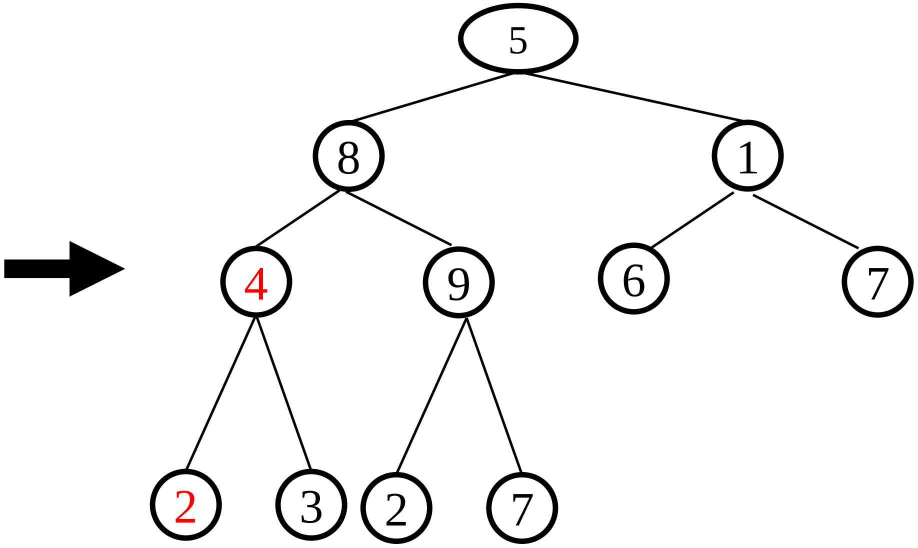
堆即建好。复杂度O(n)。证明较难，略

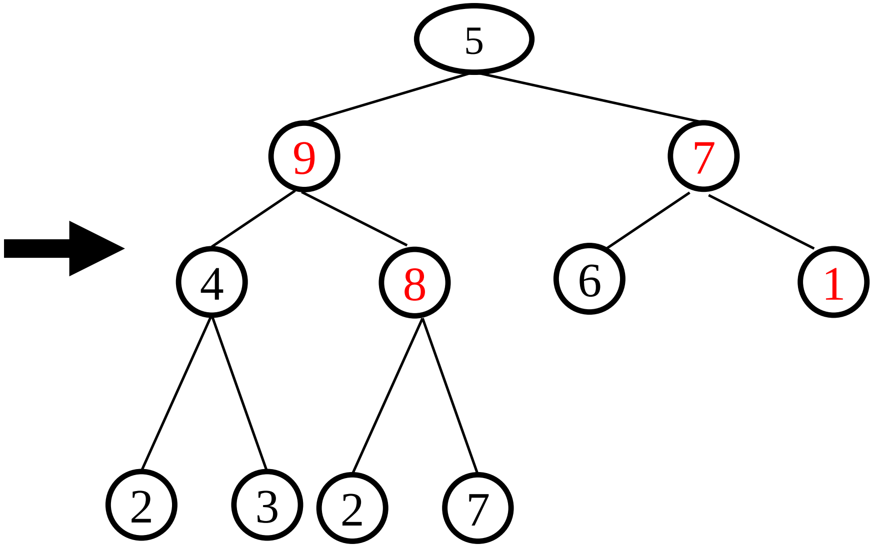
堆的操作：建堆

无序列表

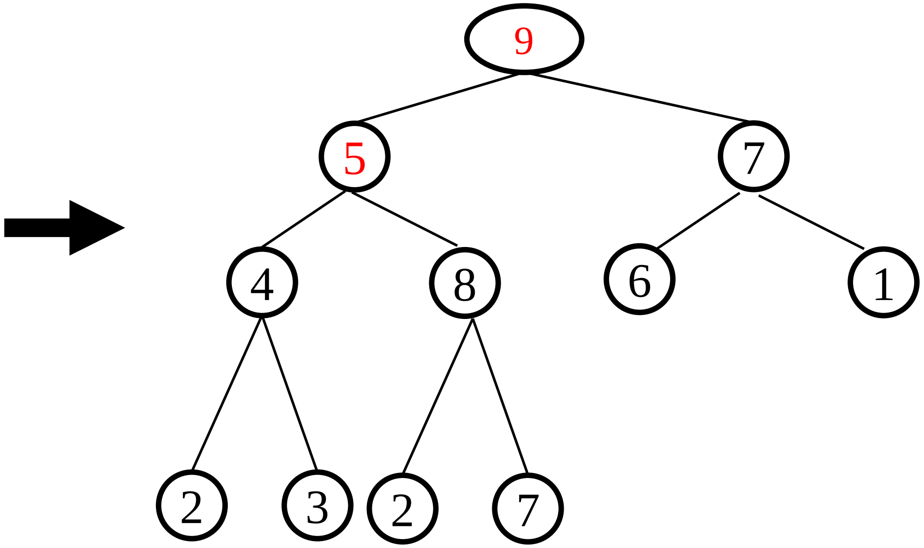


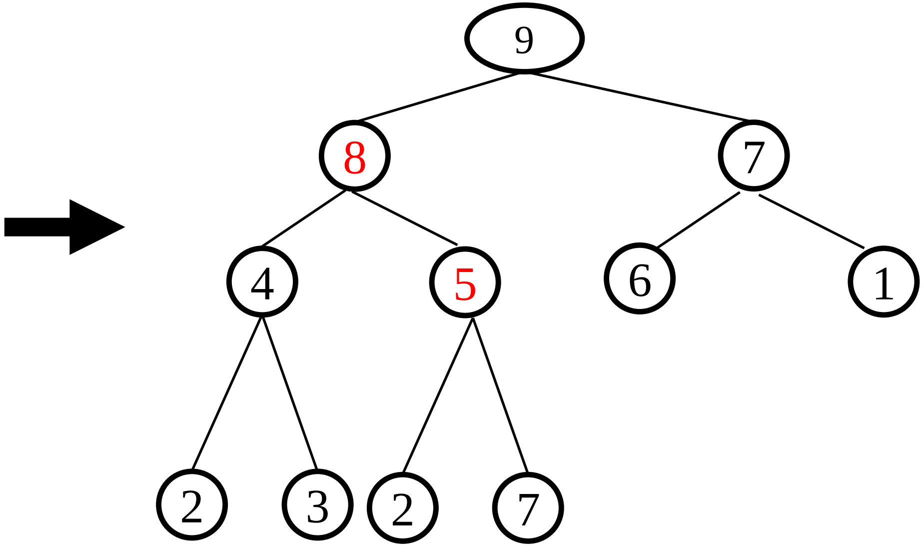
堆的操作：建堆



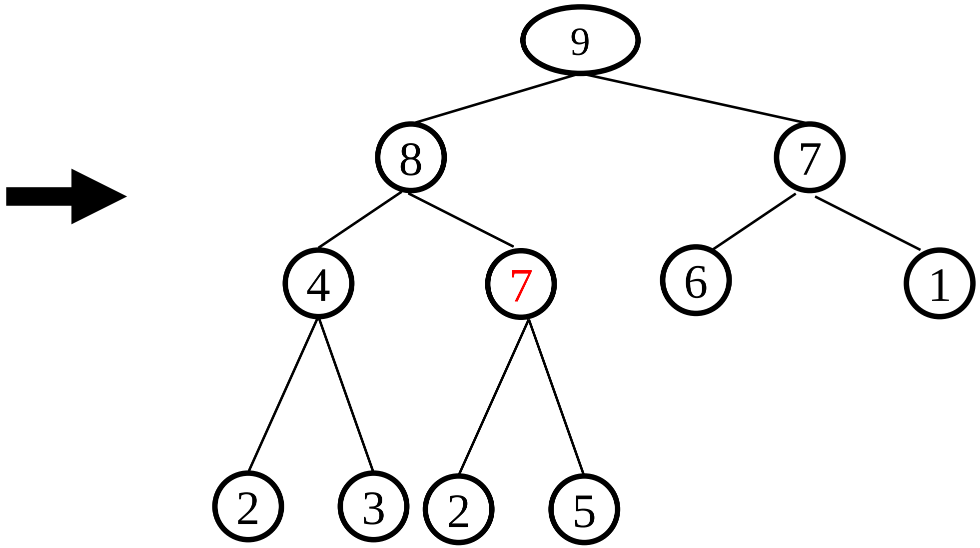


堆的操作：建堆





堆的操作：建堆



堆的应用

香港维多利亚湾

堆的应用：哈夫曼编码树的构造

```
Initial leaves
{(A 8) (B 3) (C 1) (D 1) (E 1) (F 1) (G 1) (H 1)}Merge
{(A 8) (B 3) ({C D} 2) (E 1) (F 1) (G 1) (H 1)}Merge
{(A 8) (B 3) ({C D} 2) ({E F} 2) (G 1) (H 1)}Merge
{(A 8) (B 3) ({C D} 2) ({E F} 2) ({G H} 2)}Merge
{(A 8) (B 3) ({C D} 2) ({E F G H} 4)}Merge
{(A 8) ({B C D} 5) ({E F G H} 4)} Merge
{(A 8) ({B C D E F G H} 9)}Final merge
{({A B C D E F G H} 17)}
```

哈夫曼编码树不唯一

用"堆"存放节点集合，便于快速取出最小权值的两个节点，以及加入合并后的新节点。

堆的应用：堆排序

1) 将待排序列表a变成一个堆(O(n))
2) 将a[0]和a[n-1]交换，然后对新a[0]做下移，维持前n-1个元素依然是堆。此时优先级最高的元素就是a[n-1]
3) 将a[0]和a[n-2]交换，然后对新a[0]做下移， 维持前n-2个元素依然是堆。此时优先级次高的元素就是a[n-2]

……

直到堆的长度变为1，列表a就按照优先级从低到高排好序了。

整个过程相当不断删除堆顶元素放到a的后部。堆顶元素依次是优先级最高的、次高的....

一共要做n次下移，每次下移O(log(n))，因此总复杂度O(nlog(n))

堆排序

如果用递归实现，需要O(log(n))额外栈空间(递归要进行log(n)层)。

如果不用递归实现，需要O(1)额外空间。

堆的实现

美国胡佛水坝

堆的实现

```
class heap:
    def __init__(self,array,better):
        #若x为堆顶元素，y为堆中元素，则 better(x,y)为True
        self.a = array #array是列表
        self.size = len(array)
        self.better = better  #better是比较函数
        #i的儿子是2*i+1和2*i+2
    def top(self):
        return self.a[0]
    def pop(self):#删除堆顶元素
        tmp = self.a[0]
        self.a[0] = self.a[-1]
        self.a.pop()
        self.size -= 1
        self.goDown(0)   #goDown是下移操作，将a[0]下移
        return tmp
```

堆的实现

```
def append(self,x):    #往堆中添加x
    self.size += 1
    self.a.append(x)
    self.goUp(self.size-1)  #goUp是上移
def goUp(self,i): #将a[i]上移
    #只在append的时候调用，不能直接调用或在别处调用
    #被调用时，以a[i]为根的子树，已经是个堆
    if i == 0:
        return
    f = ( i -1 )// 2  #父节点下标
    if self.better(self.a[i],self.a[f]):
        self.a[i],self.a[f]  = self.a[f],self.a[i]
        self.goUp(f) #a[f]上移
```

堆的实现

```
def goDown(self,i): #a[i]下移
    #前提：在a[i]的两个子树都是堆的情况下，下移
    if i * 2 + 1 >= self.size: #a[i]没有儿子
        return
    L,R = i * 2 + 1, i * 2 + 2
    if R >= self.size or self.better(self.a[L],self.a[R]):
        s = L
    else:
        s = R
    #上面选择大的儿子
    if self.better(self.a[s],self.a[i]):
        self.a[i],self.a[s] = self.a[s],self.a[i]
        self.goDown(s)
```

堆的实现

```
def makeHeap(self):  #建堆
    i = 0
    while i * 2 + 2 < self.size:
        i = i * 2 + 2
    #找倒数第二层最后一个节点的下标i
    for k in range(i,-1,-1):
        self.goDown(k)

def heapSort(self):  #建好堆之后调用，进行堆排序
    for i in range(self.size-1,-1,-1):
        self.a[i],self.a[0] = self.a[0],self.a[i]
        self.size -= 1
        self.goDown(0)
    self.a.reverse()
```

堆的实现

```
#下面是堆的用法，不是堆内部的代码
def heapSort(a,better):  #对列表a进行堆排序,哪个better哪个排在前面
    hp = heap(a,better)
    hp.makeHeap()
    hp.heapSort()


s = [i for i in range(17)]
random.shuffle(s)
print(s)
heapSort(s,lambda x,y : x < y)
print(s)
```

Python中的堆


新疆安集海大峡谷

python中的堆：heapq

```
import heapq

heapq.heapify(s)            将列表s变成一个堆
heapq.heappush(s,item)      往已经是堆的s里面添加元素item
heapq.heappop(s)            弹出堆顶元素(会减少s长度)
......

def heapsort(iterable): #iterable是个序列
#函数返回一个列表，内容是iterable中元素排序的结果
    h = []
    for value in iterable:
        h.append(value)
    heapq.heapify(h)
    return [heapq.heappop(h) for i in range(len(h))]
```

不便之处：没有设定排序规则的机会。如果要形成大元素在顶的整数堆，只能取相反数进堆。出来的时候再取相反数