2 数学基础与数据结构

2 数学基础与数据结构

- 集合、关系和函数
- 求和计算
- 递归方程求解 (*)
- Master定理(*)
- 基本数据结构 (*)

递归方程求解

- 常系数线性同质递归方程 (linear homogeneous with constant coefficients)
- 几类特殊的非同质递归方程求解

常系数线性同质递归方程

$$T(n) = a_1 T(n-1) + a_2 T(n-2) + \cdots + a_k T(n-k)$$

称为k 阶常系数线性同质递归方程。

其特征方程(characteristic equation)为：

$$x^k = a_1 x^{k-1} + a_2 x^{k-2} + \cdots + a_k \quad \Rightarrow \quad x^k - a_1 x^{k-1} - a_2 x^{k-2} - \cdots - a_k = 0$$

我们仅仅关注一阶和二阶同质方程，因为对于使用此类递归方程来分析算法，往往是一阶或二阶的。对于一阶情形，求解过程非常直观(直接递推)：

$$T(n) = a\ \ T(n-1) = a^2 T(n-2) = \cdots = a^n T(0)$$

对于二阶情形，特征方程变为：

$$x^2 - a_1 x - a_2 = 0$$

设该二次方程(quadratic equation)的两个根为：x_1, x_2 。那么 $T(n)$ 可以表示为：

$$T(n) = \begin{cases} c_1 \cdot x_1^{\,n} + c_2 \cdot x_2^{\,n} & ,if \quad x_1 \neq x_2 \\ c_1 \cdot r^{\,n} + c_2 \cdot n \cdot r^{\,n} & ,if \quad x_1 = x_2 = r \end{cases}$$

然后利用初始值：$T(n_0)$ 及 $T(n_0+1)$,使用待定系数法解出系数即可。

例：

已知 $T(n)=3T(n-1)+4T(n-2)$ 并且 $T(0)=1,T(1)=4$

解：特征方程为 $x^2-3x-4=0$ ⇒ $x_1=-1,x_2=4$

$$\therefore T(n)=c_1(-1)^n+c_2 4^n \Rightarrow \begin{cases} T(0)=1=c_1+c_2 \\ T(1)=4=-c_1+4c_2 \end{cases}$$

$$\Rightarrow c_1=0,c_2=1 \Rightarrow T(n)=4^n$$

举例

- 斐波那契数列是由0，1开始，之后的每一项等于前两项之和：

  0，1，1，2，3，5，8，13，21，34，55，89，144..... 。

递归形式的算法：

```
procedure Fib(n)
    if n=1 or n=2 then return 1
    else return Fib(n-1)+Fib(n-2)
```

时间复杂度分析：

已知 $T(n)=T(n-1)+T(n-2)$ 并且 $T(1)=T(2)=1$

解：特征方程为 $x^2-x-1=0$ ⇒ $x_1=\frac{1+\sqrt{5}}{2}, x_2=\frac{1-\sqrt{5}}{2}$

$$\therefore T(n)=c_1\left(\frac{1+\sqrt{5}}{2}\right)^n+c_2\left(\frac{1-\sqrt{5}}{2}\right)^n$$

$$\Rightarrow c_1=\frac{1}{\sqrt{5}}, c_2=-\frac{1}{\sqrt{5}}$$

$$\Rightarrow T(n)=\frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n-\frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n$$

非同质递归方程的求解

$$T(n) = T(n-1) + g(n), n \geq 1 \Rightarrow T(n) = T(0) + \sum_{i=1}^{n} g(i)$$

$$T(n) = g(n)T(n-1), n \geq 1 \Rightarrow T(n) = g(n)g(n-1)\cdots g(1)T(0)$$

$$T(n) = g(n)T(n-1) + h(n), n \geq 1 \Rightarrow ?$$

$$T(n)=g(n)T(n-1)+h(n),n\geq 1 \quad \text{且 } T(0) \text{ 已知。}$$

解：令 $$k(n)=\begin{cases}\dfrac{T(n)}{g(n)g(n-1)\cdots g(1)},n\geq 1\\ T(0),n=0\end{cases}$$

$$k(n)=\frac{T(n)}{g(n)g(n-1)\cdots g(1)},n\geq 1 \Rightarrow T(n)=g(n)g(n-1)\cdots g(1)k(n)$$

$$\Rightarrow T(n-1)=g(n-1)g(n-2)\cdots g(1)k(n-1)$$

$$\Rightarrow \underbrace{g(n)g(n-1)\cdots g(1)k(n)}_{T(n)}=g(n)\underbrace{g(n-1)\cdots g(1)k(n-1)}_{T(n-1)}+h(n)$$

$$\Rightarrow k(n)=k(n-1)+\frac{h(n)}{g(n)\cdots g(1)}$$

$$\Rightarrow \quad k(n) = k(n-2) + \frac{h(n-1)}{g(n-1)\cdots g(1)} + \frac{h(n)}{g(n)\cdots g(1)}$$

$$\Rightarrow \quad k(n) = k(0) + \sum_{i=1}^{n} \frac{h(i)}{g(i)g(i-1)\cdots g(1)} = T(0) + \sum_{i=1}^{n} \frac{h(i)}{g(i)g(i-1)\cdots g(1)}$$

$$\Rightarrow \quad T(n) = g(n)g(n-1)\cdots g(1)\left(T(0) + \sum_{i=1}^{n} \frac{h(i)}{g(i)g(i-1)\cdots g(1)} \right)$$

例： $T(n) = n\ \ T(n-1) + n!, T(0) = 0$

解：这里 $g(n) = n, h(n) = n!$

$$\therefore T(n) = n(n-1)\cdots 1(0 + \sum_{i=1}^{n} \frac{i!}{i!}) = n!n$$

Master Theorem（主定理）

设$a \geq 1$，$b > 1$为常数。$s(n)$为一给定的函数，$T(n)$递归定义如下：

$$T(n) = a \cdot T(n/b) + s(n)$$

并且$T(n)$有适当的初始值。那么，当n充分大时，有：

(1)若存在$\varepsilon > 0$，使得$s(n) = O(n^{\log_b(a)-\varepsilon})$ 成立，那么有$T(n) = \Theta(n^{\log_b(a)})$

(2)若$s(n) = \Theta(n^{\log_b(a)})$，那么$T(n) = \Theta(n^{\log_b(a)} \cdot \log n)$

(3)若存在$\varepsilon > 0$，使得$s(n) = \Omega(n^{\log_b(a)+\varepsilon})$ 成立，并且存在$c < 1$，使得$a \cdot s(n/b) \leq c \cdot s(n)$，那么有$T(n) = \Theta(s(n))$

参考：《算法导论》

几个例子

$$T(n) = 9T(n/3) + n$$

$$\because a = 9, b = 3, s(n) = n \quad \therefore \log_b^a = 2 \quad \therefore \exists \varepsilon = 1$$

$$s(n) = O(n^{\log_b^a - \varepsilon}) \qquad \therefore T(n) = \Theta(n^2)$$

$$T(n) = T(2n/3) + 1$$

$$\because a = 1, b = \tfrac{3}{2}, s(n) = 1 \quad \therefore \log_b^a = 0$$

$$s(n) = \Theta(n^{\log_b^a}) \qquad \therefore T(n) = \Theta(\log n)$$

$$T(n) = 3T(n/4) + n\log n$$

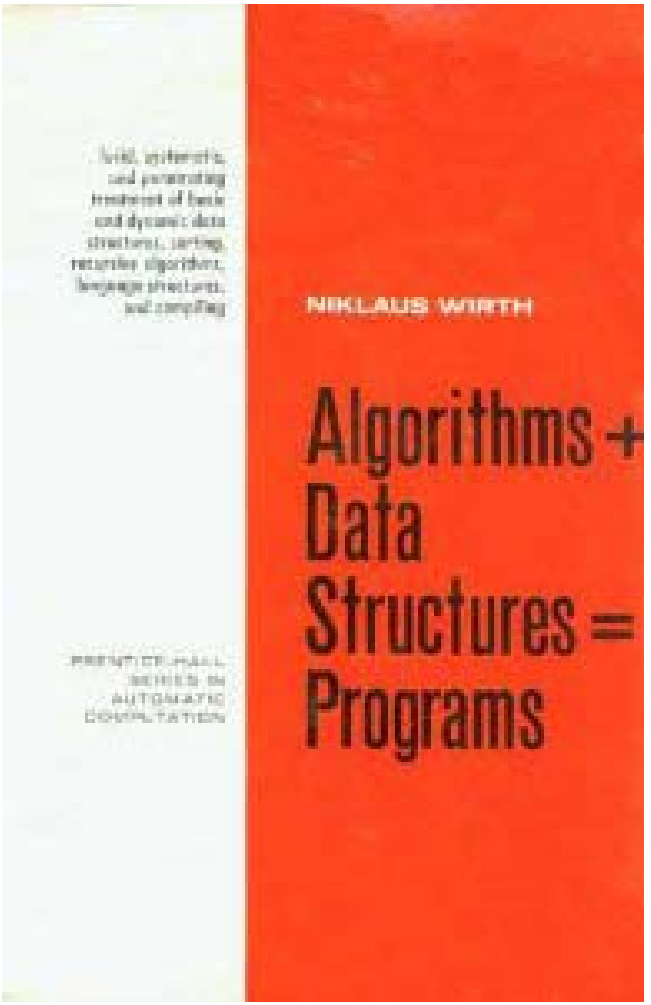
$$\because a = 3, b = 4, \log b^a \approx 0.793, s(n) = n\log n$$

$$\therefore \exists \varepsilon = 0.21, s(n) = n\log n = \Omega(n^{0.79+0.21})$$

- $\therefore T(n) = \Theta(n\log n)$

数据结构

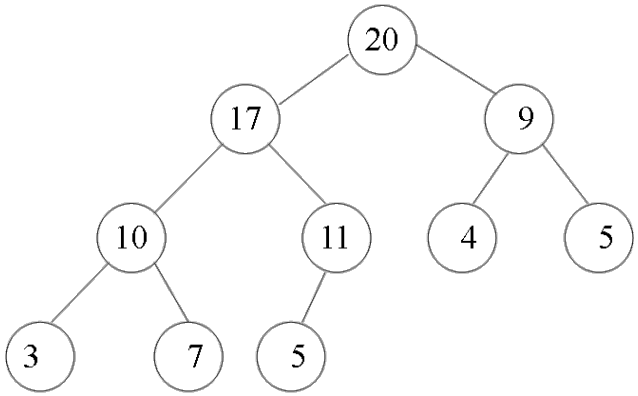
- 算法的实现离不开数据结构。选择一个合适的数据结构对设计一个有效的算法有十分重要的影响。结构化程序设计创始人Niklaus Wirth(瑞士苏黎士高工)提出一个著名的论断：“程序=算法+数据结构”。1984年，Wirth因开发了Euler、Pascal等一系列崭新的计算语言而荣获图灵奖，有“结构化程序设计之父”之美誉。
- 本章我们将回顾几种重要的数据结构，包括二叉树、**堆**、**不相交集**。



堆(Heap)

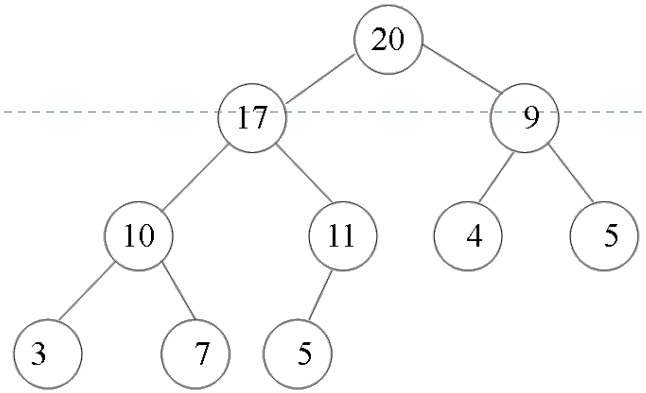
- 在许多算法中，需要大量用到如下两种操作：插入元素和寻找最大(小)值元素。为了提高这两种运算的效率，必须使用恰当的数据结构。
  - 普通队列：易插入元素，但求最大(小)值元素需要搜索整个队列。
  - 排序数组：易找到最大(小)值，但插入元素需要移动大量元素。
  - 堆则是一种有效实现上述两种运算的简单数据结构。

- 堆的定义：堆是一个几乎完全的二叉树，每个节点都满足这样的特性：任一父节点的键值(key)不小于子节点的键值。(最大堆)



- 有n个节点的堆T,可以用一个数组H[1…n]用下面的方式来表示：
  - T的根节点存储在H[1]中
  - 假设T的节点x存储在H[j]中，那么，它的左右子节点分别存放在H[2j]及H[2j+1]中(如果有的话)。
  - H[j]的父节点如果不是根节点，则存储在H[⌊ j/2⌋]中。

20	17	9	10	11	4	5	3	7	5
1	2	3	4	5	6	7	8	9	10



观察结论：

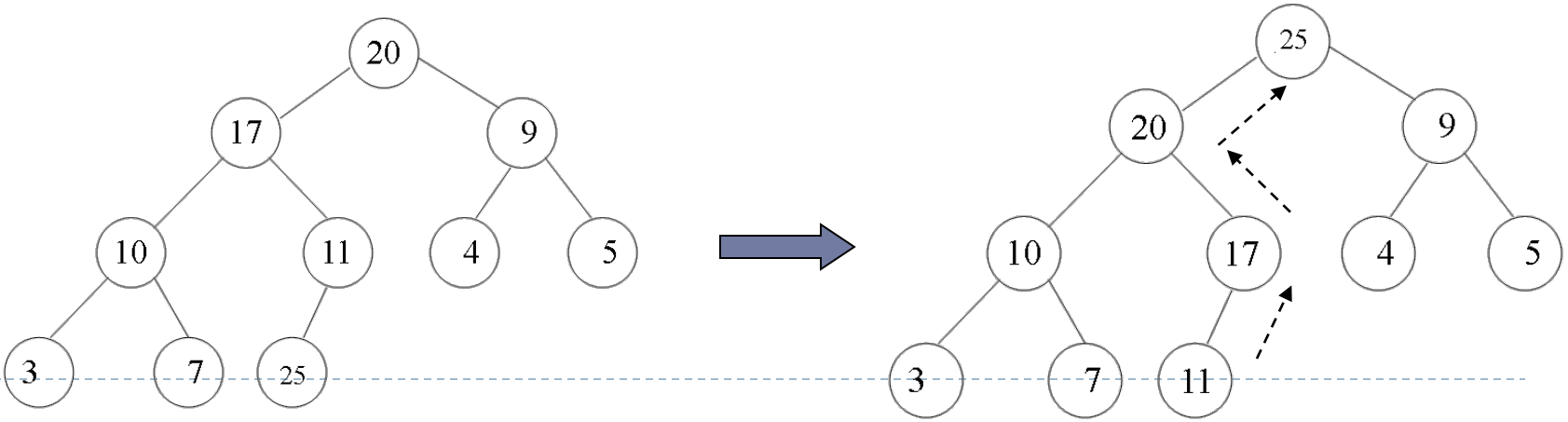
- 根节点键值最大，叶子节点键值较小。从根到叶子，键值以非升序排列。
- 节点的左右儿子节点键值并无顺序要求。
- 堆的数组表示呈“基本有序”状态。相应地，并非节点的高度越高，键值就越大。

堆的基本操作

- make-heap(A): 从数组A创建堆
- insert(H,x): 插入元素x到堆H中
- delete(H,i): 删除堆H的第i项
- delete-max(H): 从非空堆H中删除最大键值并返回数据项

辅助运算Sift-up

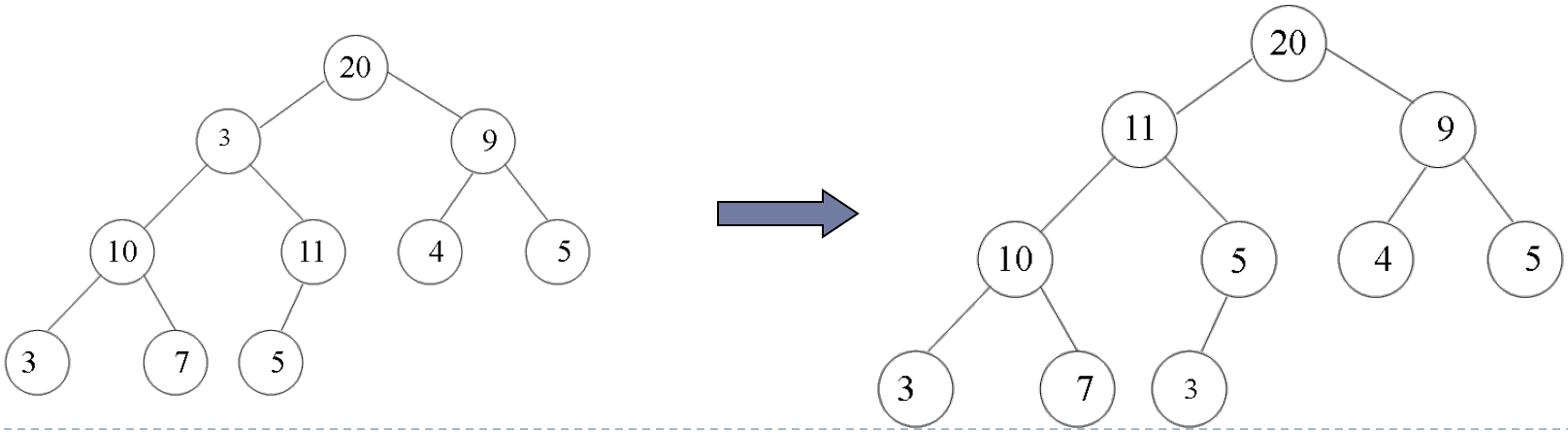
- 若某个节点H[i]键值大于其父节点的键值，就违背了堆的特性，需要进行调整。
- 调整方法：上移。
- 沿着H[i]到根节点的唯一一条路径，将H[i]移动到合适的位置上：比较H[i]及其父节点H[⌊i/2⌋]的键值，若key(H[i])>key(H[⌊i/2⌋])，则二者进行交换，直到H[i]到达合适位置。



过程 Sift-up(H,i)
输入：数组H[1…n]，索引1≤ i≤n
输出：上移H[i] (如果需要)，使它的键值不大于父节点的键值
1. done←false
2. if i=1 then exit {根节点}
3. repeat
4. if key(H[i])>key(H[⌊i/2⌋]) then 互换 H[i] 和H[⌊i/2⌋]
5. else done←true {调整过程至此已经满足要求，可退出}
6. i←⌊i/2⌋
7. until i=1 or done {调整进行到根节点，或到某一节点终止}

辅助运算Sift-down

- 假如某个内部节点H[i] (i≤⌊n/2⌋)，其键值小于儿子节点的键值，即key(H[i])<key(H[2i])或key(H[i] <key(H[2i+1]) (如果右儿子存在)，违背了堆特性，需要进行调整。
- 调整方法：下渗。
- 沿着从H[i]到子节点(可能不唯一，则取其键值较大者)的路径，比较H[i]与子节点的键值，若key(H[i]) < max(H[2i], H[2i+1])则交换之。这一过程直到叶子节点或满足堆特性为止。



过程 Sift-down(H,i)
输入：数组H[1…n]，索引1≤ i≤n
输出：下滲H[i] (若它违背了堆特性)，使H满足堆特性
1. done←false
2. if 2i>n, then exit {叶子节点，无须进行}
3. repeat
4. i←2i
5. if i+1<n and key(H(i+1))> key(H(i)) then i=i+1 //有右儿子,取
//左右孩子中较大者
6. if key(H[⌊i/2⌋])< key(H[i]) then 互换 H[i] 和 H[⌊i/2⌋]
7. else done←true {调整过程至此已经满足堆特性，可退出}
8. end if
9. until 2i>n or done {调整进行到叶节点，或到某一节点终止}

操作insert(H,x): 插入元素x到堆H中

- 思路：先将x添加到H的末尾，然后利用Sift-up，调整x在H中的位置，直到满足堆特性。

输入：堆H[1…n]和元素x
输出：新堆H[1…n+1]，x是其中元素之一。
1. n←n+1 {堆大小增1}
2. H[n] ← x;
3. Sift-up(H,n) {调整堆}

树的高度为⌊logn⌋,所以将一个元素插入大小为n的堆所需要的时间是O(logn).

操作 delete(H,i)

- 思路：先用H[n]取代H[i]，然后对H[i]作Sift-up或Sift-down)，直到满足堆特性。

输入：非空堆H[1…n]，索引i，1≤i≤n.
输出：删除H[i]之后的新堆H[1…n-1].
1. x←H[i]; y←H[n];
2. n←n-1; {堆大小减1}
3. if i=n+1 then exit {要删除的刚好是最后一个元素，叶节点}
4. H[i]←y; {用原来的H[n]取代H[i]}
5. if key(y) ≥key(x) then Sift-up(H,i)
6. else Sift-down(H,i);
7. end if

所需要的时间是O(logn).

操作delete-max(H)

输入：堆H[1…n]
输出：返回最大键值元素，并将其从堆中删除
1. x←H[1]
2. delete(H,1)
3. return x

make-heap(A): 从数组A创建堆

- 方法1：从一个空堆开始，逐步插入A中的每个元素，直到A中所有元素都被转移到堆中。
- 时间复杂度为O(nlogn).为什么？（阅读教材）

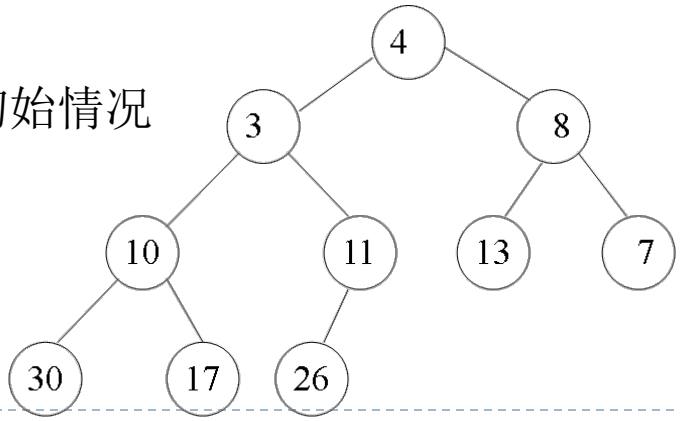
$$\sum_{j=1}^{n} \log j = \Theta(n \log n)$$

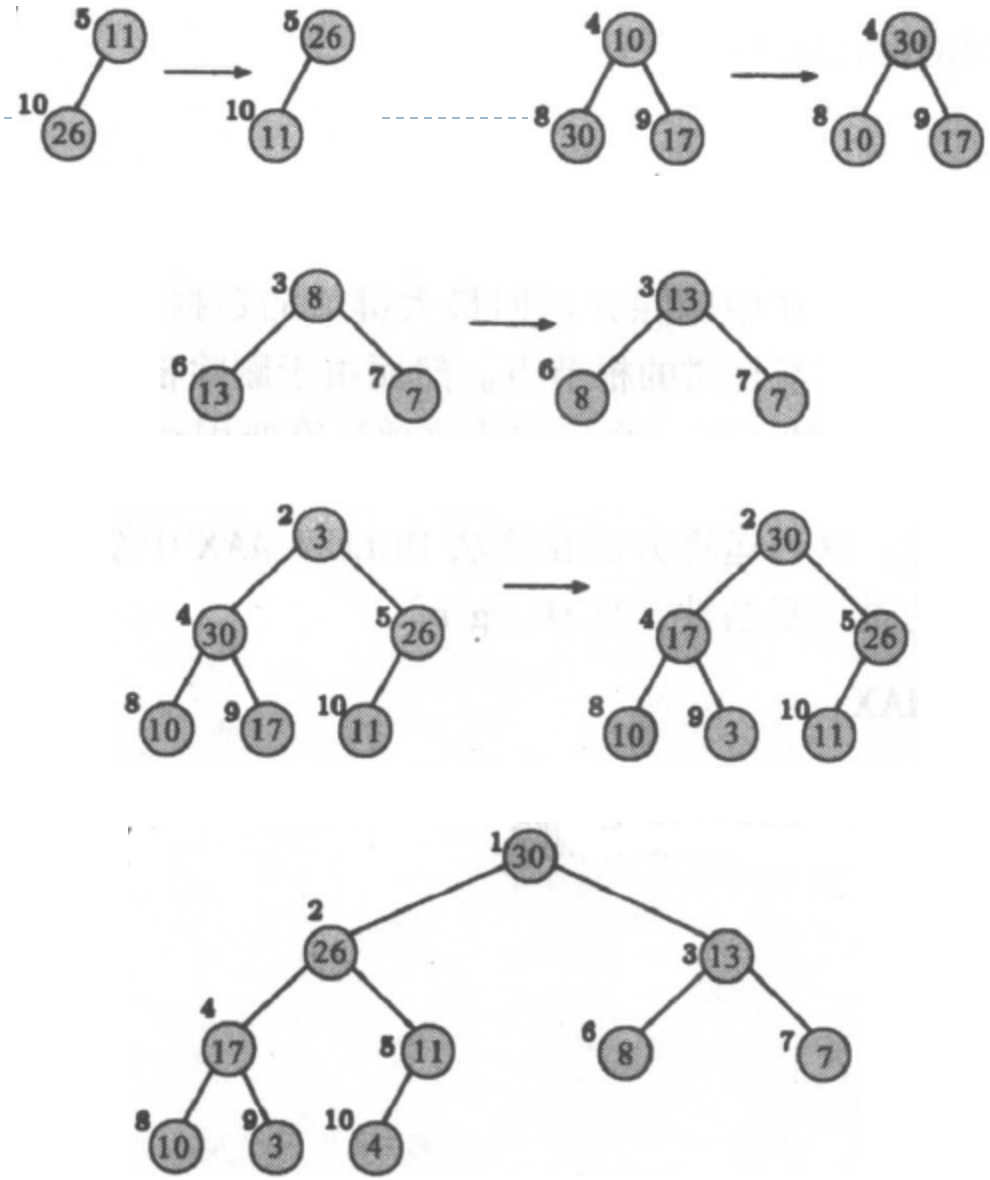
方法2：

MAKEHEAP (创建堆)
输入：数组A[1…n]
输出：将A[1…n]转换成堆
1. for i←⌊n/2⌋downto 1
2. Sift-down(A,i) {使以A[i]为根节点的子树调整成为堆，故调用down过程}
3. end for

例：给定数组A[1…10] = {4, 3, 8, 10, 11, 13, 7, 30, 17, 26}

初始情况




5 11
10 26
5 26
10 11
4 10
8 30
9 17
4 30
8 10
9 17
3 8
6 13
7 7
3 13
6 8
7 7
2 3
4 30
5 26
8 10
9 17
10 11
2 30
4 17
5 26
8 10
9 3
10 11
1 30
2 26
3 13
4 17
5 11
6 8
7 7
8 10
9 3
10 4

复杂度分析

- 树高k=⌊logn⌋，第i层正好2^i个节点, $0 \le i < k$,(不含最深的叶子节点层), 每个节点的down过程最多执行k-i次,故down过程执行次数上限为

$$\sum_{i=0}^{k-1}(k-i)2^i = \sum_{j=k}^{1} j2^{k-j} \quad (令 k-i=j)$$

$$= 2^k \sum_{j=1}^{k} j2^{-j} = 2^k \Theta(1)$$

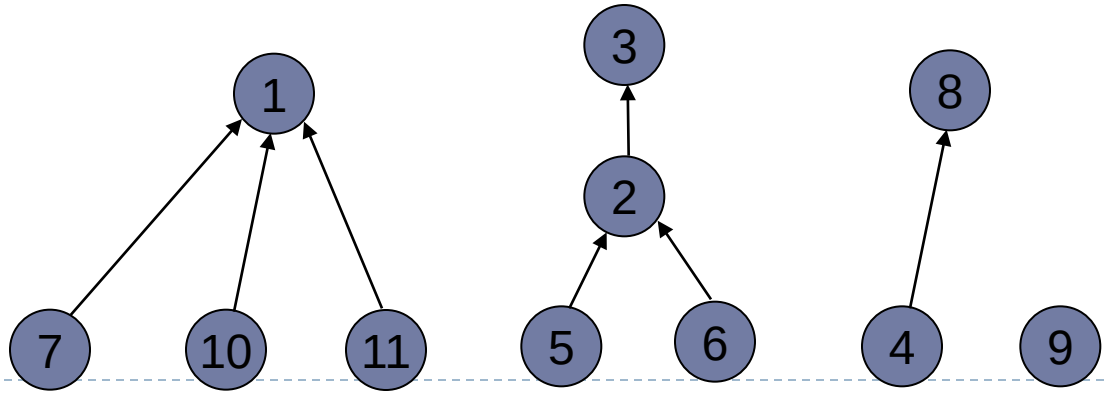
$$\le n \cdot \Theta(1) < 2n$$

- 因为 $T(2n) = T(n) + \Theta(n)$ ，时间复杂度为O(n).

不相交集(Disjoint Sets)

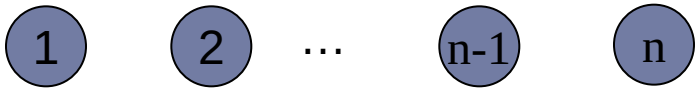
- 假设有n个元素，被分成若干个集合。例如S={1,2,…11}分成4个子集1:{1,7,10,11}，3:{2,3,5,6}，8:{4,8}，9:{9}并分别命名。
- 事实上，每个子集可以用树表示，除根节点外，每个节点都有指针指向父节点。上例可以用树表示为:

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



- 假如要执行如下计算任务：
  - FIND(x): 寻找包含元素x的集合的名字
  - UNION(x,y): 将包含元素x和y的两个集合合并，重命名。
- 记root(x)为包含元素x的树的根，则FIND(x)返回root(x).
- 执行合并UNION(x, y)时
  - 首先依据x找到root(x),记为u。u=root(x)
  - 依据y找到root(y)，记为v。v=root(y)
  - 然后，将u指向v。
- 优点：简单明了
- 缺点：多次合并后，树高度可能很大，查找困难。

例：初始状态：{1},{2},…,{n}



执行合并序列：UNION(1,2),UNION(2,3),…UNION(n-1,n).我们得到的结果是：

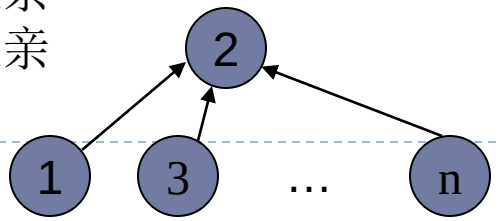
执行查找序列：FIND(1), FIND(2),…, FIND(n).需要比较的次数是：

$$n+(n-1)+\cdots+2+1=\frac{n(n+1)}{2}$$

目标：降低树的高度。措施：Rank Heuristic。

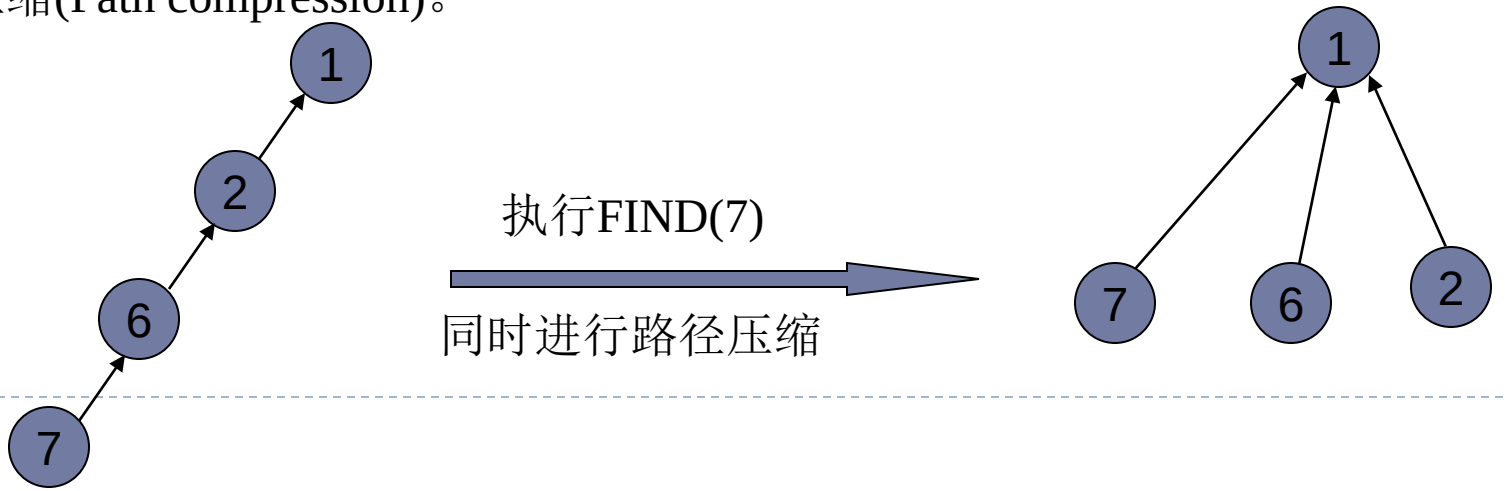
1.给每个树的根节点定义一个秩(rank)，表示该树的高度。
2.在执行UNION(x, y)，首先找到u=root(x)，v=root(y)。
3.然后比较rank(u)和rank(v)

若rank(u) = rank(v)，则使u指向v，v成为u的父亲，同时rank(v)+1
若rank(u) < rank(v)，则使u指向v，v成为u的父亲
若rank(u) > rank(v)，则使v指向u，u成为v的父亲



Algorithm: UNION
输入：两个元素x, y.
输出：将包含x, y的两棵树合并
1. u←FIND(x); v←FIND(y)
2. if rank(u)≤rank(v) then　　// Rank Heuristic
3.　　p(u) ←v
4.　　if rank(u) = rank(v) then rank(v) = rank(v) + 1
5. else
6.　　p(v) ←u
7. end if

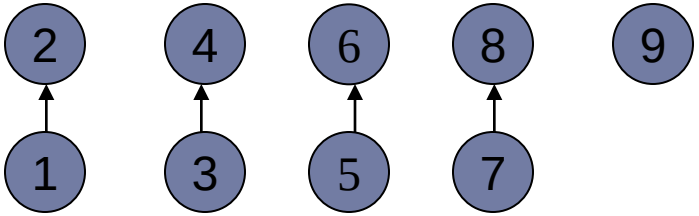
目标：进一步提高FIND的操作的性能。措施：在执行FIND操作时，同时进行路径压缩(Path compression)。



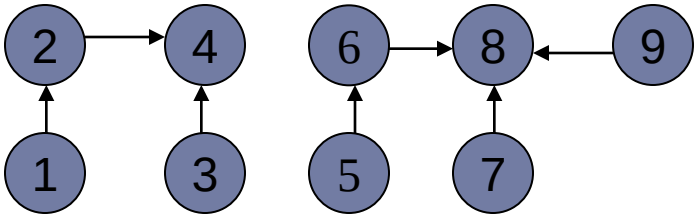
Algorithm: FIND
输入：节点x
输出：root(x)和路径压缩后的树
1. y←x
2. while p(y)≠null {寻找包含x的树的根}
3. y←p(y)
4. end while
5. root←y; y←x {重新赋值为原来的节点x}
6. while p(y) ≠null {执行路径压缩}
7. w←p(y) {父节点暂存为w}
8. p(y) ←root {该路径上的节点直接指向根节点}
9. y←w {继续下一步压缩}
10. end while
11. return root

例：初始状态：{1},{2},…,{9}

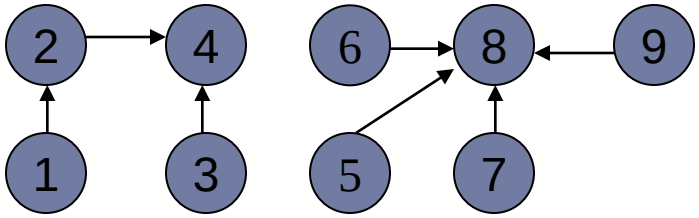
执行合并序列：UNION(1,2),UNION(3,4),UNION(5,6) ,UNION(7,8),得到的结果是：



继续执行合并序列：UNION(2,4),UNION(8,9),UNION(6,8),得到的结果是：



继续执行：FIND(5)得到的结果是：



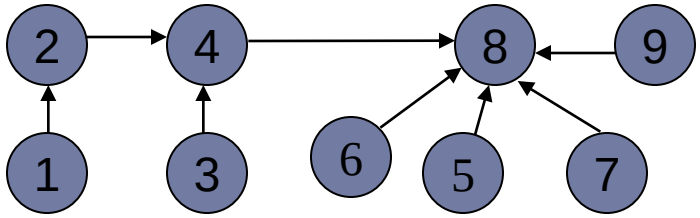
继续执行UNION(4,8)呢?

注意：路径压缩时，秩不会改变。

即执行FIND操作后，根节点的秩有可能大于树的高度。

节点的秩变成了节点高度的上界

继续执行：UNION(4,8)得到的结果是：



继续执行：FIND(1)得到的结果是：