算法概论

第四讲：选择和检索

薛健

Last Modified: 2019.12.15

主要内容

1 选择算法

1.1 选择问题

选择问题

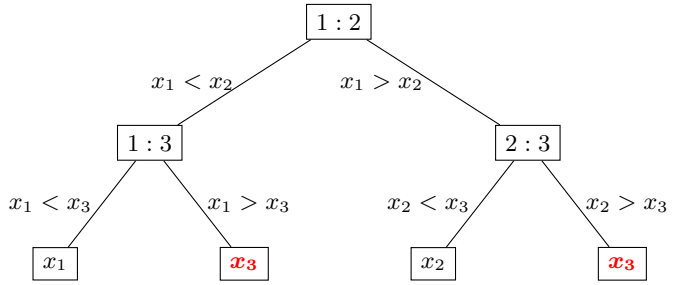
- **选择问题** (Selection Problem)：在一个有 n 个元素的序列中找到第 k 小的元素 $(1 \le k \le n)$，即：如果该序列按从小到大排序，则第 k 个元素就是所需要的结果
- 基本操作：比较。
  - 以比较为基本操作的排序问题复杂度下界是 $\Theta(n \log n)$
  - 选择问题的复杂度下界是 $\Theta(n \log n)$ 或更低？
  - 决策树分析？
- 常见选择问题：
  - 选择最大元素 (the **maximum**) 或最小元素 (the **minimum**)
  - 同时选择最大和最小元素
  - 选择中间元素（中位数，the **median**） $(k = \lceil n/2 \rceil)$

用决策树分析：选择问题决策树

- 外结点（叶结点）：选择结果（至少为 n）
- 内结点：标记两个元素的一次比较（$n_2 = n_0 - 1 \ge n - 1$）

因此决策树至少包含 $2n-1$ 个结点，其高度 $h \ge \lceil \lg(2n-1+1) \rceil - 1 = \lceil \lg n + 1 \rceil - 1 = \lceil \lg n \rceil$，但这个下界不够准确。

我们已经知道找最大元素至少需要 $n-1$ 次比较（反证法证明），哪里出问题了呢？在选择问题的决策树中，某些输出将出现在超过一个叶子结点上，如 $n = 3$ 选择最小元素的决策树：



对手论证法

- **猜数字的游戏**：心中选定一个数字让对方猜，对方可按如下方式提问：“这个数字比 * 大（小）吗？”，可提问多次，每次提问后，必须给出回应：“是”或“否”
  - 如何让对方在尽可能多次的提问后仍然猜不到所选定的数字？

  - 换一种思维方式：在最开始并不真正选定具体数字，而是随对手的问题不断调整 (*cheating*？)
  - 必须保证最后结果与对手提出的所有问题都不矛盾

- **对手论证法** (Adversary Arguments)：
  - 对于解决某类问题的算法，有一个假想的对手在千方百计阻挠其得到最终结果
  - 在算法作出每一次决策后，对手总给出不利于算法的结果
  - 对于对手的唯一约束是：他所给出的所有决策结果必须是自洽的
  - 该方法可以用来分析选择问题的复杂度下界

猜数字对手策略：假定待猜测数字范围在 $[0, 100]$，若猜的人问："这个数比 10 大吗?"，对手回答为"是"，因为这样剩下的可能性更多。

对手论证法的本质就是沿最不利于算法得到结果的执行路径（或方向）一步步确定最坏情况下的输入（对应问题复杂度下界）。

1.2 查找最大和最小元素

查找最大和最小元素

- 如果我们找最大元素后再找最小元素，比较次数为 $(n-1)+(n-2)=2n-3$，这显然不是最佳算法，在找最大元素时有一些结果可以被后面找最小元素的过程所共享
- 一个更快的算法：
  1. 比较每两个元素的大小（$n/2$ 次比较）
  2. 在得到的 $n/2$ 个较大元素中找出最大元素（$n/2-1$ 次比较）
  3. 在得到的 $n/2$ 个较小元素中找出最小元素（$n/2-1$ 次比较）
- 加上对 n 是奇数的考虑，总共的比较次数为 $\lceil 3n/2 \rceil - 2$ 次比较

Theorem 1.1. 任何在 n 个元素中查找最大和最小元素的算法在最坏情况下至少需要 $3n/2-2$ 次比较操作.

复杂度下界分析

- 一些约定：
  - 将比较看作竞赛：$x > y$ 的结果为：x 赢 y 输
  - 元素状态：
$$\begin{cases} W & \text{至少赢了一次且从未输过} \\ L & \text{至少输了一次且从未赢过} \\ WL & \text{输赢各至少一次} \\ N & \text{还未参与比较} \end{cases}$$

- 查找最大和最小元素的对手策略：

比较前状态	对手回应	新状态	有效信息数
N, N	$x > y$	W, L	2
W, N 或 WL, N	$x > y$	W, L 或 WL, L	1
L, N	$x < y$	L, W	1
W, W	$x > y$	W, WL	1
L, L	$x > y$	WL, L	1
W, L 或 WL, L 或 W, WL	$x > y$	无改变	0
WL, WL	实际情况	无改变	0

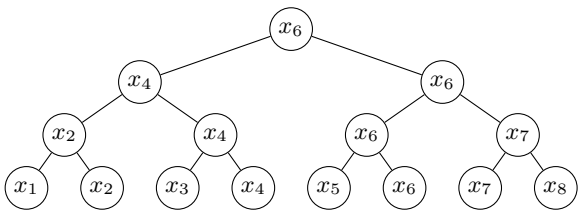
复杂度下界分析 (cont.)

- 要找到最大和最小元素，需要知道除最大元素以外的每个元素至少输掉了某次比较，而除最小元素以外的每个元素至少赢得了某次比较，即：至少需要 $2n-2$ 条有效信息才能作出最终决定
- 从算法的角度讲，每次比较最多能从对手那里得到 2 条有效信息，而这样的比较最多能进行 $n/2$ 次，共得到 n 条有效信息
- 对于余下的 $n-2$ 条信息，由于每次比较最多只能得到 1 条有效信息，所以至少需要 $n-2$ 次比较
- 因此，要得到最终结果，至少需要 $n/2+n-2=3n/2-2$ 次比较

1.3 查找第二大元素

查找第二大元素

- 如果用找最大元素的方法找第二大元素，则需要 $(n-1)+(n-2)=2n-3$ 次比较
- 更快的方法：**联赛方法** (tournament method)
  - 每两个元素进行比较，其赢家进入下一轮，直到决出最终的胜利者
  - 若元素数为奇数，则剩余一个元素直接进入下一轮
  - 按照上述规则，可建立一棵二叉树，其根结点为最大元素，比较次数为 $n-1$，而只有跟最大元素直接比较过的元素才有可能是第二大



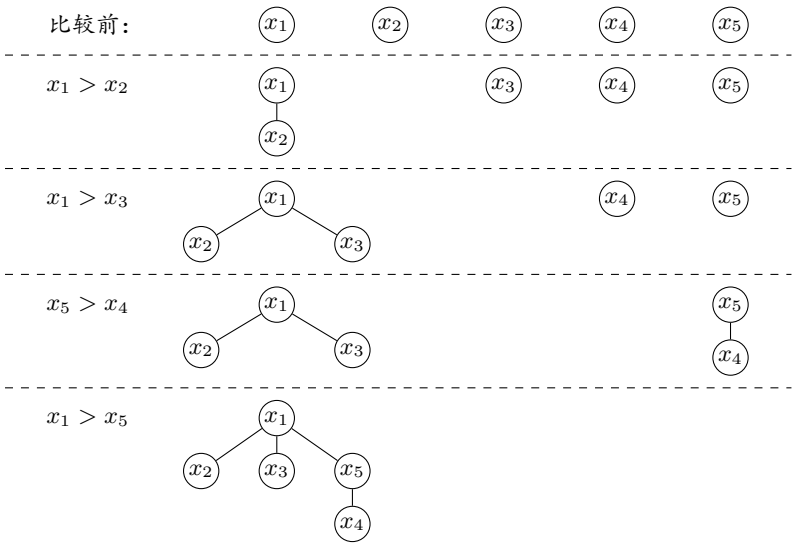
  - 与最大元素直接比较的路径上的其他元素中最大的那个即为第二大元素，比较次数 $\lceil \lg n \rceil - 1$，总的比较次数为 $\boldsymbol{n + \lceil \lg n \rceil - 2}$

复杂度下界分析

- 我们已经知道：第二大元素肯定在比较至少输过一次的元素中，而要区分出所有的输家，至少需要 $n-1$ 次比较
- 如果最大元素参与了其中 $\lceil \lg n \rceil$ 次比较，则为了得到第二大元素，其中 $\lceil \lg n \rceil - 1$ 个元素将再次在与第二大元素的比较中落败，所以至少需要 $n + \lceil \lg n \rceil - 2$ 次比较
- 因此，我们只要找到一种对手策略能够迫使算法将最大元素至少与 $\lceil \lg n \rceil$ 个不同元素进行比较：
  - 为每个元素指定一个权值 $w(x)$，初始化为 1，当算法比较 x 和 y 时根据对手回应调整权值

比较前权值	对手回应	更新权值
$w(x) > w(y)$	$x > y$	$w(x) \leftarrow \text{prior}(w(x)+w(y)); w(y) \leftarrow 0$
$w(x) = w(y) > 0$	$x > y$	$w(x) \leftarrow \text{prior}(w(x)+w(y)); w(y) \leftarrow 0$
$w(x) < w(y)$	$x < y$	$w(y) \leftarrow \text{prior}(w(x)+w(y)); w(x) \leftarrow 0$
$w(x) = w(y) = 0$	实际情况	无改变

例子



比较元素对	权值	胜者	更新后权值	元素键值
x_1, x_2	$w(x_1) = w(x_2)$	x_1	$2,0,1,1,1$	$20,10,*,*,*$
x_1, x_3	$w(x_1) > w(x_3)$	x_1	$3,0,0,1,1$	$20,10,15,*,*$
x_5, x_4	$w(x_5) = w(x_4)$	x_5	$3,0,0,0,2$	$20,10,15,30,40$
x_1, x_5	$w(x_1) > w(x_5)$	x_1	$5,0,0,0,0$	$41,10,15,30,40$

复杂度下界分析

- 上述对手策略是否满足要求？
  1. 一个元素输掉一次比较当且仅当其权值为 0
  2. 在前三种情况中，被选为胜者的元素有非 0 权值，这表示该元素在前面的比较中还未输过，因此可以给他赋予任意大的键值从而保证与前面作出的回应不产生矛盾
  3. 由权值的更新方式可知所有元素的权值和始终等于 n
  4. 当算法停止时，仅有一个元素可有非 0 权值，否则将至少有两个元素从未输过一次比较，这时对手可以合理选择这两个元素的键值使算法所找出的第二大元素不正确
- 假设当算法终止时，x 是唯一一个键值非 0 的元素，则根据上述结论有 $x = \mathsf{max}, w(x) = n$
- 设 x 第 k 次赢得比较后权值 $w(x) = w_k$，则根据权值更新规则有 $w_k \leq 2w_{k-1}$
- 设 K 为 x 赢得的比较次数，则有 $n = w_K \leq 2^K w_0 = 2^K \Rightarrow K \geq \lg n \Rightarrow \boldsymbol{K \geq \lceil \lg n \rceil}$

1.4 中位数问题

查找中位数

- **中位数问题**：查找包含 n 个元素序列的中位数（median，即排序后处于 $\lceil n/2 \rceil$ 位置处的元素）
- 分治法（`QuickSort` 的分治法策略）：
  - 将元素分为两组，使其中一组中的元素小于另一组中的元素
  - 在包含元素较多的一组中继续查找
- 问题：递归调用不再是查找中位数（原始序列的中位数并不也是子序列的中位数）
- 推广 $\Rightarrow$ 一般**选择问题** (Selection Problem)：查找序列中第 k 小的元素（即从小到大排序后的第 k 个元素）
- 先排序后选择 $(O(n \log n))$？
- 有更好的算法 $(O(n))$！

算法描述

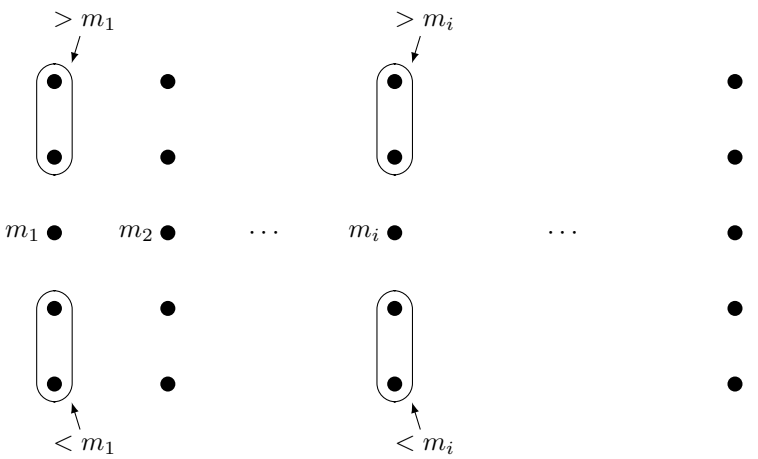
Algorithm `Select`(Set S, int k)

1 if $|S| \leq 5$ then return S 中第 k 小元素的直接查找结果;
2 将原集合元素 5 个一组分成 $|S|/5$ 组，取每组的中位数放入集合 M 中;
3 $m^* \leftarrow$ `Select`$(M, \lceil |M|/2 \rceil)$;
4 根据 m^* 将 S 分成 4 个子集：

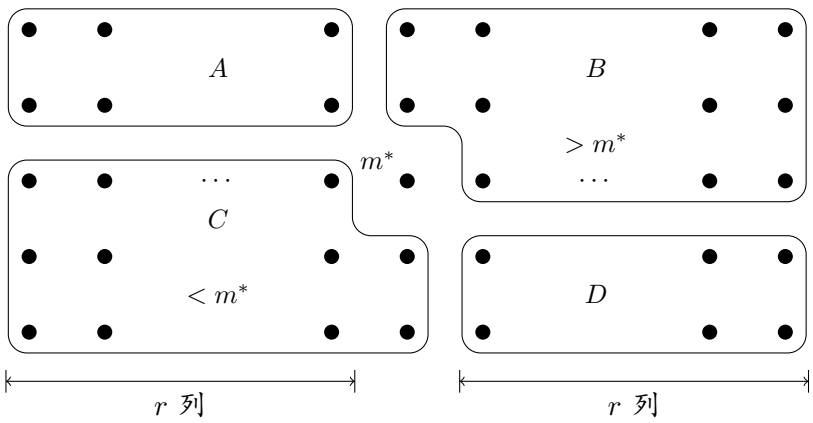
A	中位数小于 m^* 的 5 元组中大于各自中位数的元素
B	中位数大于 m^* 的 5 元组中大于等于各自中位数的元素（都大于 m^*）
C	中位数小于 m^* 的 5 元组中小于等于各自中位数的元素（都小于 m^*）
D	中位数大于 m^* 的 5 元组中小于各自中位数的元素

;
5 将 A 和 D 中的元素与 m^* 比较，其中小于 m^* 的元素与 C 合并成 S_1，大于 m^* 的元素与 B 合并成 S_2;
6 if $k = |S_1| + 1$ then return m^*;
7 else if $k \leq |S_1|$ then return `Select`(S_1, k);
8 else return `Select`$(S_2, k - |S_1| - 1)$;

分组求中位数



求中位数的中位数



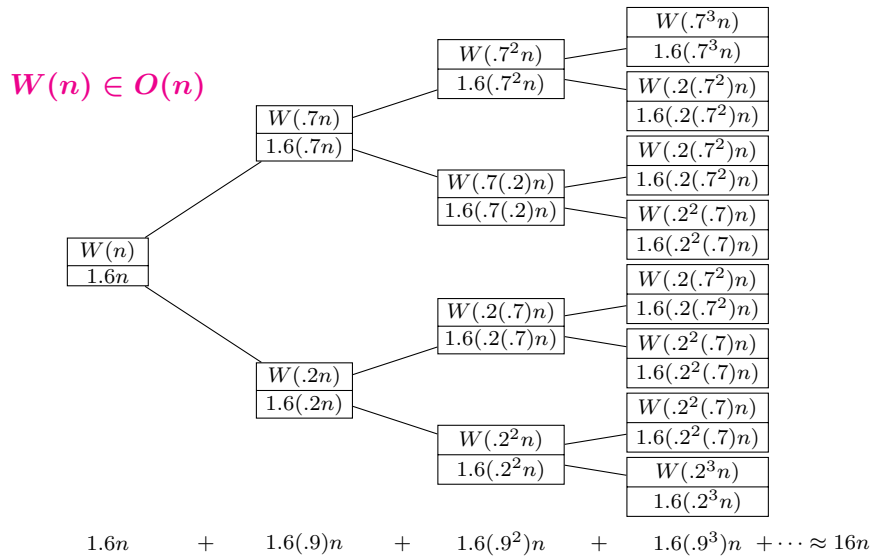
选择算法复杂度分析

- 为便于分析，假设 $n = 5(2r+1)$，忽略递归调用不满足该式所造成的影响，则可逐步分析上述算法如下：
  1. 找所有 5 元组的中位数：$6(n/5)$ 次比较 ($Why?$)
  2. 递归查找中位数集合的中位数：$W(n/5)$ 次比较
  3. 子集 A、D 中的元素与 m^* 比较：$4r$ 次比较
  4. 递归调用 `Select`，在最坏情况下，A、D 中的元素都大于或都小于 m^*：$W(7r+2)$ 次比较
- 由 $n = 5(2r+1)$ 可知 $r \approx 0.1n$，所以有：

$$\begin{aligned} W(n) &\le 1.2n + W(0.2n) + 0.4n + W(0.7n) \\ &= 1.6n + W(0.2n) + W(0.7n) \end{aligned}$$

- 主定理不适用，用递归树求解

选择算法递归树



第 k 层的非递归开销和：

$$\begin{aligned} T(k) &= 1.6(C_k^0 0.2^k + C_k^1 0.2^{k-1} \cdot 0.7 + C_k^2 0.2^{k-2} \cdot 0.7^2 + \cdots + C_k^k 0.7^k)n \\ &= 1.6(0.2+0.7)^k n = 1.6(0.9)^k n \end{aligned}$$

可用归纳法证明。

归纳步骤中递归树每增加一层一项裂为两项：$x \to 0.2x + 0.7x$

$$\begin{aligned} T(k+1) =& 1.6(C_k^0(0.2^{k+1} + 0.2^k \cdot 0.7) + C_k^1(0.2^k \cdot 0.7 + 0.2^{k-1} \cdot 0.7^2) \\ &+ \cdots + C_k^k(0.2 \cdot 0.7^k + 0.7^{k+1}))n \\ =& 1.6(C_k^0 0.2^{k+1} + (C_k^0 + C_k^1)0.2^k \cdot 0.7 + (C_k^1 + C_k^2)0.2^{k-1} \cdot 0.7^2 \\ &+ \cdots + (C_k^{k-1} + C_k^k)0.2 \cdot 0.7^k + C_k^k 0.7^{k+1})n \end{aligned}$$

注意到：

$$C_k^0 = C_{k+1}^0 = C_k^k = C_{k+1}^{k+1} = 1$$
$$C_k^i + C_k^{i+1} = C_{k+1}^{i+1}$$

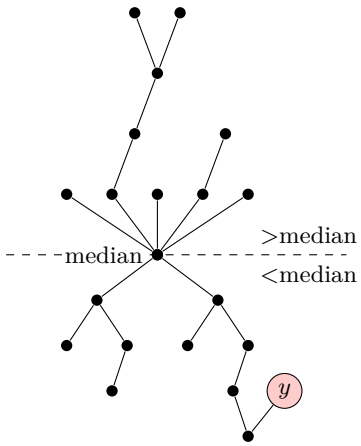
因此：

$$\begin{aligned} T(k+1) =& 1.6(C_{k+1}^0 0.2^{k+1} + C_{k+1}^1 0.2^k \cdot 0.7 + C_{k+1}^2 0.2^{k-1} \cdot 0.7^2 \\ &+ \cdots + C_{k+1}^k 0.2 \cdot 0.7^k + C_{k+1}^{k+1} 0.7^{k+1}) \\ =& 1.6(0.2+0.7)^{k+1} n = 1.6(0.9)^{k+1} n \end{aligned}$$

进一步思考：第一步分组时能否分成 3 个元素一组？7 个或更多怎么样？

中位数问题的复杂度下界

- 既然是复杂度下界分析，不妨设集合中元素各异，元素个数为奇数
- 一个算法要想找到中位数，必须知道其他每个元素和中位数的大小关系；否则，假设某元素 y 与中位数大小关系未知，则在对手论证法中，算法对手可以改变元素 y 的键值，使其落在中位数分界线的另一边，使得算法所选的中位数不再是真正的中位数
- 根据上述分析，一个算法要想找到正确的中位数，至少需要 $\boldsymbol{n-1}$ 次比较
- $n-1$ 这个下界是否足够准确？



中位数问题的复杂度下界 (cont.)

- **关键比较**：元素 x 与 $y \geq$ median 的第一次比较，且结果为 $x > y$；或与 $y \leq$ median 的第一次比较，且结果为 $x < y$。从关键比较可以知道 x 与中位数的大小关系
- 可以设计一种对手策略，迫使算法进行非关键比较：
  - 开始比较前，选定一个键值作为中位数键值，但不指定具体元素作为中位数

  - 每次比较给首次参与比较的元素赋值，尽量使元素位于中位数两侧
  - 约束条件：比中位数大的元素总数不得超过 $(n-1)/2$；同样，比中位数小的元素总数也不得超过 $(n-1)/2$
  - 元素状态：L —被赋予比中位数大的值；S —被赋予比中位数小的值；N —未参与比较
  - 元素赋值规则：

比较前状态	元素赋值规则	比较后状态
N, N	一个大于中位数，一个小于中位数	L, S
L, N 或 N, L	未参与比较的元素赋予小于中位数的值	L, S
S, N 或 N, S	未参与比较的元素赋予大于中位数的值	L, S

中位数问题的复杂度下界 (cont.)

- 对手策略（续）
  - 赋值规则表中的比较都是非关键比较
  - 当大于中位数的元素总数或小于中位数的元素总数达到 $(n-1)/2$ 后，上述元素赋值规则不再适用，这时对手策略将按照实际情况赋值，直到剩下一个元素，该元素即为中位数
- 显然，上述对手策略可迫使任何查找中位数的算法至少执行 $(n-1)/2$ 次非关键比较，而关键比较次数至少为 $n-1$，因此，我们可以得出结论：

  Theorem 1.2. 任何以比较为关键操作选取 n（n 为奇数）个元素中的中位数的算法至少需要 $3n/2-3/2$ 次比较

- $3n/2-3/2$ 是最好的下界吗？

关于中位数问题更深入的探讨

- 中位数选择算法：
  - [Manuel Blum, et al., 1973] 提出了比较次数为 $5.43n$ 的算法
  - [A. Schönhage, et al., 1976] 提出了比较次数为 $3n+o(n)$ 的算法
  - [Dorit Dor and Uri Zwick, 1999] 提出了比较次数为 $2.95n+o(n)$ 的算法
- 中位数问题的复杂度下界：
  - [Vaughan R. Pratt and Foong Frances Yao, 1973]：$1.75n-\log n$
  - [C. K. Yap, 1976]：$\frac{38}{21}n+O(1)\approx 1.81n$ 和 $\frac{79}{43}n+O(1)\approx 1.83n$
  - [Samuel W. Bent and John W. John, 1985]：$2n+o(n)$
  - [Dorit Dor, et al., 1996]：$(2+\epsilon)n+o(n)$
  - [Dorit Dor, et al., 2001]：$2.01227n+o(n)$
- 参考文献：

[1] Dorit Dor, Johan Håstad, Staffan Ulfberg and Uri Zwick. *On Lower Bounds for Selecting the Median.* SIAM journal on discrete mathematics, 14(3), pp.299–311, 2001.

利用对手论证法来设计算法

- 前面利用对手论证法来分析问题的复杂度下界
  - 在对算法每一步操作的回应中尽可能少地透露有用信息
- 其内在逻辑是：
  - **问题复杂度**（下界）：$\forall$ 算法，$\exists I$ 使算法开销至少为 $W(n)$
  - **对手论证法**：$\forall$ 算法，输入 I 不定，使用竞争的方式找到某个特定的输入 I_{w} 使得算法开销至少为 $W(n)$
  - “对手”即对应于“最坏情况下的输入”
- 同样可以利用对手论证法来设计算法，使复杂度尽可能低：
  - 每次基本操作尽量使对手回答“是”或“否”所透露的信息一样多（某种平衡）
  - 决策树 $\Longrightarrow$ 平衡二叉树（高度尽可能低）
  - 归并排序、查找最小和最大元素、查找第二大元素等算法都很好地运用了这样的策略

Exercise (5). 试设计比较策略：用 6 次比较在 5 个元素中找到中位数；用 7 次比较完成 5 个元素的排序. *deadline: 2019.12.22*

2 动态集合和搜索

2.1 基本概念

动态集合

- **动态集合** (Dynamic Sets)：元素（值、结构、个数等）随算法运行而不断改变的集合，例如：
  - 初始状态为空集，在后续运算过程中不断有新的元素添加到集合中，并且最终集合中包含多少元素并不确定
  - 初始状态为一个大的集合，在运算过程中不断从中删除元素（通常以空集作为算法停止的状态）
  - 或者在运算过程中同时添加和删除元素
- 最常用也是最基本的集合动态增长技术是**数组加倍** (Array Doubling)：在往集合中添加元素时，若用于存储的数组空间不足，则成倍增加数组大小（例：STL 中的 vector 类）

Procedure `ArrayDouble`(Set s)

1 $newLength \leftarrow 2 * s.elements.length$;
2 $newElements \leftarrow$ new Object[$newLength$];
3 将 $s.elements$ 中的元素拷贝到 $newElements$ 中;
4 $s.elements \leftarrow newElements$;

动态集合的基本操作：

- 修改操作：Insert, Delete
- 查询操作：Search, Minimum, Maximum, Successor, Predecessor

分摊时间分析

- 上面提到的 `ArrayDouble` 乍一看非常耗时，复杂度达到 $\Theta(n)$，而其最终要完成的操作仅仅是往集合中插入一个新的元素；但从集合插入 n 个元素的总执行时间来看，其复杂度并不高，也是 $\Theta(n)$
- 这样的效果实际上可以看作数组加倍的额外开销被分摊到了整个操作序列的每个插入操作上，从而使整体的时间复杂度并未提高
- 动态集合上基本操作时间开销的不确定性给相关算法的分析和设计带来了一定的难度
- **分摊时间分析** (Amortized Time Analysis)：分析动态数据结构基本操作执行时间的方法

$$\underset{amortized\ cost}{\text{分摊开销}} = \underset{actual\ cost}{\text{实际开销}} + \underset{accounting\ cost}{\text{账户开销}}$$

- 其关键是构造账户开销，使其达到以下目标：
  1. 从动态集合创建起，任意合法的操作序列，其账户开销总和非负
  2. 不管实际开销在单个操作之间波动多大，每个操作的分摊开销应基本保持一致

正如在银行开户存款，在条件好的时候存款以防万一，在偶尔条件不好的时候提款应付困境。只要还没销户（破产），账户内的存款应为非负。正常情况实际开销和账户开销为正；遇到实际开销较大的操作，账户开销为负（提款）以抵消一部分实际开销。

分摊时间分析的合理性在于：

$$\begin{aligned} \sum \text{分摊开销} &= \sum \text{实际开销} + \sum \text{账户开销} \\ \Rightarrow \sum \text{实际开销} &= \sum \text{分摊开销} - \sum \text{账户开销} \end{aligned}$$

而

$$\sum \text{账户开销} \geq 0$$

所以

$$\sum \text{实际开销} \leq \sum \text{分摊开销}$$

可以看出，分摊时间分析实际上是求实际开销总和的一个上界，因为直接求实际开销总和由于基本操作开销的不确定性而变得困难，所以转而求易于求和的分摊开销总和（每个操作的分摊开销基本保持一致）。

分摊时间分析可以用来证明在一系列操作中，通过对所有操作求平均之后，即使其中单一操作具有较大的开销，平均开销还是很小的。分摊时间分析与平均情况分析的不同之处在于它不涉及到概率；分摊时间分析保证在最坏情况下，每个操作具有平均性能。

例子

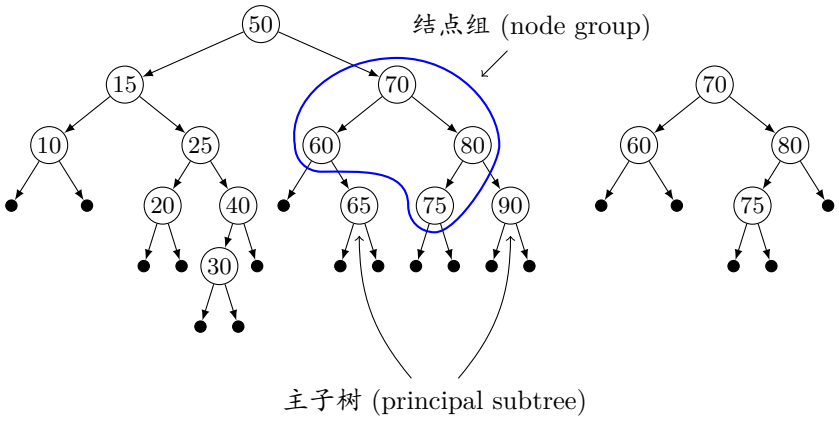
- 在 Stack 数据结构中用数组加倍的方式扩展存储空间
- pop 操作的实际开销为 1；push 操作的实际开销：
  - 不需要扩展数组空间时为 1
  - 需要扩展数组空间时为 $1 + nt \in \Theta(n)$（t 是拷贝一个元素的开销）
- 构造账户开销：
  1. 不需要数组加倍时 push 操作的账户开销为 $2t$
  2. 数组空间从 n 加倍到 $2n$ 时 push 操作的账户开销为 $-nt + 2t$
  3. pop 操作的账户开销为 0
- Stack 空间加倍的操作发生在元素数达到 $N, 2N, 4N, 8N, \cdots$ 时
  - 当达到 N 时，账户中储存的开销达到 $2Nt$，由于数组加倍而降到 $Nt + 2t$
  - 当达到 $2N$ 时，储存开销达到 $3Nt$，数组加倍又使其降到 $Nt + 2t$
  - 以此类推，当储存的开销达到 $Nt + 2t$ 后便不再低于它，符合账户开销总和非负的要求
- push 操作的分摊开销为 $1 + 2t \in \Theta(1)$

注意：账户开销中的 2 是一个关键系数，可以试一下，任何小于 2 的系数均会导致某些时候账户开销和为负。

2.2 红黑树

扩展二叉树

将空子树作为外结点补充到二叉树中，使其成为 2-tree

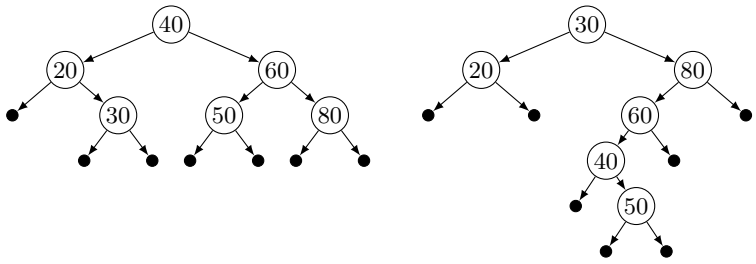


- **结点组** (node group)：二叉树中任何有边连接在一起的内结点组合（一棵子树）
- 结点组的**主子树** (principal subtree)：根结点的父结点在结点组中而其本身的任何结点都不在结点组中

二叉搜索树

Definition 2.1 (二叉搜索树 (Binary Search Tree))**.** 二叉搜索树是满足如下特性的二叉树：每个结点元素的键值大于其左子树的所有结点元素而小于等于其右子树的所有结点元素

- 例：



- 用中序 (inorder) 遍历二叉搜索树可得到从小到大排序的元素序列
- 包含相同元素的二叉搜索树平衡度 (degrees of balance) 可以差别很大

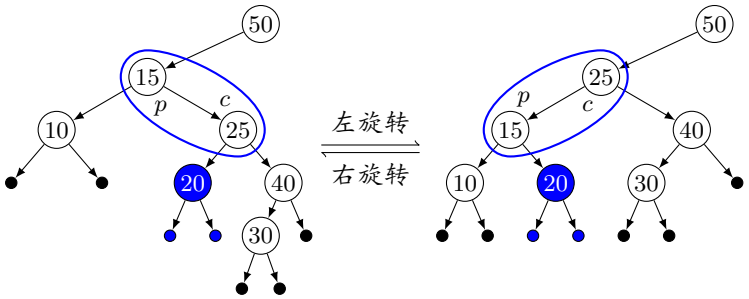
搜索算法

Algorithm `BSTSearch`(BinaryTree bst, Element K)

```
1 Element r ← null;
2 if bst ≠ null then
3 |   Element root ← Root(bst);
4 |   if K = root then  r ← root ;
5 |   else if K < root then  r ← BSTSearch(LeftSubtree(bst), K) ;
6 |   else  r ← BSTSearch(RightSubtree(bst), K) ;
7 end
8 return r;
```

- 比较次数与二叉树高度相当
- 如果二叉搜索树可以有任意结构，则最坏情况搜索时间复杂度为 $\Theta(n)$
- 如果二叉搜索树尽可能达到平衡，则最坏情况复杂度可降到 $\lg n \in \Theta(\log n)$

二叉树的旋转操作



- 旋转操作在两个相邻结点之间进行（父结点 p 和其子结点 c），涉及 3 棵主子树，如图所示左旋转过程可描述为：
  1. 改变 p 和 c 之间边的方向，p 成为 c 的左子结点
  2. 原来 p 的父结点成为 c 的父结点
  3. 原来 c 的左子树（即中间那棵主子树）变为 p 的右子树
- 左、右旋转互为反变换

经典二叉搜索树

- AVL 树：左右子树高度差绝对值 ≤ 1，发表于 1962 年的文章 “An Algorimth for the Organization of Information”，其名称来源于发明者姓名首字母（G. M. **A**delson-**V**elsky 和 E. M. **L**andis）。
- **红黑树**：1972 年由 Rudolf Bayer 发明，当时被称为对称二叉 B 树（symmetric binary B-trees），后来在 1978 年 Leonidas J. Guibas 和 Robert Sedgewick 合作发表的论文 “A Dichromatic Framework for Balanced Trees” 中被修改为如今的 “红黑树”，之所以选择 “红色” 据说是因为红色是当时作者所在单位 (Xerox PARC) 的激光打印机所能打出的最好看的颜色。红黑树的应用非常广泛，如 C++ STL 中的很多容器 (set, multiset, map, multimap) 都应用了红黑树或其变体。
- AVL 树 vs. 红黑树
  - AVL 树是严格平衡二叉树，在增、删结点时旋转操作次数多于红黑树；
  - 红黑树用非严格的平衡换取增、删操作的效率提升；
  - 在具体应用时，若搜索次数远大于增、删结点次数，则选择 AVL 树；若搜索、增、删次数相当，则选红黑树。

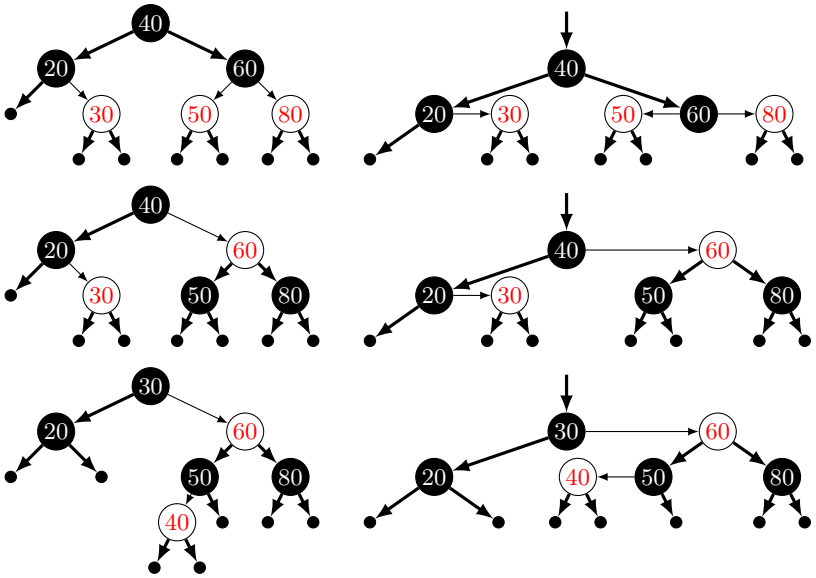
红黑树定义

Definition 2.2 (红黑树 (Red-Black Tree，RB Tree))**.** 将一棵二叉树的每个结点涂成红色或黑色，终点是黑色结点的边称为**黑边** (black edge)，路径的**黑色长度** (black length)定义为该路径上的黑边数目，结点的**黑色深度** (black depth)定义为从根结点到该结点路径的黑色长度，从给定结点到某一外结点路径称为该结点的**外路径** (external path)，则一棵二叉树是红黑树，当且仅当：

1. 任何红色结点无红色孩子结点
2. 给定结点 u，其所有外路径的黑色长度均相等，其值称为结点 u 的**黑色高度** (black height)
3. 根结点和所有外结点是黑色结点

根结点为红色且满足上述条件的二叉树称为**近似红黑树** (almost-red-black tree, ARB tree).

例子



红黑树递归定义

Definition 2.3 (RB_h 树和 ARB_h 树)**.** 顶点着色为红或黑且外结点为黑的二叉树按如下定义构成 RB_h 树或 ARB_h 树：

1. 外结点是 RB_0 树
2. 对 $h \geq 1$，根结点为红色且左、右子树是 RB_{h-1} 树的二叉树是 ARB_h 树
3. 对 $h \geq 1$，根结点为黑色且左、右子树是 RB_{h-1} 树或 ARB_h 树的二叉树是 RB_h 树

Lemma 2.4. RB_h 树和 ARB_h 树的黑色高度为 h

一些结论

Lemma 2.5. 若 T 是一棵 RB_h 树（黑色高度为 h 的红黑树），则有：

1. T 至少有 $2^h - 1$ 个黑色内结点
2. T 至多有 $4^h - 1$ 个内结点
3. 任一黑色结点的深度至多是其黑色深度的 2 倍

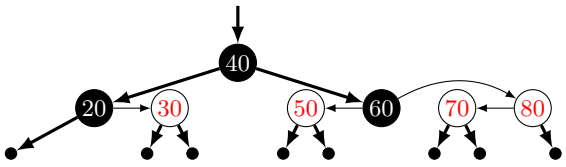
若 A 是一棵 ARB_h 树（黑色高度为 h 的近似红黑树），则有：

1. A 至少有 $2^h - 2$ 个黑色内结点
2. A 至多有 $\frac{1}{2}(4^h) - 1$ 个内结点
3. 任一黑色结点的深度至多是其黑色深度的 2 倍

Theorem 2.6. 若 T 是一棵包含 n 个内结点的红黑树，则其任一结点深度均不大于 $2\lg(n+1)$，即 T 的高度至多为 $2\lg(n+1)$

红黑树结点插入操作

- 红黑树的定义给出了关于结点颜色和黑色高度的约束条件，若要插入一个结点，则必须保证结点插入后的二叉树仍然能满足红黑树的约束条件
- 插入过程分为两个阶段：
  1. 首先看待插入元素 K 是否已经包含在红黑树中，即在红黑树所代表的二叉搜索树中查找元素 K，直到到达一个外结点（空树），然后将该外结点替换成一棵仅包含一个内结点 K 的子树
  2. 修正任何违反颜色约束的地方（在插入操作中，任何时候都不会出现违反黑色高度约束的地方）
- 例：插入的第一阶段产生的违反颜色约束的情况



修正操作

Definition 2.7.

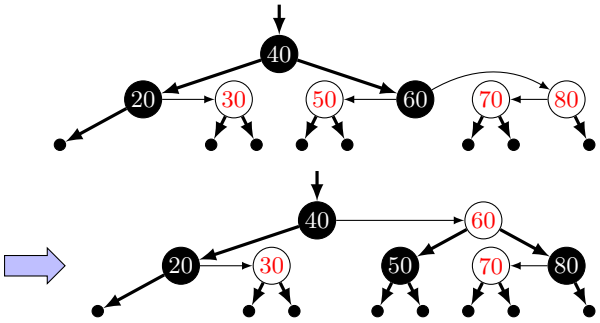
- **结点组** (cluster)：包含一个黑色内结点以及从该结点仅通过非黑色边可以到达的红色结点所组成的内结点集合，唯一的黑色结点称为该结点组的根
- **关键组** (critical cluster)：若结点组中存在这样的结点：从结点组的根到该结点的路径长度大于 1，则该结点组称为关键组（存在违反颜色约束的情况）

一些结论：

- 二叉树存在违反红黑树颜色约束的情况当且仅当其结点组中存在关键组
- 在插入操作过程中出现的关键组可能包含 3 个或 4 个结点
- 插入操作的第一阶段最多产生一个关键组
- 插入操作的第二阶段 (rebalancing) 采取某种策略消除产生的关键组，若不再产生新的关键组，则插入结束；否则最多在更接近根的位置产生一个新的关键组，修正过程将继续进行，直到无关键组产生

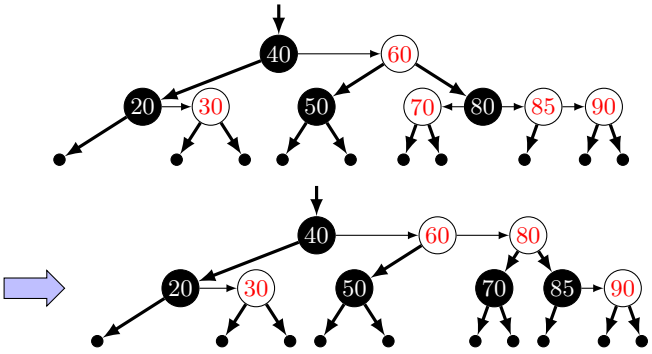
4 结点关键组消除：Color Flip

- 关键组根结点改成红色结点
- 根结点的左右子结点改成黑色结点
- 如果关键组根结点恰好也是整棵树的根结点，则将其改回黑色，只有在这种情况下整棵树的黑色高度被改变
- 例 1：不产生新的关键组

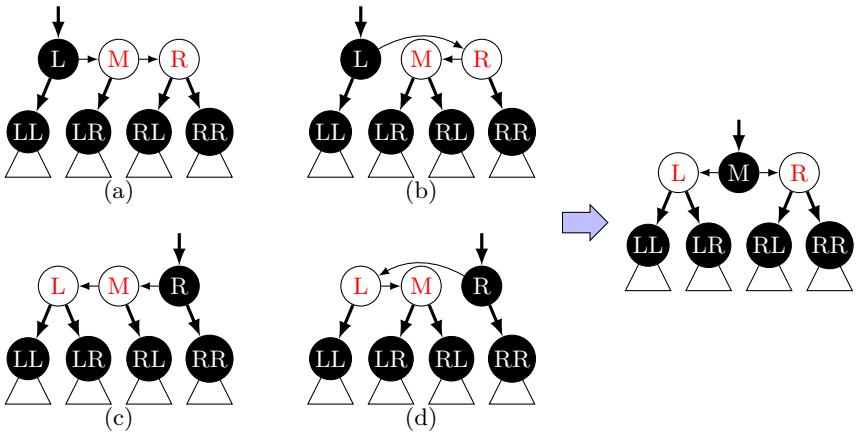


4 结点关键组消除：Color Flip (cont.)

- 例 2：有新的关键组产生



3 结点关键组消除



红黑树插入算法设计

- 返回值结构：

RBTree	**Element** root
	RBTree left
	RBTree right
	int color
InsReturn	**RBTree** newTree
	int status

- 常量定义：

color	red	根结点为红色
	black	根结点为黑色
status	ok	插入结束
	rbr	根结点为黑色，左右子结点为红色
	brb	根结点为红色，左右子结点为黑色
	rrb	根结点和左子结点为红色
	brr	根结点和右子结点为红色

插入算法

Algorithm `RBTIns`(RBTree $oldTree$, Element $newNode$)

```
1  InsReturn ans, ansLeft, andRight;
2  if oldTree = null then
3      ans.newTree ← 根结点是 newNode 的单结点 RBTree;
4      ans.status ← brb;
5  else
6      if newNode < oldTree.root then
7          ansLeft ← RBTIns(oldTree.left, newNode);
8          ans ← RepairLeft(oldTree, ansLeft);
9      else
10         ansRight ← RBTIns(oldTree.right, newNode);
11         ans ← RepairRight(oldTree, ansRight);
12     end
13 end
14 return ans;
```

左子树修正：

Algorithm `RepairLeft`(RBTree $oldTree$, InsReturn $ansLeft$)

```
1  InsReturn ans;
2  if ansLeft.status = ok then
3      ans.newTree ← oldTree;
4      ans.status ← ok;
5  else
6      oldTree.left ← ansLeft.newTree;
7      if ansLeft.status = rbr then                    /* 无需再修正 */
8          ans.newTree ← oldTree;
9          ans.status ← ok;
10     else if ansLeft.status = brb then/* 左子树没问题，检查根结点颜色
        */
11         if oldTree.color = black then
12             ans.status ← ok;
13         else  ans.status ← rrb ;
14         ans.newTree ← oldTree;
15     else if oldTree.right.color = red then          /* 4 结点关键组 */
16         ColorFlip(oldTree);
17         ans.newTree ← oldTree;
18         ans.status ← brb;
19     else                                            /* 3 结点关键组 */
20         ans.newTree ← RebalLeft(oldTree, ansLeft.status);
21         ans.status ← ok;
22     end
23 end
24 return ans;
```

3 结点关键组消除：

Algorithm `RebalLeft`(RBTree $oldTree$, int $leftStatus$)

```
1  RBTree L, M, R, LR, RL;
2  if leftStatus = rrb then                          /* 情况 (c) */
3      R ← oldTree;
4      M ← oldTree.left;
5      L ← M.left;
6      RL ← M.right;
7      R.left ← RL;
8      M.right ← R;
9  else                                  /* leftStatus = brr, 情况 (d) */
10     R ← oldTree;
11     L ← oldTree.left;
12     M ← L.right;
13     LR ← M.left;
14     RL ← M.right;
15     R.left ← RL;
16     L.right ← LR;
17     M.right ← R;
18     M.left ← L;
19 end
20 L.color ← red;
21 R.color ← red;
22 M.color ← black;
23 return M;
```

4 结点关键组消除：

Algorithm `ColorFlip`(RBTree $oldTree$)

```
1  oldTree.color ← red;
2  oldTree.left.color ← black;
3  oldTree.right.color ← black;
```

插入算法 (cont.)

- 由于最终的修正结果可能使根结点成为红色，仅当此时需要再次修正，为了不破坏递归调用的一致性，需要提供一个封装过程来处理这种情况：

Procedure `RBTInsert`(RBTree $oldTree$, Element $node$)

```
1  InsReturn ans ← RBTIns(oldTree, node);
2  if ans.newTree.color ≠ black then
3      ans.newTree.color ← black;
4  end
5  return ans.newTree;
```

算法分析

Lemma 2.8. 如果 RBTIns 的参数 $oldRBTree$ 是一棵 RB_h 树或 ARB_{h+1} 树，则返回值中的 newTree 和 status 取如下组合之一：

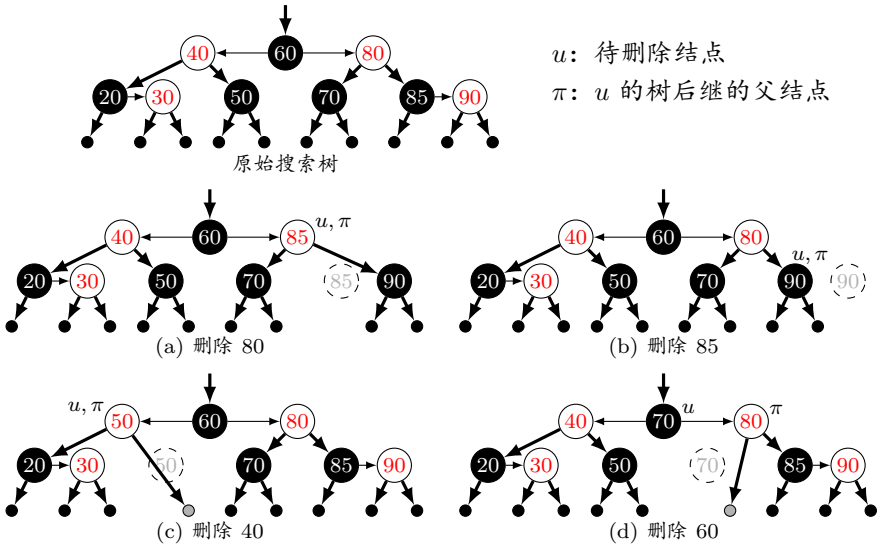
1. status=ok，newTree 是一棵 RB_h 树或 ARB_{h+1} 树
2. status=rbr，newTree 是一棵 RB_h 树
3. status=brb，newTree 是一棵 ARB_{h+1} 树
4. status=rrb,newTree.color=red,newTree.left 是一棵 ARB_{h+1} 树,newTree.right 是一棵 RB_h 树
5. status=brr,newTree.color=red,newTree.right 是一棵 ARB_{h+1} 树,newTree.left 是一棵 RB_h 树

Theorem 2.9. 上述红黑树插入算法能够将新结点正确地插入到红黑树中而不破坏其性质，插入一个结点到包含 n 个结点的红黑树最坏情况时间复杂度为 $\Theta(\log n)$

红黑树结点删除操作

- 从红黑树删除一个结点要比插入一个结点复杂的多
  - 插入总可以在叶结点处进行，而删除可以出现在任意位置
  - 删除操作有可能造成对高度规则的破坏，修正高度的错误比修正颜色的错误来得复杂
- 从二叉搜索树删除一个结点（不考虑高度规则）
  - 逻辑删除：结点中存储的元素被删除
  - 结构删除：结点被删除
  - 树后继 (tree successor)：2-tree 中任一内结点 u 的树后继是其右子树中最左边的内结点或者其右子树（如果其右子树的左子树为外结点）
  - 在二叉搜索树中删除一个结点，实际上是用其树后继的元素替换该结点元素（逻辑删除）并删除其树后继（结构删除）
  - 如果其树后继是外结点，则表明该结点元素最大，可以直接对该结点进行结构删除

结点删除实例



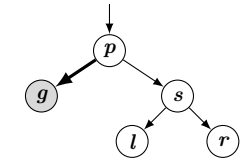
红黑树结点删除

- 红黑树结点删除操作可归纳为以下几个步骤
  1. 搜索定位待删除结点 u
  2. 若 u 的右子结点为外结点，则可直接对 u 进行结构删除
  3. 若 u 的右子结点为内结点，找到 u 的树后继，将其所存储的元素拷贝到 u 中（u 的颜色暂时不作调整），并对其树后继结点进行结构删除
  4. 修复所有由于结构删除所造成的对黑色高度约束的破坏
- 其中，步骤 4 是关键，需要详细考察；回顾前面的例子：
  - 在 (a) 中，尽管有黑色结点被结构删除，但它的右子结点是一个内结点，该结点一定是红色 (Why?)，因此可以直接将其改为黑色而不破坏红黑树特性
  - 在 (b) 中，对红色结点进行结构删除不会对红黑树产生任何影响
  - (c) 和 (d) 待删除结点的树后继是黑色结点，并且其右子结点也是黑色结点（一定是外结点），因此结构删除此树后继结点后，其右子结点（灰色）的黑色高度不足，红黑树将不再保持平衡，需要修正

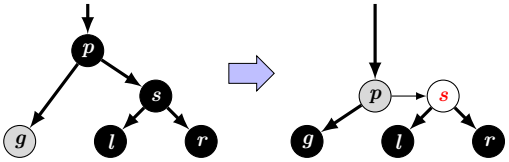
恢复黑色高度

- **灰色结点** (gray node)：一棵 RB_{h-1} 树的根结点（若标记为黑色），但其父结点需要一棵 RB_h 子树（黑色高度少了 1）
- 在修正开始时，灰色结点是一个外结点，随着修正过程进行，灰色状态将向上传递
- 修复黑色高度的基本策略：

- 寻找附近能够被变成黑色的红色结点，通过局部结构调整恢复高度平衡
- 如果找不到这样的红色结点，则将结点灰色状态向上传递（递归处理），当传递到根结点，则直接将根改回黑色即可



修复所涉及的关键结点

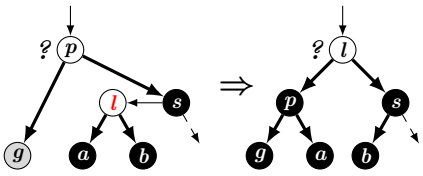


灰色结点向上转移

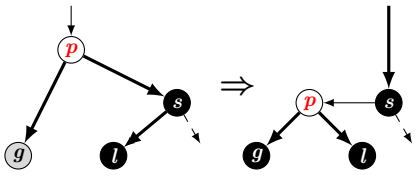
恢复黑色高度 (cont.)

- p, s, l, r 中任一个是红色结点，则灰色结点不需要向上转移，经局部结构调整即可消除黑色高度不平衡状态
- 构造以 p 为根的结点组，使得其所有主子树为 RB_{h-1} 树（和灰色结点 g 为根的子树一样），结构调整便局限在这个结点组及其主子树中，每棵主子树均可用一个外结点代表，则修复目标如下：
  1. 若 p 是红色结点，则该结点组应该调整为 RB_1 或 ARB_2 树
  2. 若 p 是黑色结点，则该结点组应该调整为 RB_2 树
- 若灰色结点 g 是左子结点，则按 l, p, r, s 的顺序分为 4 种情况进行处理（附后）
- 灰色结点 g 是右子结点的情况与左子结点处理完全对称
- 时间复杂度：
  - 子结构调整 $O(1)$
  - 灰色状态转移 $O(\log n)$

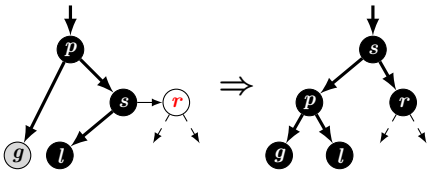
子结构调整



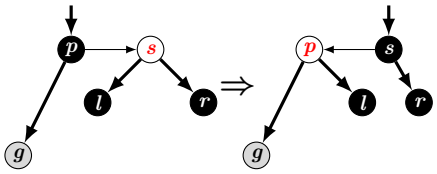
(a) l 是红色结点；p 可以是红色或黑色，调整后 l 的颜色与之相同



(b) p 是红色结点



(c) r 是红色结点



(d) s 是红色结点；调整后根据 l 左子结点的颜色转变为 (a) 或 (b)

2.3 散列技术

散列技术

- **散列** (Hashing)（也称哈希）技术通常用于实现字典数据结构：对每一个可能出现的键值赋予唯一序列索引使得元素的查找、插入、删除等操作非常方便和快捷
- 一些基本概念：
  - **散列码** (hash code)：键值对应的存储位置索引
  - **散列函数** (hash function)：用于计算键值所对应的散列码的函数
  - **冲突** (collision)：不同键值映射到同一个散列码
  - **散列表** (hash table)：数组 $H[0..h-1]$，用于存放 h 个散列元 (hash cell)，元素键值被散列函数映射到 $[0, h-1]$；通常散列技术的具体实现就是设计和维护一个这样的散列表
- 散列表的设计所需解决的问题：
  1. 使用什么样的散列函数？
  2. 如何处理冲突？

封闭地址散列

- **封闭地址散列** (Closed Address Hashing) 也叫链式散列 (Chained Hashing) 采用最简单的冲突处理策略：
  - H 中的每个散列元 $H[i]$ 是一个链表，链表中元素的散列码均为 i
  - 插入元素 K：计算 K 的散列码 i；将该元素插入链表 $H[i]$
  - 搜索元素 K：计算 K 的散列码 i；在链表 $H[i]$ 中查找元素 K
  - 负载因子 (load factor) $\alpha = n/h$，其中，n 为散列表中存储的元素数，即负载因子是散列表中平均每个散列元链表所存储的元素数
- 成功搜索的平均复杂度：$a + \frac{1}{n}\sum_{i=0}^{h-1}\frac{(L_i+1)L_i}{2}$
- 最坏情况下（散列函数映射不均匀或者添加的元素分布不均匀）搜索时间复杂度可到 $\Theta(n)$，与线性搜索复杂度相当
- 如果散列函数设计得足够好，使元素键值平均地映射到 $[0, h-1]$，则成功搜索的平均复杂度可以达到 $O(1+\alpha)$
- 不成功的搜索平均复杂度约为成功搜索的 2 倍

成功搜索平均复杂度：链表 L_i 的平均搜索长度是 $\frac{L_i+1}{2}$，散列函数命中 L_i 的概率为 L_i/n；总的平均搜索长度为：

$$\sum_{i=0}^{h-1}\sum_{j=0}^{L_i-1}\frac{j+1}{n} = \frac{1}{n}\sum_{i=0}^{h-1}\sum_{j=0}^{L_i-1}(j+1) = \frac{1}{n}\sum_{i=0}^{h-1}\frac{L_i(L_i+1)}{2}$$

如果元素键值平均地映射到 $[0, h-1]$，$L_i = \alpha$，成功搜索的平均复杂度为

$$a + \frac{1}{n}\sum_{i=0}^{h-1}\frac{(\alpha+1)\alpha}{2} = a + \frac{1}{n}\cdot\frac{(\alpha+1)h\alpha}{2} = a + \frac{1+\alpha}{2}$$

开放地址散列

- **开放地址散列** (Open Address Hashing)：
  - H 中每个散列元 $H[i]$ 直接存储元素而不是指向一个链表
  - 当冲突发生时，重新计算散列码 (rehashing) 找下一个可能的存储位置，因此，对于每个元素都有一系列可能的散列码与之对应
  - 灵活性不如封闭地址散列（负载因子不大于 1 的情况很少出现）
  - 但比封闭地址散列更节省空间，搜索速度也更快
- 最简单的散列码重算策略——线性探查 (linear probing)：
  - $\text{Rehash}(i) = (i+1) \mod h$
  - 可以证明，当负载因子达到 1 时，成功搜索的平均复杂度将达到 $\sqrt{n}$（长探查链的存在导致）
  - 为什么还要引入开放地址散列？$\Rightarrow$ 在低负载因子的情况下有更好的表现（负载因子 $\alpha < 1$ 时成功搜索可以达到 $\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$）
  - 利用“数组加倍”技术可使负载因子始终保持在 0.5 以下
- 更好的散列码重算策略——双散列 (double hashing)：
  - $d = \text{HashIncr}(K)$
  - $\text{Rehash}(i, d) = (i+d) \mod h$

散列函数设计

- 一个简单的设计原则是生成尽可能随机的散列码
- 例如采用“乘同余法 (multiplicative congruential) ”生成均匀分布的伪随机序列：乘以一个常数，然后对另一个常数求余
- 一个例子：
  1. 选择 h 为 2 的幂：$h = 2^x,\ h \geq 8$
  2. $a = 8\lfloor h/23 \rfloor + 5$
  3. 若元素键值是整数：$\text{HashCode}(K) = (aK) \mod h$
  4. 若元素键值是一对整数 (K_1, K_2)：$\text{HashCode}(K_1K_2) = (a^2K_1 + aK_2) \mod h$
  5. 若元素键值是一个字符串,将其当作一列整数 $k_1, k_2 \cdots$：$\text{HashCode}(K) = (a^l k_1 + a^{l-1}k_2 + \cdots + ak_l) \mod h$

6. 当负载因子大于 0.5 时，将数组空间加倍，更新 h 和 a 的值，将原表中的元素在新参数值下插入到新表中
7. 若采用双散列策略，HashIncr 可设计得尽量简单，但要保证在这个增量下不断重新计算散列码可以覆盖整个散列表，例如可取 $\text{HashIncr}(K) = (2k_1 + 1) \mod h$

动态集合 + 散列技术：布隆过滤器

- **布隆过滤器** (Bloom Filter) 1970 年由 Burton Howard Bloom 在论文 "Space/Time Trade-offs in Hash Coding with Allowable Errors" 中首先提出，是一种空间效率很高的动态集合，用于快速地查询一个元素是否在一个集合中
- 数据结构：Bit Vector（m 位）
- 添加元素：通过 k 个散列函数将这个元素映射成 Bit Vector 中的 k 个 bit，把它们置为 1
- 查询操作：用同样的散列函数进行计算，看散列值对应的 bit 位是否为 1，如果有一个不为 1，则这个元素一定不在集合中，否则只能表明这个元素很有可能在集合中
- 时间复杂度：插入和查询都是 $O(k)$
- 错误率 (probability of false positives)：假定散列值均匀分布，且各散列函数相互独立 $\left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx (1 - e^{-kn/m})^k$

更多关于散列技术的分析

- 参考文献：

[1] Donald E. Knuth. *The Art of Computer Programming - Volume 3: Sorting and Searching (2nd Edition)* (Section 6.4), Addison-Wesley, 1997

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. *Introduction to Algorithms (2nd Edition)* (Chapter 11), The MIT Press, 2001

[3] Gaston H. Gonnet and Ricardo Baeza-Yates. *Handbook of Algorithms and Data Structures*, Addison-Wesley, 1991

2.4 动态等价关系及合并-查找程序

动态等价关系

- 集合 **S** 上的 **等价关系** (equivalence ralation) **R** 是 **S** 上满足下列性质的二元关系：

1. **自反性** (reflexive)：$\forall x \in \mathbf{S},\ x\mathbf{R}x$
2. **对称性** (symmetric)：$\forall x, y \in \mathbf{S},\ x\mathbf{R}y \Rightarrow y\mathbf{R}x$
3. **传递性** (transitive)：$\forall x, y, z \in \mathbf{S},\ x\mathbf{R}y, y\mathbf{R}z \Rightarrow x\mathbf{R}z$

- **等价类** (equivalence class)：给定一个集合 $\mathbf{S}$ 和在 $\mathbf{S}$ 上的一个等价关系 $\equiv$，则 $\mathbf{S}$ 中的一个元素 x 的等价类是在 $\mathbf{S}$ 中等价于 x 的所有元素构成的子集，即 $[x] = \{y \in \mathbf{S} | x \equiv y\}$
- **动态等价关系** (dynamic equivalence relation)：随计算过程不断改变的等价关系
- 问题：动态等价关系的表示、更改和查询，即处理包含下面两种指令的指令序列

1. **IS** $s_i \equiv s_j$?
2. **MAKE** $s_i \equiv s_j$（之前 $s_i \equiv s_j$ 为假）

- 复杂度分析包含两个因素（输入数据）：

1. 包含 n 个元素的集合
2. 包含 m 条指令 (**IS** 或 **MAKE**) 的指令序列

- 两种显而易见的实现方式：
  - 矩阵实现：利用矩阵表示等价关系，最少需要 $n^2/2$ 个存储单元，**IS** 指令只需检查一个单元，复杂度为 $\Theta(1)$，**MAKE** 需要拷贝矩阵某些行的值，因此在最坏情况下，指令序列包含 m 个 **MAKE** 指令，需要至少 mn 次操作
  - 数组实现：用数组存储等价关系，数组元素 $A[i]$ 记录包含集合元素 s_i 的等价类标识，则 **IS** $s_i \equiv s_j$ 指令需要检查 $A[i]$ 和 $A[j]$ 是否相等，复杂度为 $\Theta(1)$，而 **MAKE** $s_i \equiv s_j$ 指令则需要检查所有数组元素，将所有等于 $A[i]$ 的元素重新赋值为 $A[j]$，因此在最坏情况下仍然需要至少 mn 次操作，但与矩阵方式相比，空间复杂度有所降低
- 有没有更快的方法？

“合并-查找”程序

- **合并-查找** (Union-Find) 程序是包含如下两条基本操作的程序：

1. `Union`(u, v)：合并等价类 $[u]$ 和 $[v]$ 成为一个等价类
2. `Find`(s)：查找集合元素 s 所在的等价类（标识）

- 前面所定义的 **IS** 和 **MAKE** 可以用这两条基本操作完成：

IS $s_i \equiv s_j$	**MAKE** $s_i \equiv s_j$
$u = \texttt{Find}(s_i)$;	$u = \texttt{Find}(s_i)$;
$v = \texttt{Find}(s_j)$;	$v = \texttt{Find}(s_j)$;
$(u = v)$?	$\texttt{Union}(u, v)$

- 初始化等价类：MakeSet(1), MakeSet(2), $\cdots$, MakeSet(n)
- 等价类的存储：in-tree (结点仅可访问其祖先，不能访问其子孙)

MakeNode	构造一棵包含一个结点的 in-tree
SetParent	改变一个结点的父结点
SetNodeData	设置结点所存储的元素值
IsRoot	测试结点是否是根结点（没有父结点的结点）
Parent	返回结点的父结点
NodeData	返回结点存储的元素值

“合并-查找”程序的应用

- 合并-查找程序主要应用在数据的内容和结构在处理过程中不断改变的情形 (on-line operation)
  - 可用于求带权无向图的最小生成树的 Kruskal 算法（在有关图算法的内容中会提到）
  - 可用于在程序设计语言中对等价声明 (equivalence declarations) 的处理。等价声明意味着多个变量或数组元素将共享同样的存储位置，例如，如下 Fortran 中的 EQUIVALENCE 语句调用

    EQUIVALENCE $(A, B(3)), (B(4), C(2)), (X, Y, Z),$
    $(J(1), K), (B(1), X), (J(4), L, M)$

    将产生如下效果：

		A					$J(1)$	$J(2)$	$J(3)$	$J(4)$	$J(5)$
$B(1)$	$B(2)$	$B(3)$	$B(4)$	$B(5)$			K			L	
X		$C(1)$	$C(2)$	$C(3)$	$C(4)$	$C(5)$				M	
Y											
Z											

  - 可以用于实现动态集合的其他基本操作

“合并-查找”程序的时间复杂度

- 每棵 in-tree 的根结点可作为等价类标识，则：
  - Find(s) 即返回包含结点 s 的 in-tree 的根结点：从 s 开始不断调用 Parent直到遇到根结点
  - Union(u, v) 的参数 u 和 v 必须为根结点且 $u \neq v$，其操作就是将 u 和 v 所代表的 in-tree 合并成一棵 in-tree：直接调用 SetParent(u, v) 即可完成

- 时间复杂度分析的输入规模：包含 n 个元素的集合以及规模为 m 的合并-查找程序：由 MakeSet 构成的等价类初始化序列及 m 个 Union 或 Find 操作组成的操作序列
- 显而易见，合并-查找程序的执行时间与对根结点的访问 (lookup 和 assignment) 次数成正比，因此可以将根结点的访问作为基本操作，并称其为链接操作 (link operation)，认为其时间复杂度为 $\Theta(1)$，则：
  - MakeSet 和 Union 包含 1 次根结点赋值操作
  - Find(s) 包含 $d+1$ 次根结点查询操作，d 为结点 s 的深度
  - 合并-查找程序最坏情况时间复杂度为 $\Theta(mn+n) = \boldsymbol{\Theta(mn)}$

加权合并

- 合并-查找程序复杂度为 $\Theta(mn)$ 的主要原因是 in-tree 的高度得不到限制（最高可达 $n-1$），使得最坏情况下 Find 操作很低效
- 改进办法是设计更好的 Union 操作，使 in-tree 的高度得到控制
- **加权合并** (Weighted Union)：WUnion(u, v) 总是取 u 和 v 中权值较大的那个作为合并后 in-tree 的根，其时间复杂度仍然为 $\Theta(1)$
- 最简单的权值就是取 in-tree 所包含的结点数

Lemma 2.10. 如果用 WUnion 实现等价类 *in-tree* 的合并操作，即选择结点数较多的 *in-tree* 的根做为合并后 *in-tree* 的根，而另一棵 *in-tree* 则作为其一棵子树，则一系列 WUnion 构造的任何包含 k 个结点的 *in-tree* 高度至多为 $\lfloor \lg k \rfloor$.

Proof. 使用数学归纳法.
奠基：$k=1$，高度为 $0 = \lfloor \lg 1 \rfloor$.
归纳：假设对 $m < k$，结论成立，即任何由 WUnion 序列构造的包含 m 个结点的树高度至多为 $\lfloor \lg m \rfloor$，则对于由 T_1 和 T_2 两棵树通过 WUnion 构造的含 k 个结点，高度为 h 的树，不妨设 T_1 结点更多，则 T_2 作为子树链接到 T_1 的根结点，设 k_1, h_1 和 k_2, h_2 分别为 T_1 和 T_2 的结点数和高度，则有：$h_1 \le \lfloor \lg k_1 \rfloor$, $h_2 \le \lfloor \lg k_2 \rfloor$，合并后树的高度 $h = \max(h_1, h_2+1)$，又因为 $k_2 \le k/2$，故 $h_2 \le \lfloor \lg k \rfloor - 1$，而显然 $h_1 \le \lfloor \lg k \rfloor$，所以 $h \le \lfloor \lg k \rfloor$. □

使用加权合并的合并-查找程序时间复杂度

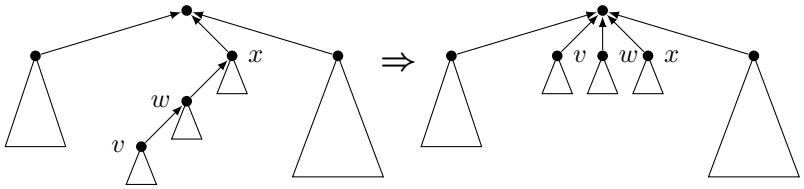
Theorem 2.11. 包含 n 个元素的集合上含有 m 条基本操作的合并-查找程序，若使用 WUnion 和 Find，则在最坏情况下需要执行 $\Theta(n + m \log n)$ 次链接操作.

Proof. 对于 n 个元素的集合，最多可执行 $n-1$ 次 WUnion，构造出一棵含 n 个结点的树，根据引理，其高度至多为 $\lfloor \lg n \rfloor$，则每个 Find 操作最多执行 $\lfloor \lg n \rfloor + 1$ 次链接操作，而每个 WUnion 操作执行一次链接操作，因

此最坏情况下的合并-查找程序对应 m 条操作全部为 Find，总共需要最多 $m(\lfloor \lg n \rfloor + 1)$ 次的链接操作，因此可以得到合并-查找程序链接操作次数的上界为 $O(n + m \log n)$，此外可以很容易构造出一个合并-查找程序使其链接操作次数下界为 $\Omega(n + m \log n)$. □

路径压缩

- Find 操作也可以改进以提高其速度
- **路径压缩** (path compression)：CFind(v) 在向上搜索 v 所在树的根结点过程中，将搜索路径上遇到的所有结点的父结点都改为最终找到的根结点



带路径压缩的查找操作

Procedure CFind(v)

```
1 oldParent ← Parent(v);
2 if oldParent =null then root ← v ;
3 else
4 |   root ← CFind(oldParent);
5 |   if oldParent ≠ root then SetParent(v, root);
6 end
7 return root;
```

- CFind 链接操作次数是 Find 的两倍
- 但 CFind 的使用使得 in-tree 的高度保持在很矮的状态
- 可以证明，使用 CFind 和无加权的 Union 作为基本操作，合并-查找程序的最坏情况时间复杂度仍然为 $\Theta(n + m \log n)$ *

同时使用 WUnion 和 CFind

- WUnion 和 CFind 操作是否相容？
  - 显然 WUnion 不会影响 CFind
  - 由于 CFind 只改变 in-tree 的高度而不改变其结点数，因此不会影响 in-tree 的权值，因而也不影响 WUnion 的操作
- 复杂度分析

*Michael J. Fischer. Efficiency of Equivalence Algorithms. In: *Complexity of Computer Computations*, pages 153–167. Plenum Press, New York, 1972.

- 从直观上讲应该比 $\Theta(n + m \log n)$ 更好一些
- CFind 的时间开销比较难把握，比数组加倍的分析更复杂一些
- 可以用分摊时间分析法来进行分析

由 WUnion 所构造的 in-tree 的一些性质

Definition 2.12 (森林 F，结点高度和阶)**.** 对于一个特定的合并-查找程序 P，F 是 WUnion 操作序列（忽略所有的 CFind 操作）构造出的森林 (forest)，结点 v 的高度 (node height) 是森林 F 中以 v 为根的子树的高度，v 的阶 (rank) 定义为其结点高度.

Lemma 2.13. 在森林 F 中：

1. 阶为 r 的结点至多有 $n/2^r$ 个
2. 任意结点的阶不大于 $\lfloor \lg n \rfloor$
3. 在执行合并-查找程序 P 的任意时刻，任一 *in-tree* 从叶结点到根结点的路径上结点的阶呈严格递增序列，当 CFind 改变一个结点的父结点时，新的父结点的阶大于原父结点阶.

一个新的函数：$\lg^*$

Definition 2.14 (函数 H 和 $\lg^*$)**.** 定义函数 H 如下：

$$\begin{aligned} H(0) &= 1, \\ H(n) &= 2^{H(n-1)} \quad \text{for } n > 0 \end{aligned}$$

则：$\lg^*(m) = \min\{n | H(n) \geq m\} \quad (m > 0)$

n	0	1	2	3	4	5	6	$\cdots$
$H(n)$	1	2	4	16	65536	2^{65536}	$2^{2^{65536}}$	$\cdots$
$\lg^*(n)$		0	1	2	2	3	3	$\cdots$

n	16	17	$\cdots$	65536	65537
$H(n)$	??	??	$\cdots$	??	??
$\lg^*(n)$	3	4	$\cdots$	4	5

- 由定义可知：$\forall$常数$p \geq 0,\ \lg^*(n) \in o(\log^{(p)}(n))$

$\lg^*$ 的递归定义：

$$\begin{aligned} \lg^*(1) &= 0, \\ \lg^*(n) &= 1 + \lg^*(\lceil \lg n \rceil) \quad \text{for } n > 1 \end{aligned}$$

结点组

Definition 2.15 (结点组 (node groups)). 结点组 $s_i = \{v | v \in F, \lg^*(1 + \text{rank}(v)) = i\}$

- 不同的阶和其对应的结点组：

r (rank)	0	1	$2 \sim 3$	$4 \sim 15$	$16 \sim 65535$	$65536 \sim (2^{65536} - 1)$
i (group)	0	1	2	3	4	5

Lemma 2.16. 包含 n 个元素的集合 **S** 中不同结点组个数至多为 $\lg^*(n+1)$

Proof. 任一结点的阶最大为 $\lfloor \lg n \rfloor$，因此最大结点组下标为：

$$\lg^*(1 + \lfloor \lg n \rfloor) = \lg^*(\lceil \lg(n+1) \rceil) = \lg^*(n+1) - 1$$

而下标从 0 开始，所以最多结点组个数为 $\lg^*(n+1)$ □

时间分摊策略设计

1. `MakeSet`
   - *accounting cost*: $4\lg^*(n+1)$; *actual cost*: 1; *amortized cost*: $1 + 4\lg^*(n+1)$
2. `WUnion`
   - *accounting cost*: 0; *actual cost*: 1; *amortized cost*: 1
3. `CFind`
   - 设调用 `CFind`(v) 时从 v 到根结点的路径为 $w_0, w_1, \cdots, w_k$，其中 w_k 为根结点：当 $k \geq 2$ 时，对路径上的每一个结点对 (w_{i-1}, w_i)，若 w_{i-1} 和 w_i 所在的结点组相同且 $1 \leq i \leq k-1$，则 *accounting cost* 为 -2，称结点 w_{i-1} 提取开销 2；其他情况 *accounting cost* 为 0
   - *actual cost*: $2k$
   - *amortized cost*: $2k - 2((k-1) - (\text{路径上不同结点组个数} - 1)) \leq \mathbf{2k - 2((k-1) - (\lg^*(n+1) - 1)) = 2\lg^*(n+1)}$
   - 尽管最坏情况下 `CFind` 的实际开销可达 $2\lg n$，经过时间分摊后，每个 `CFind` 的开销最多仅为 $\mathbf{2\lg^*(n+1)}$

分摊策略的合理性

Lemma 2.17. 上述时间分摊策略是合理的，即其 *accounting cost* 始终非负

Proof. 等价类初始化总共调用 n 次 `MakeSet`，存入 *accounting cost* 开销总和为 $4n\lg^*(n+1)$；任一结点 w 提取开销发生在 w 处于某一次 `CFind` 操作路径上，且与其父结点在同一结点组内，且其父结点不是根结点。当 `CFind` 给其指定新的父结点时，根据前面的引理，新父结点的阶大于其原父结点的阶，则在不断调用 `CFind` 的同时，其不断更换的父结点的阶也在不断增长，一旦某次新父结点的阶高到与其不在同一个结点组内，则 w 就不再有提取开销的可能，在此之前，其提取开销的次数最多不超过其所在结点组中不同阶的结点个数 ($< H(i)$)，因此，我们有了一个 F 中所有结点可能提取开销次数的上界：

$$\sum_{i=0}^{\lg^*(n+1)-1} H(i)\cdot(\text{结点组 } i \text{ 中的结点个数})$$

根据引理，阶为 r 的结点最多为 $n/2^r$，因此结点组 i 中的结点个数可计算如下：

$$\sum_{r=H(i-1)}^{H(i)-1}\frac{n}{2^r}\le\frac{n}{2^{H(i-1)}}\sum_{j=0}^{\infty}\frac{1}{2^j}=\frac{2n}{2^{H(i-1)}}=\frac{2n}{H(i)}$$

则提取开销总和上界为：

$$2\sum_{i=0}^{\lg^*(n+1)-1}H(i)\left(\frac{2n}{H(i)}\right)=4n\lg^*(n+1)$$

不超过由 `MakeSet` 存入的开销总和，因此得证。 □

`WUnion` + `CFind` 实现的合并-查找程序复杂度上界

Theorem 2.18. 在有 n 个元素的集合上执行包含 m 条由 `WUnion` 和 `CFind` 操作组成的合并-查找程序最坏情况下总共需要 $O((n+m)\lg^*(n))$ 次链接操作

- 一点讨论：
  - 从 $\lg^*$ 的定义可以看出它是一个增长速度相当缓慢的函数，在实际有意义的输入规模下，基本上可以认为 $\lg^* n\le 5$，可以把它当作常数
  - 从证明过程可以看出我们在建立不等式时条件是相当宽松的，这是否意味着由 `WUnion` 和 `CFind` 实现的合并-查找程序复杂度有可能达到 $\Theta(n+m)$？
  - 是否存在 `Union` 和 `Find` 的某种实现技术使合并-查找程序达到线性复杂度？这仍然是一个未解决的问题
- 参考文献：

[1] J. E. Hopcroft and J. D. Ullman. Set Merging Algorithms. *SIAM Journal on Computing*, 2(4):294–303, 1973

[2] R. E. Tarjan. On the Efficiency of a Good but Not Linear Set Union Algorithm. *Journal of the ACM*, 22(2):215–225, 1975

[3] M. L. Fredman and M. E. Saks. The Cell Probe Complexity of Dynamic Data Structures. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, pages 345–354, 1989

2.5 优先队列和配对森林

优先队列和减少键值操作

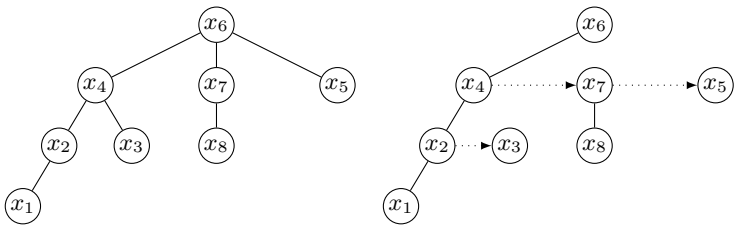
- 最小优先队列 (minimizing priority queue) 所需提供的操作：

  构造方法： `Create`
  状态访问： `IsEmpty`, `GetMin`, `GetPriority`
  基本操作： `Insert`, `DeleteMin`, `DecreaseKey`

- 利用二叉堆实现最小优先队列：
  - 基本操作可在 $O(\log n)$ 时间复杂度内实现（`DecreaseKey` 需要辅助数据结构）
  - `IsEmpty` 和 `GetMin` 时间复杂度为 $O(1)$
  - `GetPriority` 在辅助数据结构的帮助下可以达到 $O(1)$
  - 辅助数据结构：设置一个字典数据结构 xref，关键字为元素唯一 id，对应的值为该元素在二叉堆中的位置（最简单的实现：id 为整数，字典可用数组存储）
- 如果对 `DecreaseKey` 操作调用相当频繁，是否有进一步降低复杂度的方法？

配对森林

- **配对森林** (Pairing Forest)：一组满足偏序树特性的 out-tree 构成的森林（这里只研究满足最小偏序树特性的配对森林）
  - **偏序树特性** (partial order tree property)：树中任一结点的键值小于（或大于）其所有子结点的键值
  - out-tree：与 in-tree 相对，即一般的树结构，从任一结点可以访问其所有子结点
  - 在配对森林中，每一条从根结点到叶结点的路径都构成一个键值（或优先值）的递增序列
  - 配对森林可用链表结构存储：



取最小元素（联赛策略）

Algorithm `GetMin`(pq)

```
1 while pq.forest 多于 1 棵树 do pq.forest ← PairForest(pq.forest) ;
2 return 仅剩一棵树的根结点 id;
```

Procedure `PairForest`($oldForest$)

```
1  remainTrees ← oldForest;
2  while remainTrees ≠ null do
3      t1 ← First(remainTrees);
4      remainTrees ← Rest(remainTrees);
5      if remainTrees = null then newForest ←
         Cons(t1, newForest) ;
6      else
7          t2 ← First(remainTrees);
8          t ← PairTree(t1, t2);
9          newForest ← Cons(t, newForest);
10         remainTrees ← Rest(remainTrees);
11     end
12 end
13 return newForest;
```

取最小元素 (cont.)

Procedure `PairTree`(Tree $t1$, Tree $t2$)

```
1 if Root(t1).priority < Root(t2).priority then
2     newTree ← BuildTree(Root(t1), Cons(t2, Children(t1)));
3 else
4     newTree ← BuildTree(Root(t2), Cons(t1, Children(t2)));
5 end
```

- 复杂度分析
  - 如果在调用 `GetMin` 之前配对森林中有 k 棵树，则比较次数为 $k-1$（找最小元素所需的最少比较次数）
  - 在最坏情况下 k 可能等于结点数 n，但配对操作会使树的数目减少，因此最坏情况并不多见
  - 关于该操作的准确复杂度分析仍然是未知的

插入和删除

Procedure `Insert`(pq, v, w)

```
1 构造 newNode 使其 id 为 v，优先值 priority 为 w;
2 newTree ← BuildTree(newNode, null);
3 xref[v] ← newTree;
4 pq.forest ← Cons(newTree, pq.forest);
```

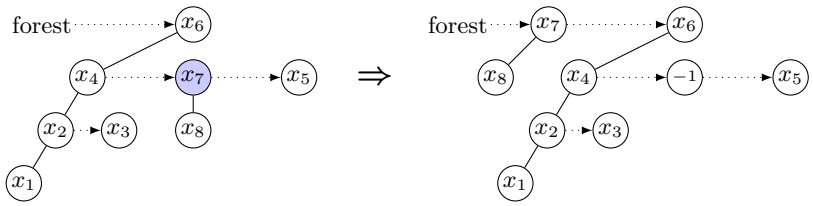
时间复杂度为 $O(1)$.

Procedure `DeleteMin`(pq)

```
1 GetMin(pq);
2 t ← First(pq.forest);
3 pq.forest ← Children(t);
```

时间复杂度与 `GetMin` 相同.

降低键值



Procedure `DecreaseKey`(pq, v, w)

```
1 构造 newNode 使其 id 为 v，优先值 priority 为 w;
2 oldTree ← xref[v];
3 oldNode ← Root(oldTree);
4 newTree ← BuildTree(newNode, Children(oldTree));
5 xref[v] ← newTree;
6 oldNode.id ← −1;
7 pq.forest ← Cons(newTree, pq.forest);
```

时间复杂度为 $O(1)$.