算法设计与分析

- LECTURE 5

Lecture 5

查找

Outline

- 折半查找
- 平衡二叉搜索树

- 并查集
- 哈希表
- 平摊分析

查找问题

- 查找问题
  - 在一组键值中找指定的键值
- 期望查找代价
  - 蛮力方法：$O(n)$
  - 理想代价：$O(1)$
  - 合理期待：$O(\log n)$

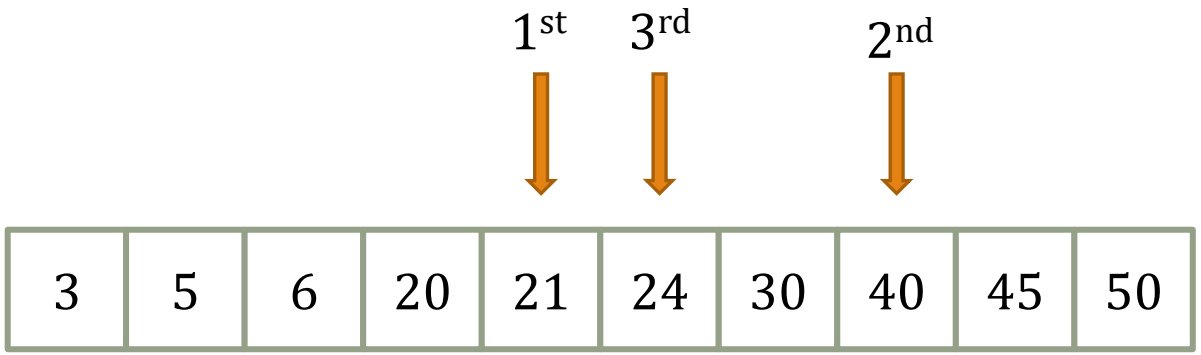
查找问题

- 关键要素：
  - 如何组织数据进行有效查找
  - log n 查找
    - 每次将搜索空间折半
    - 如何组织数据保证 log n 查找？

- log n 查找
  - 折半查找
  - 平衡二叉搜索树(BST)
    - 红黑树

折半查找

- 例子：查找“24”
  - 有序数组，常数时间寻址
  - 折半搜索空间



折半查找—应用

- 山峰数组的顶部
  - 单峰数组
- 不在数组里的最小元素
  - 排序好的自然数
- A[i]=i
  - 排序好的整数

剑指 Offer II 069. 山峰数组的顶部

难度 简单 56 收藏 分享 切换为英文 接收动态 反馈

符合下列属性的数组 `arr` 称为 **山峰数组（山脉数组）** ：

- `arr.length >= 3`
- 存在 `i`（`0 < i < arr.length - 1`）使得：
  - `arr[0] < arr[1] < ... arr[i-1] < arr[i]`

1095. 山脉数组中查找目标值

难度 困难 136 收藏 分享 切换为英文 接收动态 反馈

（这是一个 **交互式问题** ）

给你一个 **山脉数组** `mountainArr` ，请你返回能够使得 `mountainArr.get(index)` **等于** `target` **最小** 的下标 `index` 值。

如果不存在这样的下标 `index` ，就请返回 `-1` 。

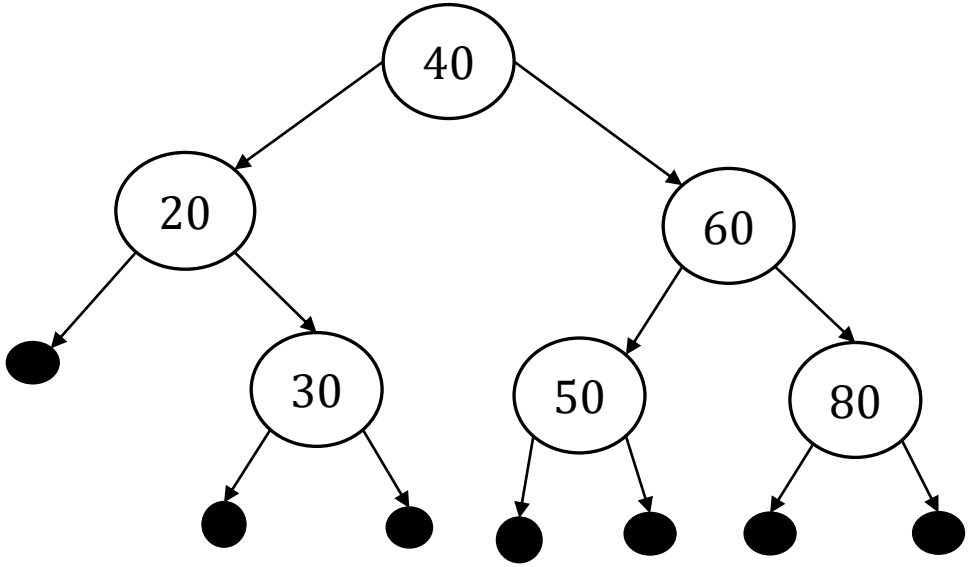
Lecture 5

平衡二叉搜索树

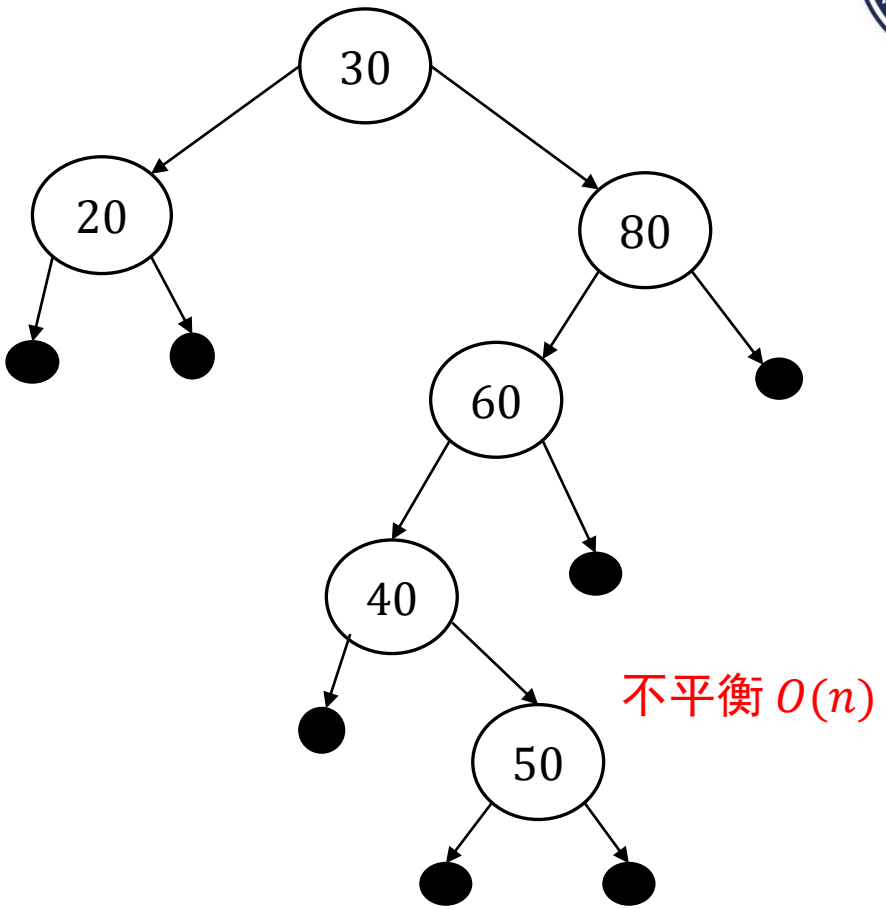
Outline

- 二叉搜索树
  - 定义和基本操作
- 红黑树定义
- 红黑树基本操作
  - 插入
  - 删除

二叉搜索树



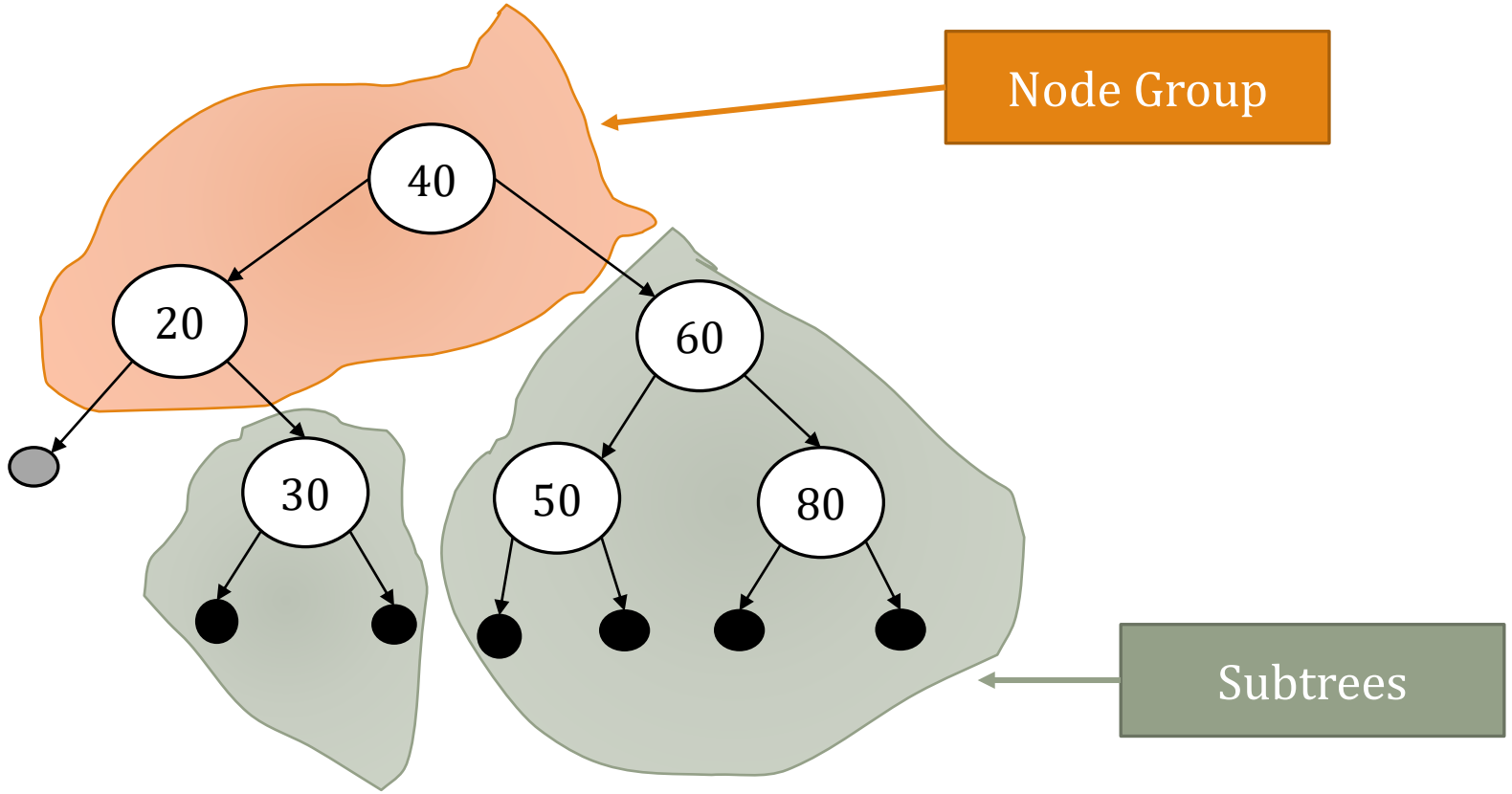
好的平衡 $O(\log n)$



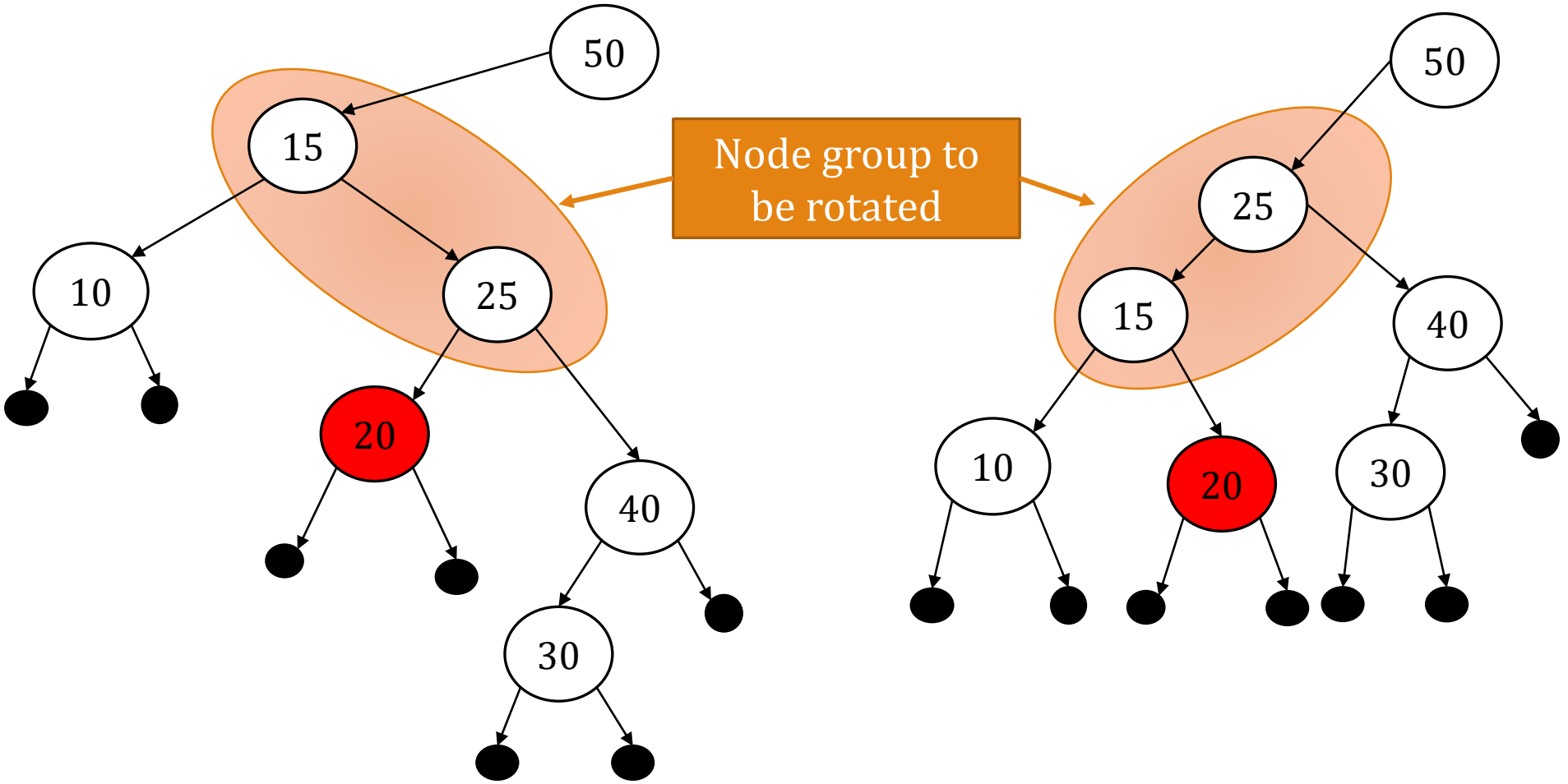
不平衡 $O(n)$

如何构建平衡的二叉树？

Node group



Rotation



红黑树

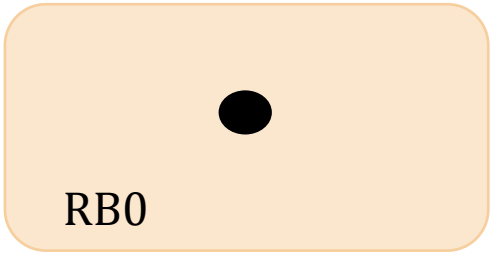
- 红黑树（Red-Black Tree）是一棵二叉搜索树，且满足：
  - 颜色限制：红色父节点没有红色子节点
  - 黑色高度限制：以任意节点为根的子树中，所以外部节点的黑色深度相等。
  - 根节点为黑色，所有外部节点为黑色。

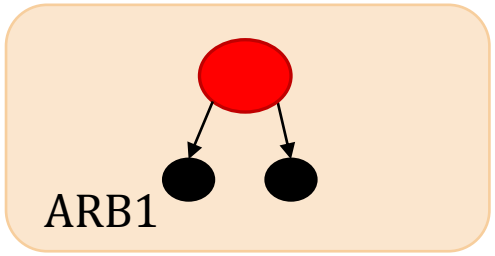
- 准红黑树（Almost-red-black tree，ARB tree)
  - 根节点是红色，满足其他限制

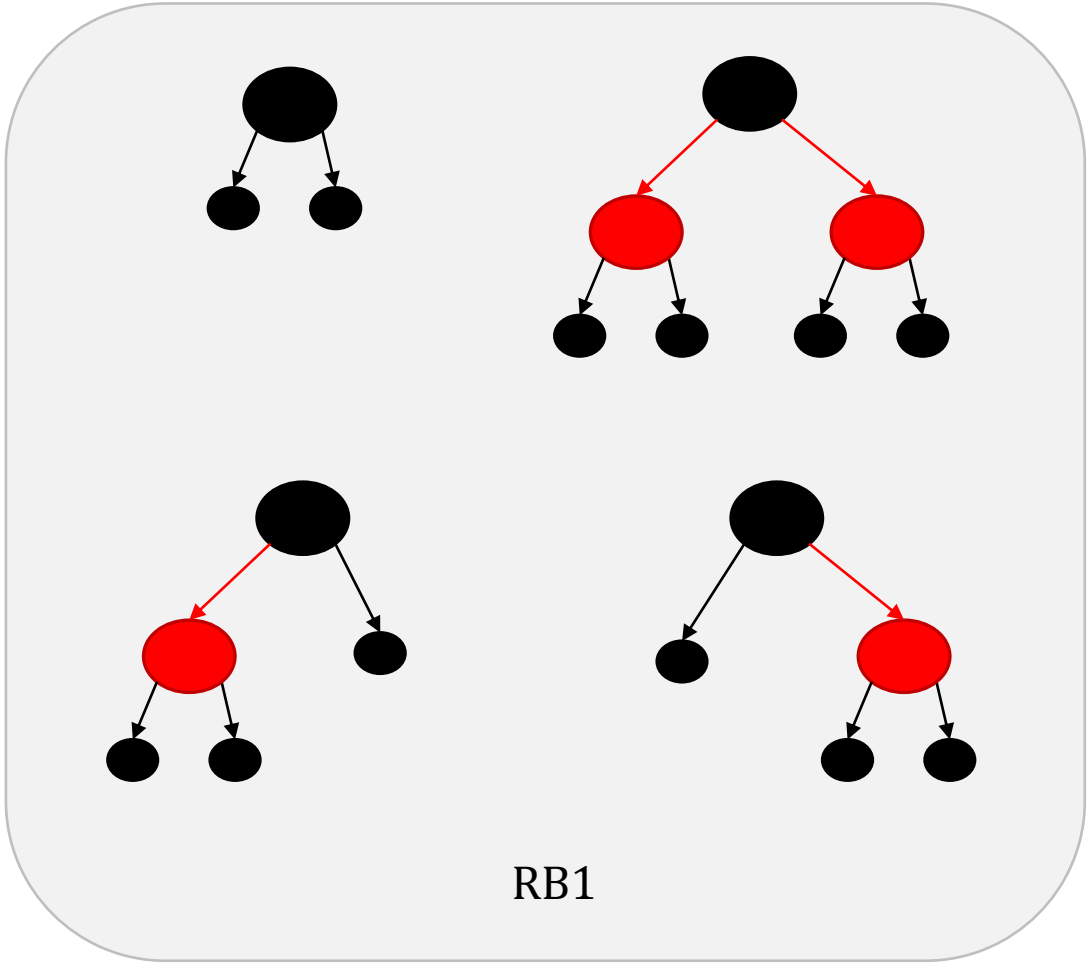
红黑树递归定义

- 唯一一个外部节点（同时也是根节点）构成一棵黑色高度为0的红黑树 RB_0
- 对于 $h \geq 1$，一棵二叉树为准红黑树 ARB_h，如果它的根节点为红色，且其左右子树均为 RB_{h-1}
- 对于 $h \geq 1$，一棵二叉树为红黑树 RB_h，如果它的根节点为黑色，且其左右子树分别为一棵 RB_{h-1} 或者 ARB_h

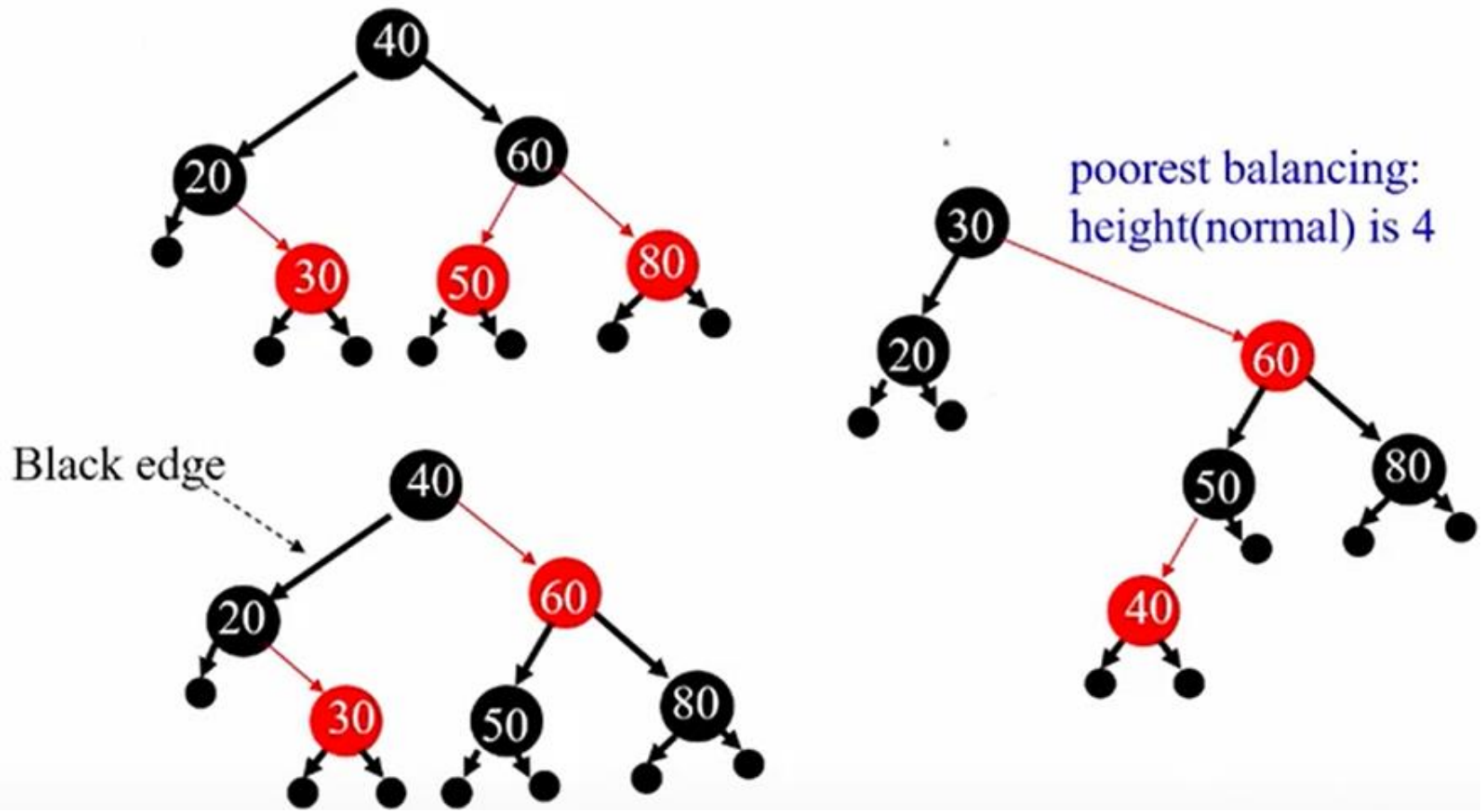
红黑树与准红黑树







6个节点的红黑树


40
20
60
30
50
80
Black edge
40
20
60
30
50
80
poorest balancing:
height(normal) is 4
30
20
60
50
80
40

红黑树的平衡性

- 假设 T 为一棵 RB_h，则：
  - T 有不少于 $2^h - 1$ 个内部黑色节点
  - T 有不超过 $4^h - 1$ 个内部节点
  - 任何黑色节点的普通高度至多是其黑色高度的2倍

- 假设 T 为一棵 ARB_h，则：
  - T 有不少于 $2^h - 2$ 个内部黑色节点
  - T 有不超过 $\frac{1}{2}4^h - 1$ 个内部节点
  - 任何黑色节点的普通高度至多是其黑色高度的2倍

红黑树平衡性

- 定理：假设T为一个有 n 个内部节点的红黑树，则红黑树的普通高度不超过 $2\log(n+1)$，基于红黑树的查找代价为 $O(\log n)$

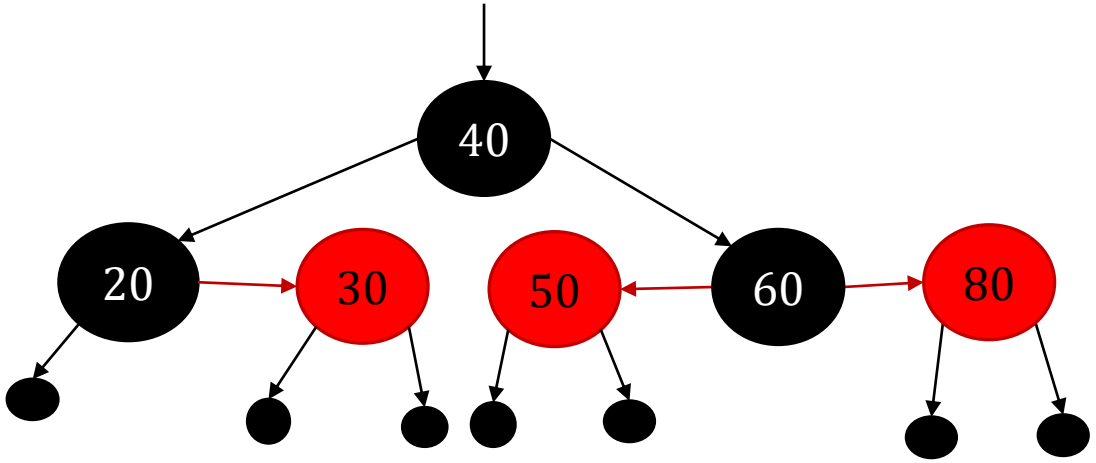
Lecture 5

平衡二叉搜索树

Outline

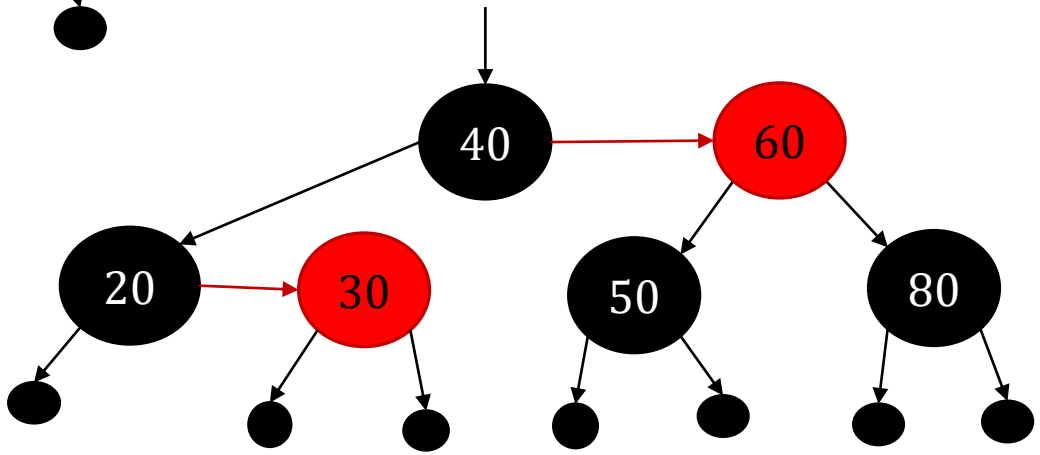
- 红黑树基本操作
  - 插入
  - 删除

Black-depth



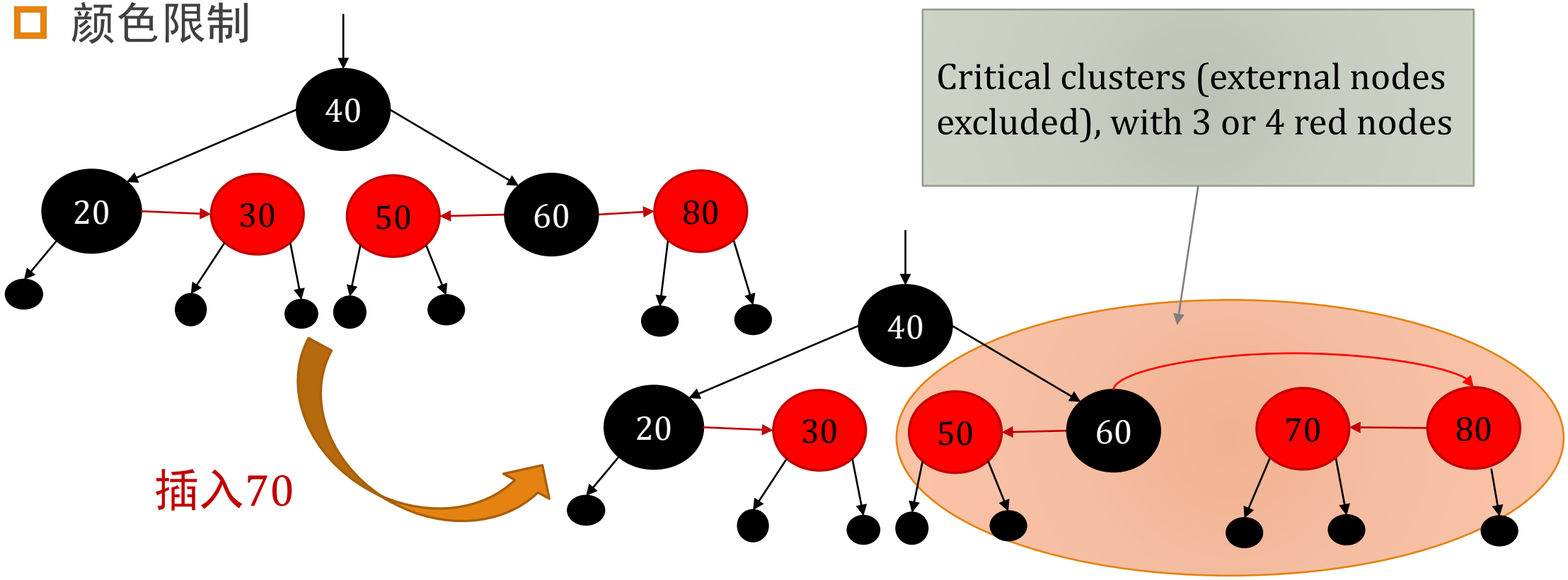
最大黑色高度为2

红色水平画，黑色加一层

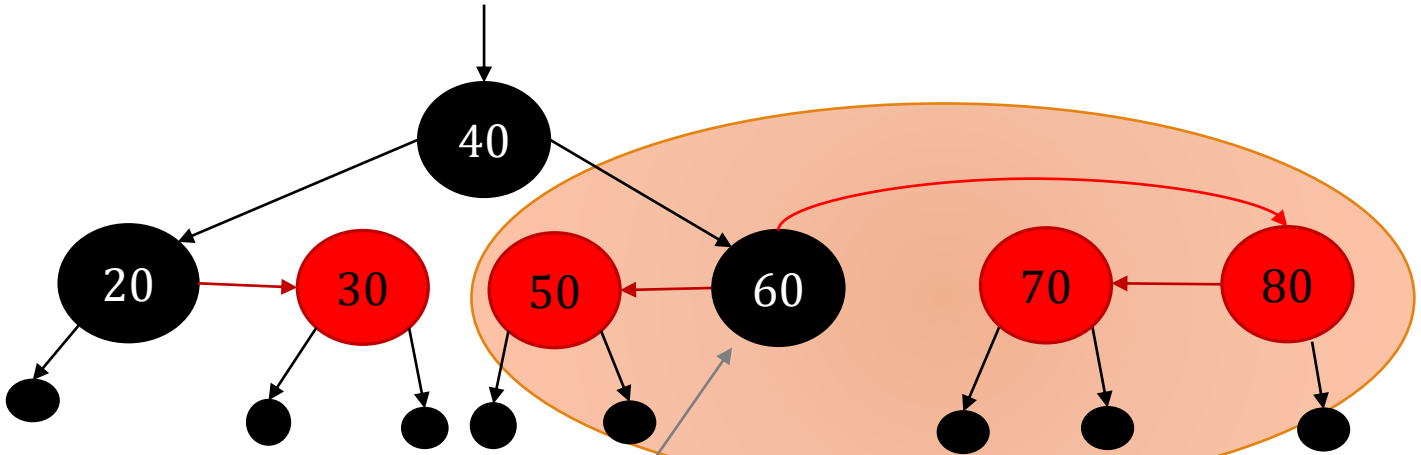


红黑树的插入

- 黑色高度限制：
  - 如果插入红色节点，无破坏
- 颜色限制

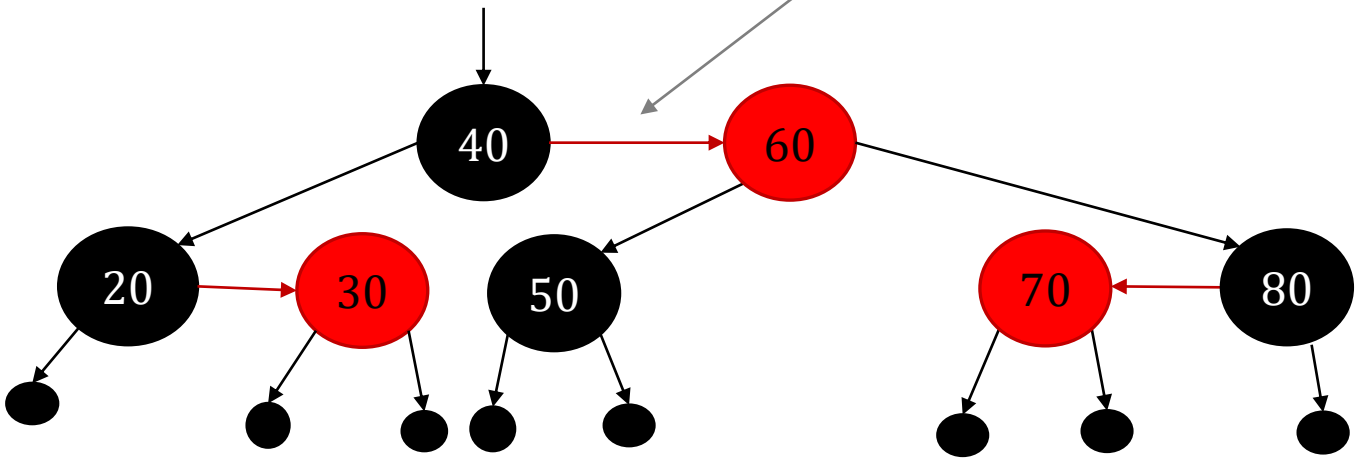


4-node Critical Cluster

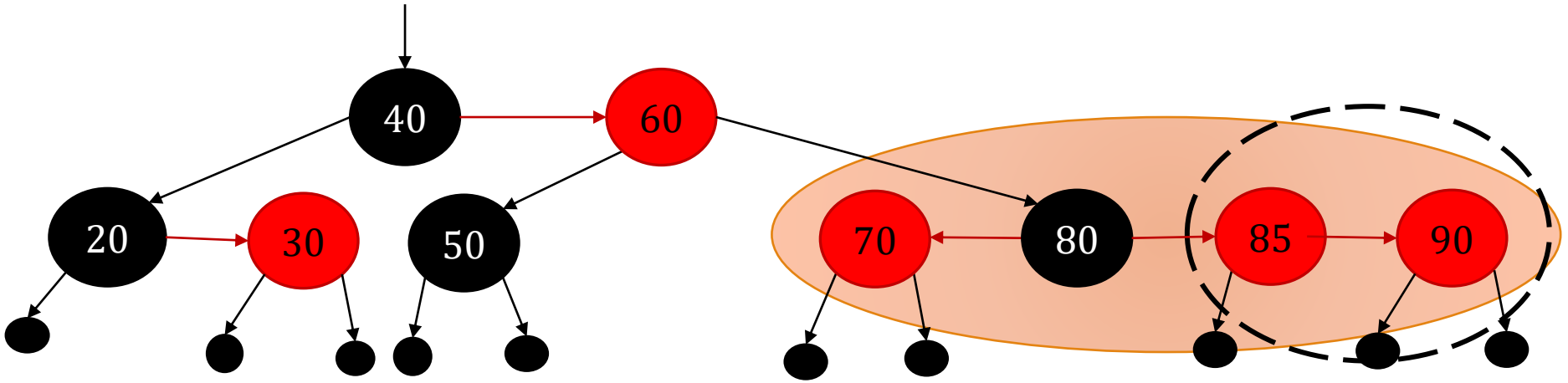


Root of critical cluster exchanges colors with its subtrees.

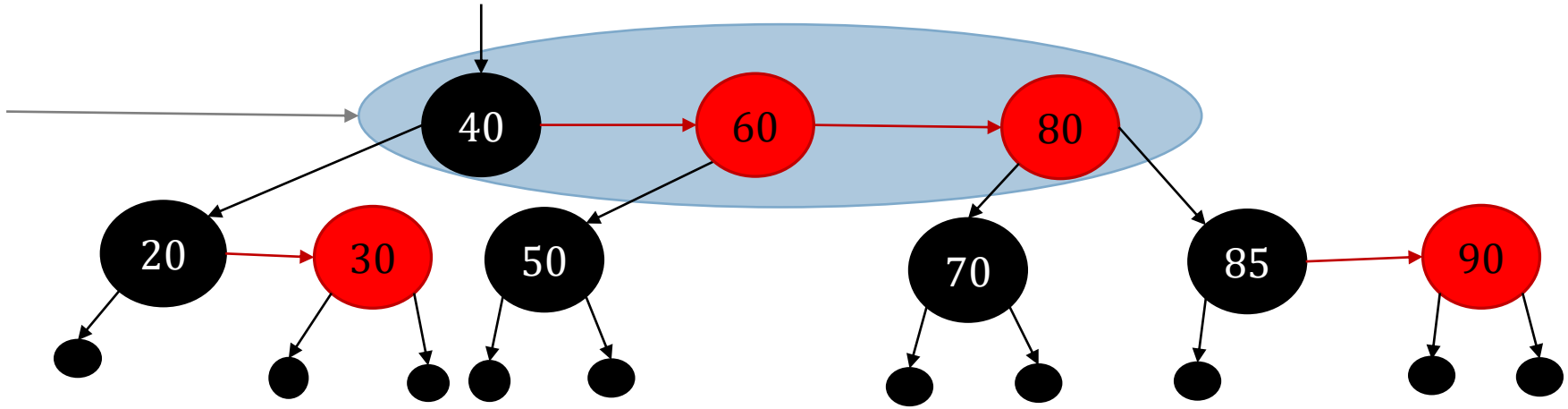
No new critical cluster occurs, inserting finished.



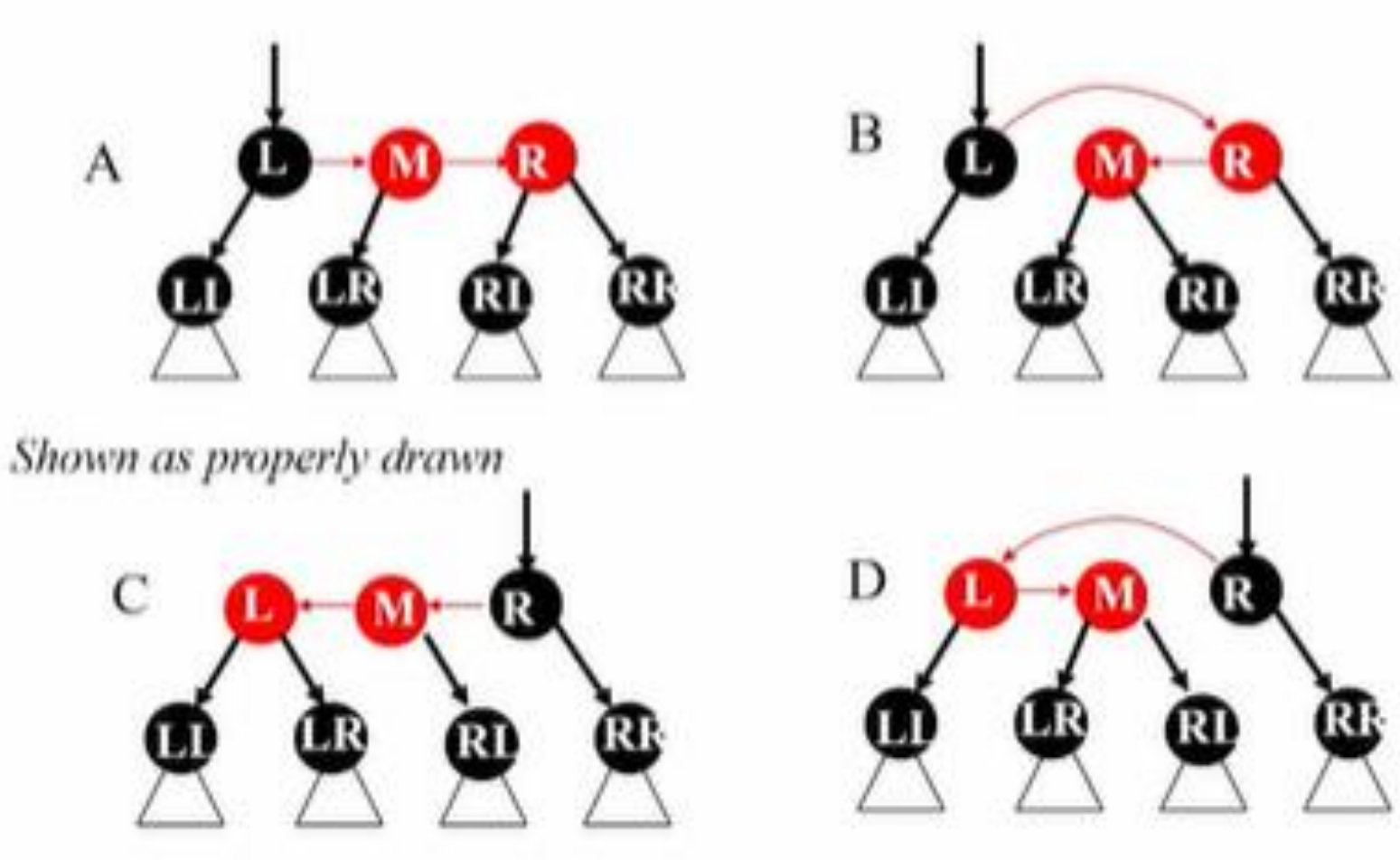
4-node Critical Cluster



New critical cluster with 3 nodes

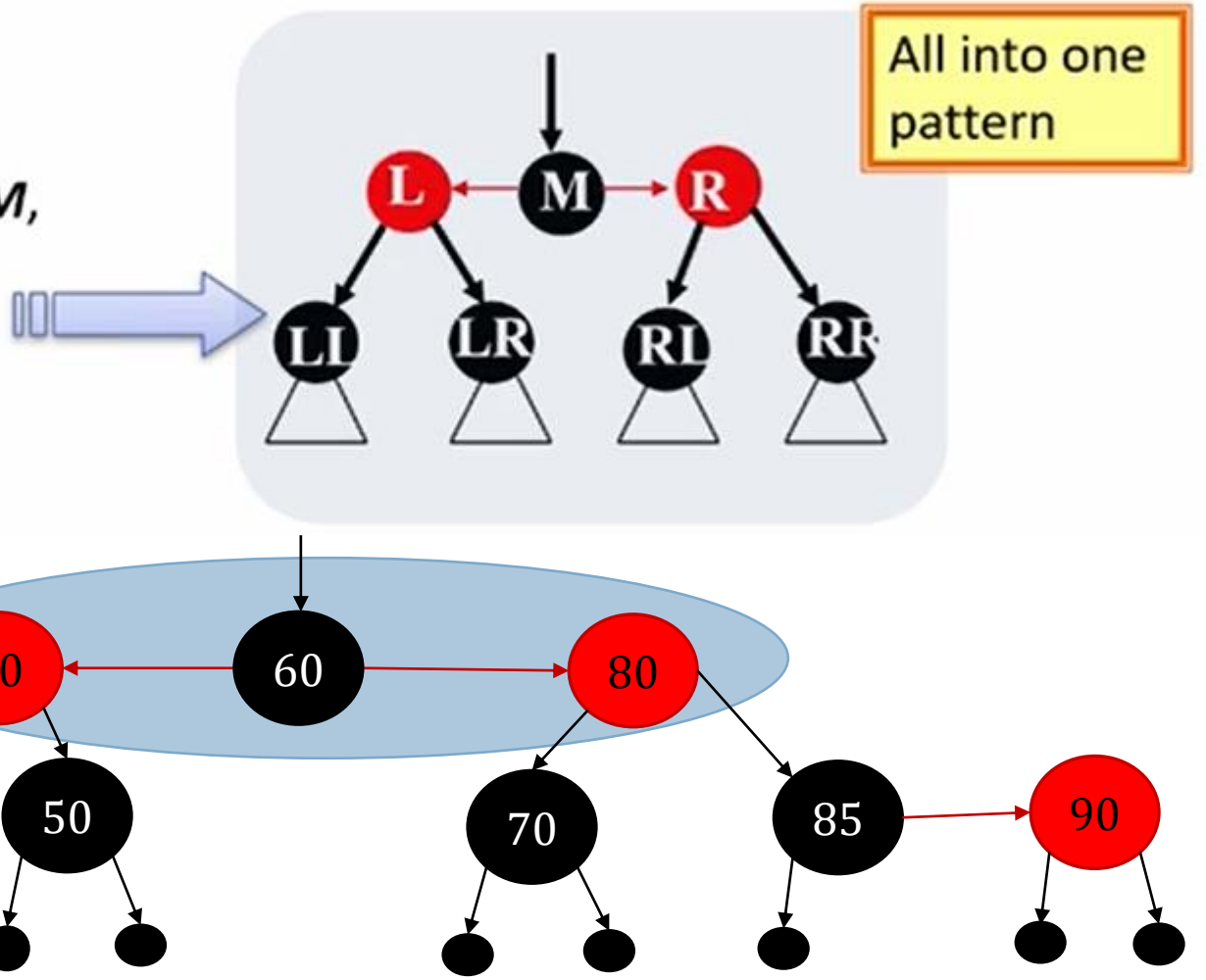


3-node Critical Cluster

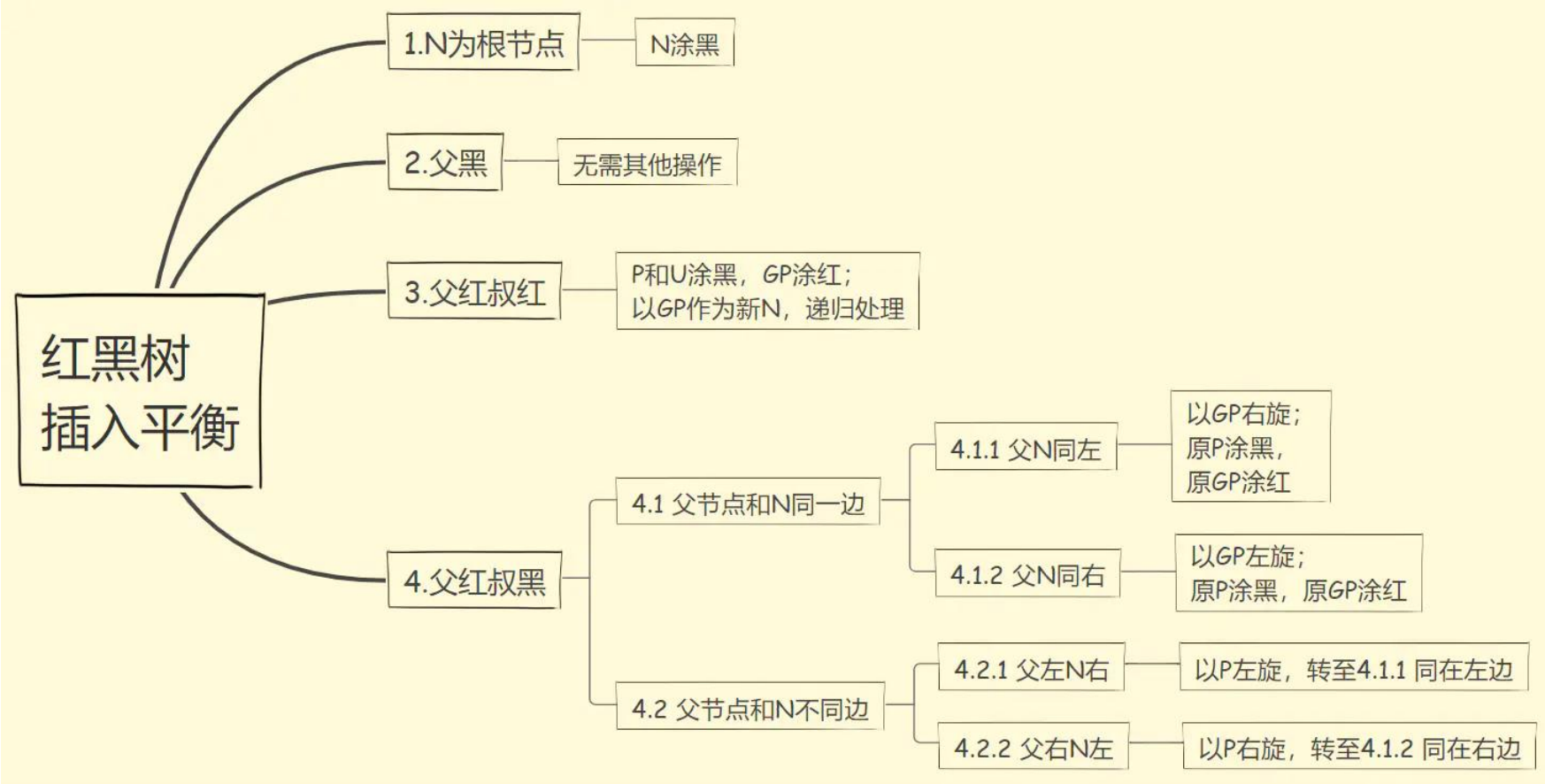


3-node Critical Cluster

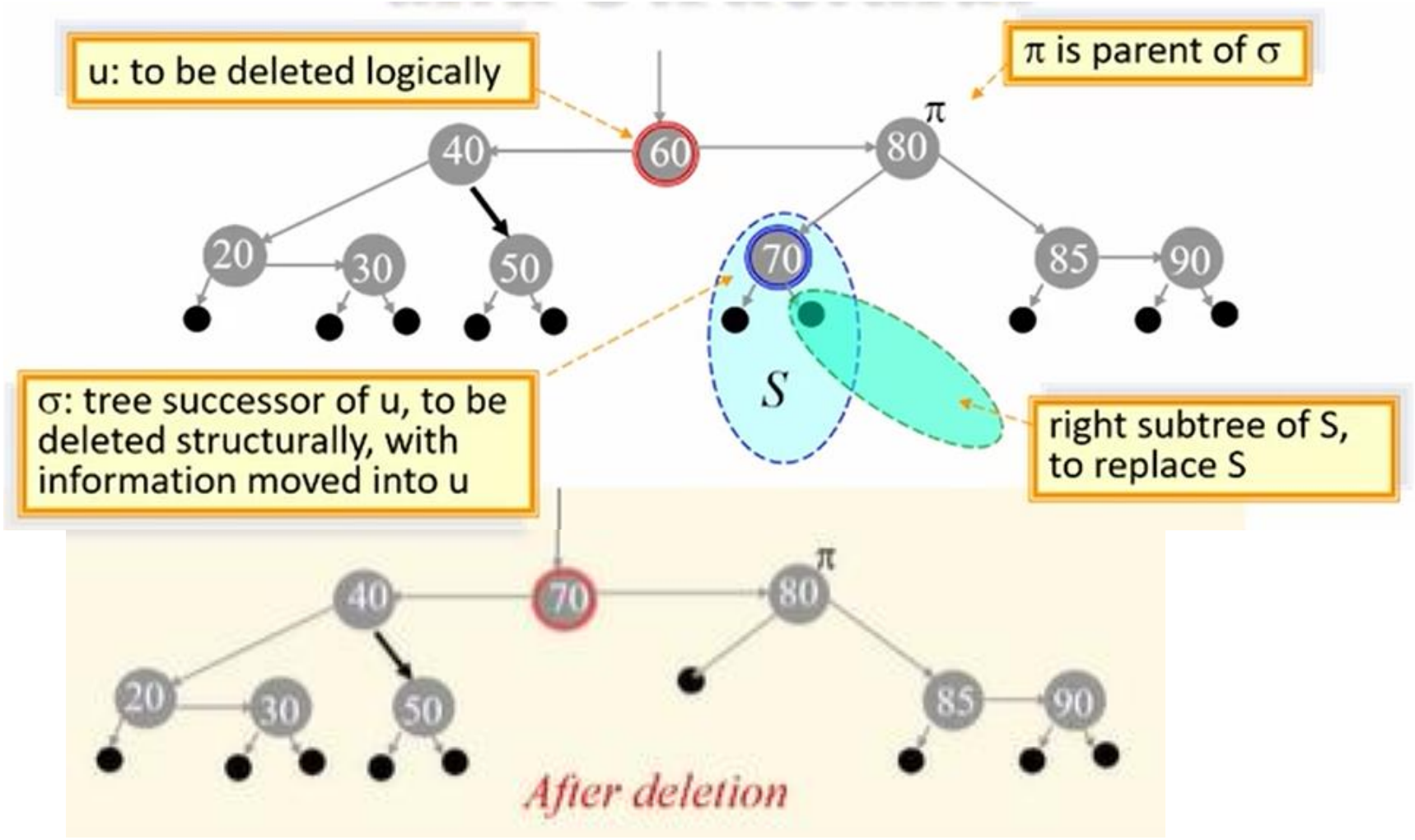
Root of the critical cluster is changed to *M*, and the parentship is adjusted accordingly



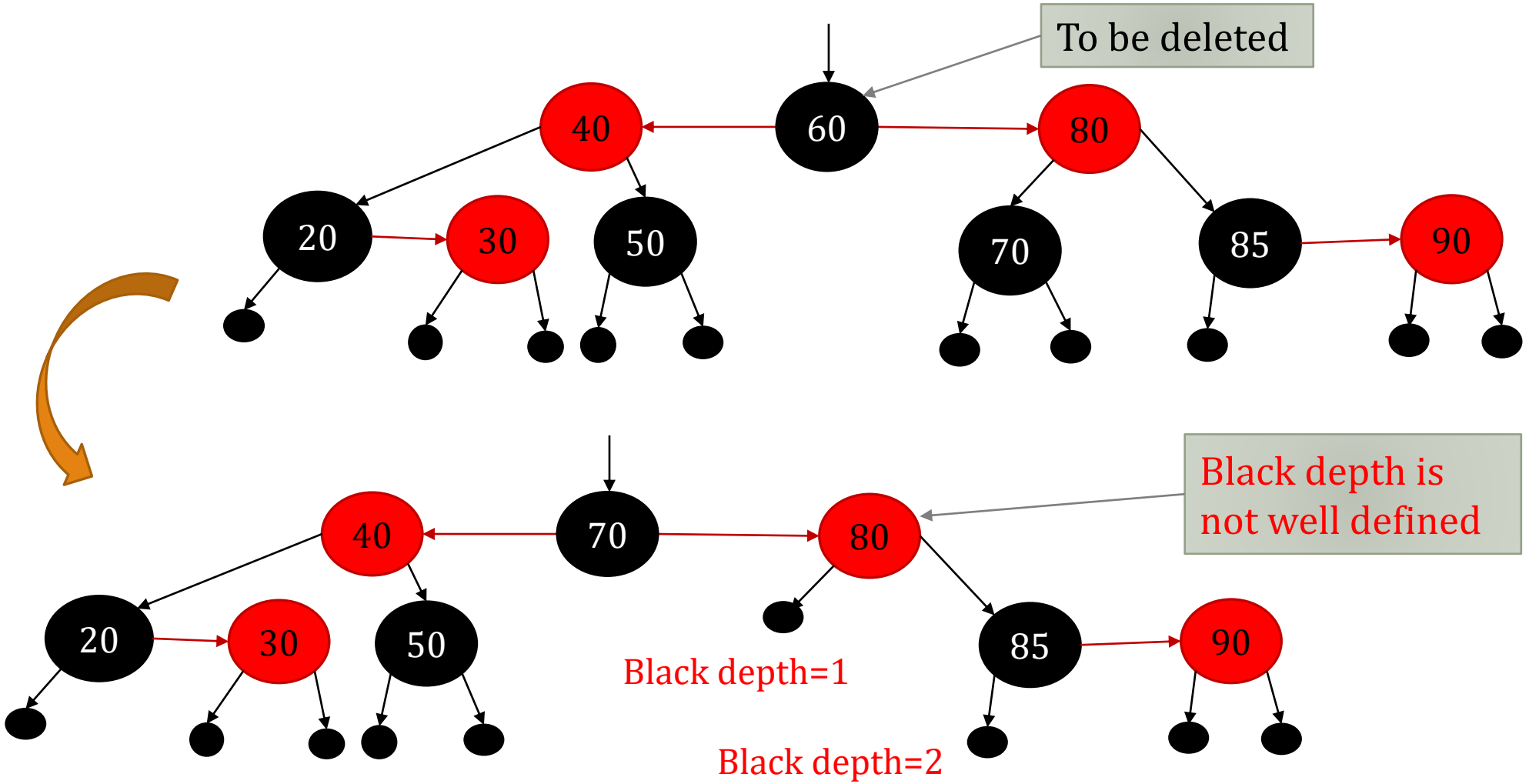
红黑树的插入


红黑树
插入平衡
1.N为根节点
N涂黑
2.父黑
无需其他操作
3.父红叔红
P和U涂黑，GP涂红；
以GP作为新N，递归处理
4.父红叔黑
4.1 父节点和N同一边
4.1.1 父N同左
以GP右旋；
原P涂黑，
原GP涂红
4.1.2 父N同右
以GP左旋；
原P涂黑，原GP涂红
4.2 父节点和N不同边
4.2.1 父左N右
以P左旋，转至4.1.1 同在左边
4.2.2 父右N左
以P右旋，转至4.1.2 同在右边

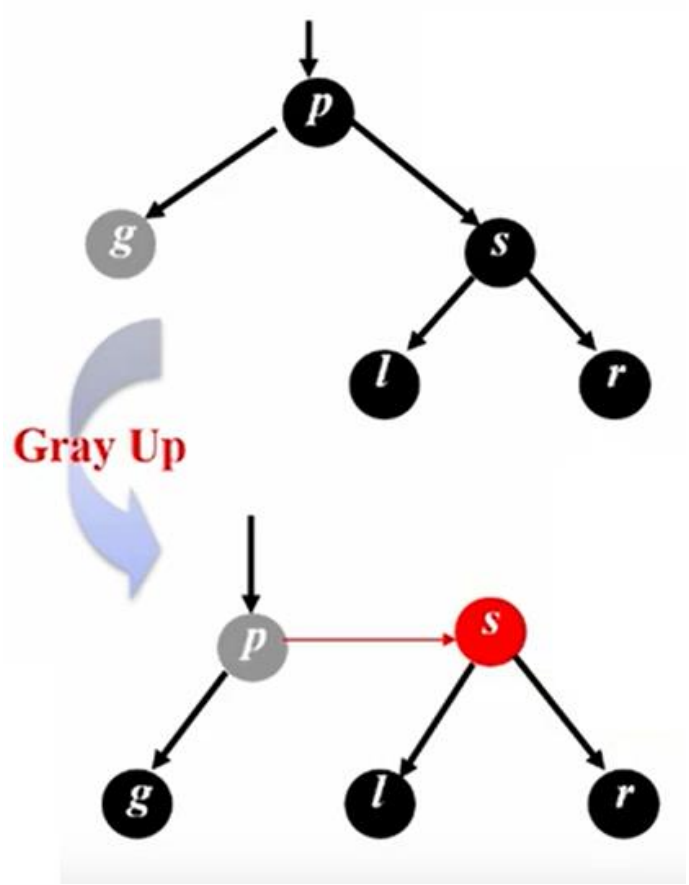
红黑树—逻辑和结构删除


u: to be deleted logically
π is parent of σ
σ: tree successor of u, to be deleted structurally, with information moved into u
right subtree of S, to replace S
S
After deletion

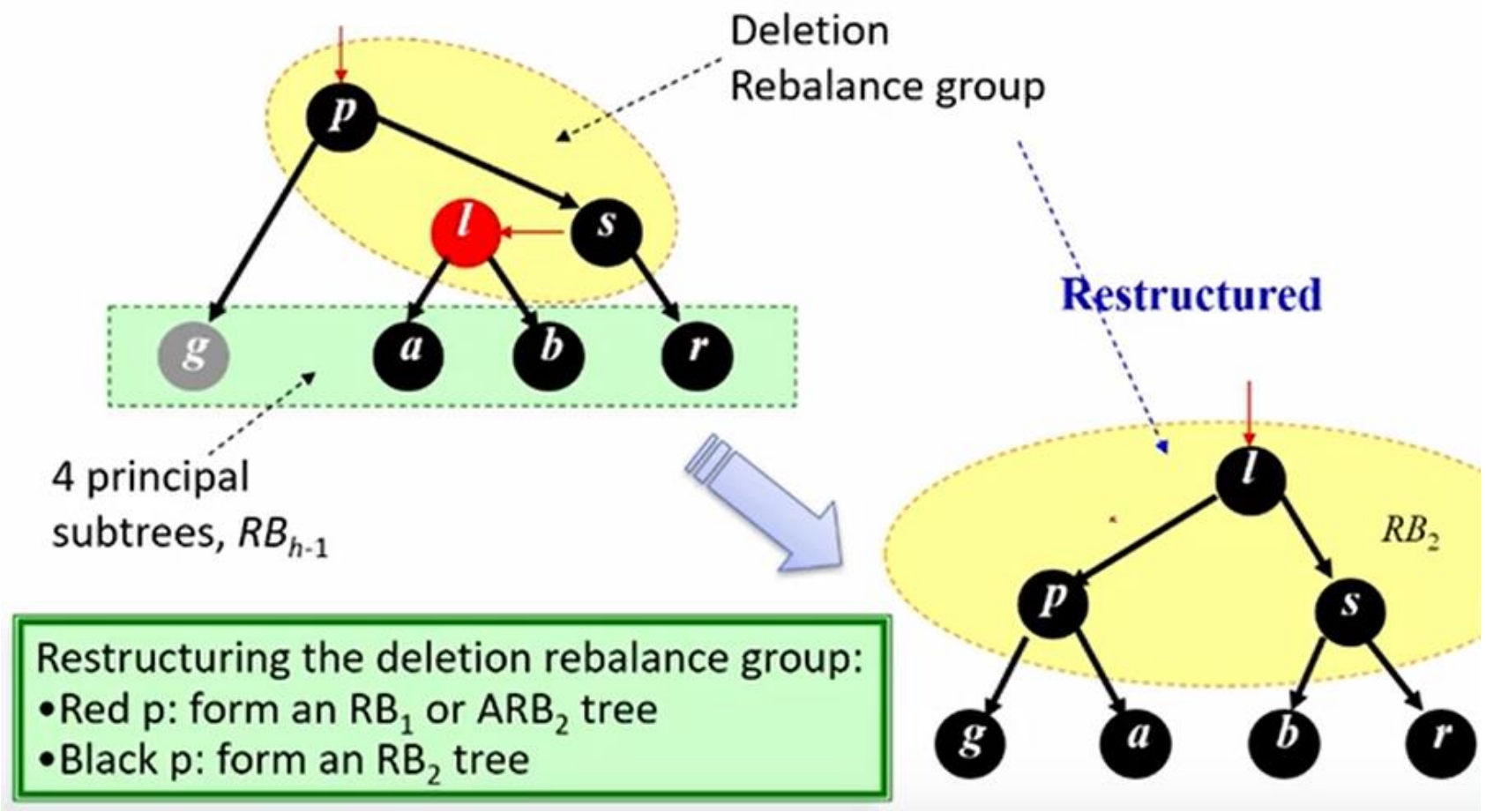
红黑树的删除


To be deleted
60
40
80
20
30
50
70
85
90
Black depth is not well defined
70
40
80
20
30
50
85
90
Black depth=1
Black depth=2

红黑树的删除—propagation



红黑树的删除—without propagation



扩展阅读

- 彻底理解红黑树（二）之 插入
- 彻底理解红黑树（三）之 删除

Outline

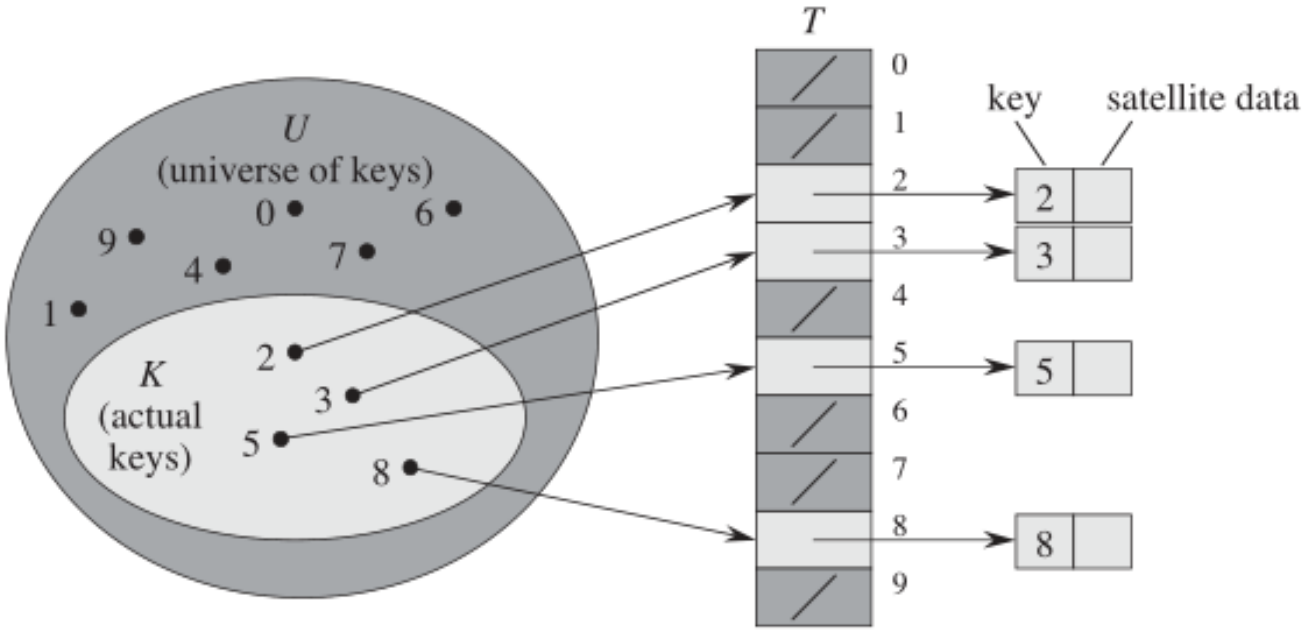
- 哈希表
- 冲突消解技术
  - 封闭寻址
  - 开放寻址
- 平摊分析

查找代价

- 蛮力方法：$O(n)$
- 平衡二叉搜索树：$O(\log n)$
- 哈希查找，近似线性时间：
  - $O(1+\alpha)$

查找—蛮力方法

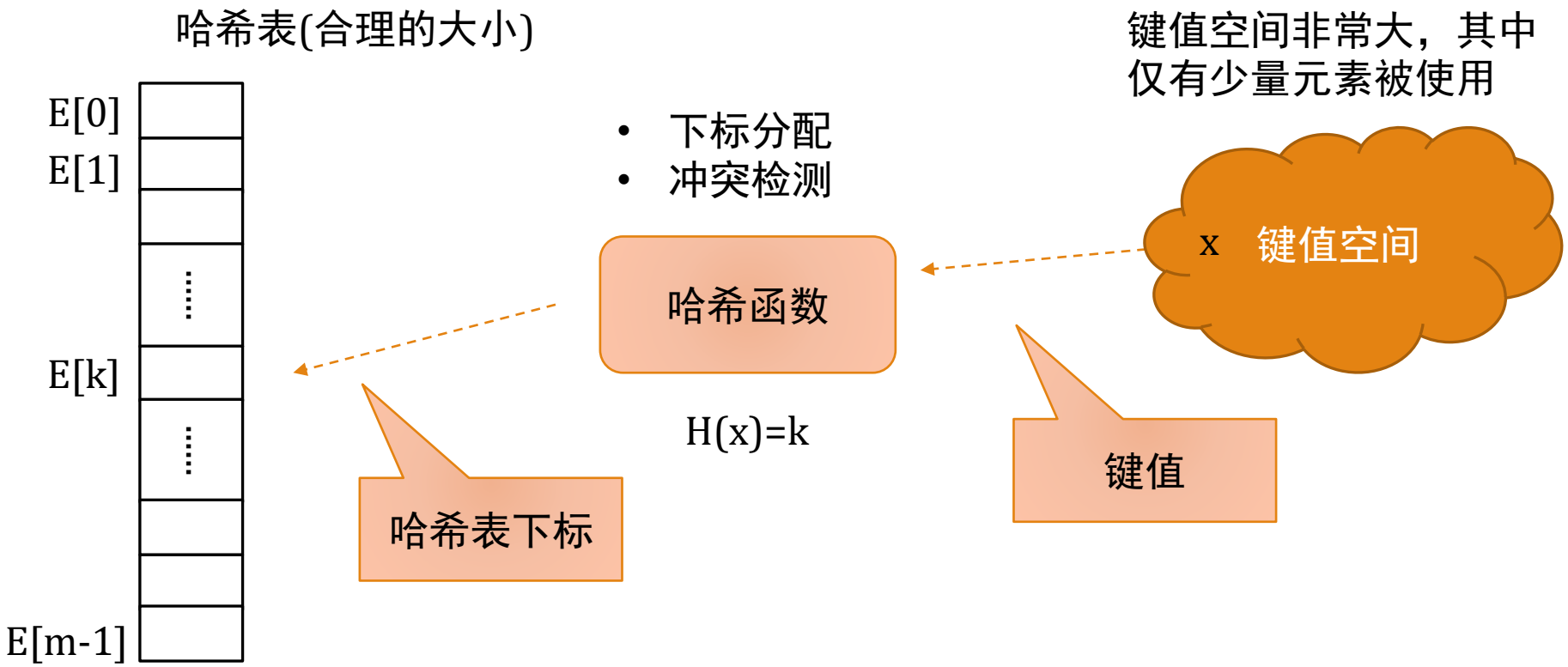
- 直接寻址表



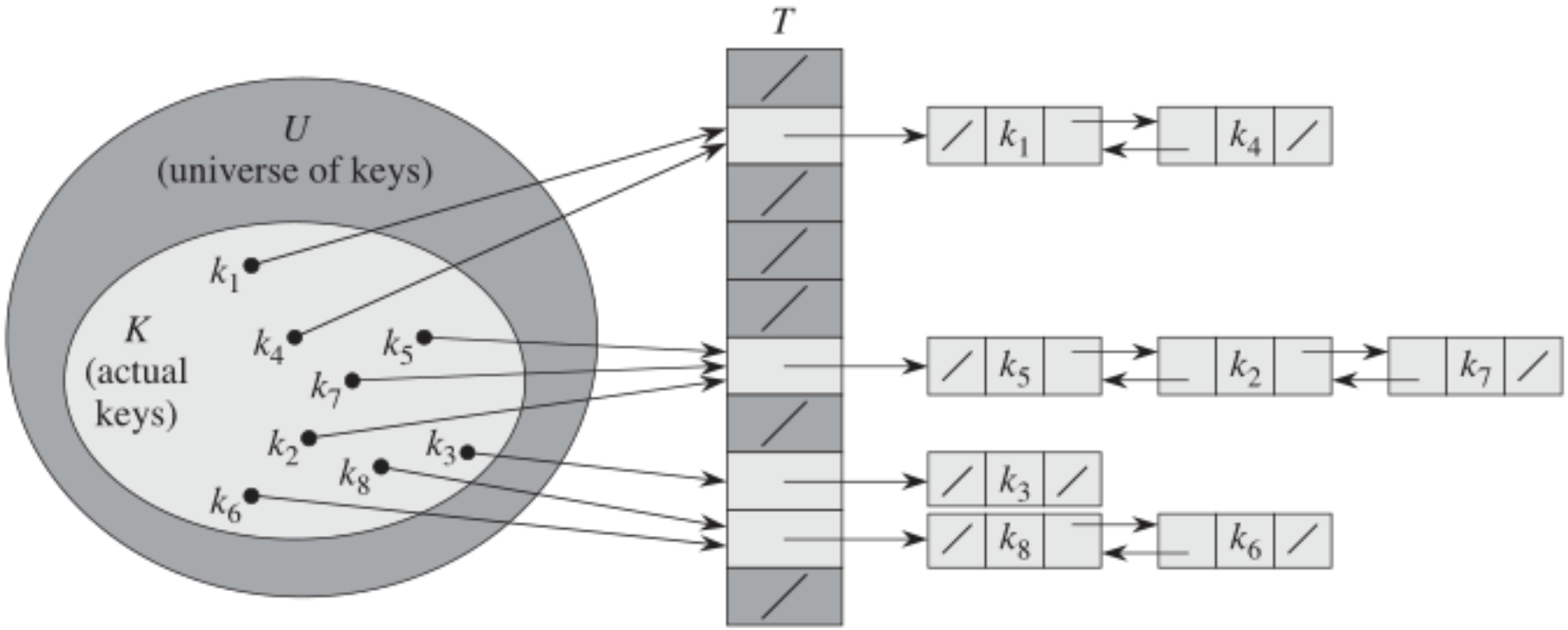
哈希表

- **哈希表**（Hash Table，也叫散列表），是根据键（Key）而直接访问在内存存储位置的数据结构。
- 哈希表通过计算一个**关于键值的函数**，将所需查询的数据映射到表中一个位置来访问记录，这加快了查找速度。
- 这个映射函数称做**哈希函数**，存放记录的数组称做哈希表。

哈希表


哈希表(合理的大小)
E[0]
E[1]
E[k]
E[m-1]
键值空间非常大，其中
仅有少量元素被使用
下标分配
冲突检测
哈希函数
H(x)=k
x 键值空间
哈希表下标
键值

封闭寻址冲突消解



封闭寻址分析

- 假设：简单均匀哈希
  - n 个元素等概率地被哈希到所有可能的m个位置之一
  - 对于 j=0,1,2,…,m-1，链表平均长度 E[j] 是 n/m
  - 负载因子 $\alpha = \frac{n}{m}$，反映了哈希表的“拥挤”程度

- 一次不成功查找的期望代价：
  - 计算哈希值
  - 遍历整个链表，没有找到
  - 总代价：O(1+n/m)

封闭寻址分析

- 对于一次成功的查找（假设 x_i 是第 i 个插入到哈希表的元素）
- 对于元素 x_i，被查找的概率是 $1/\mathrm{n}$
- 查找代价主要受与 x_i 在同一链表且在它前面的元素的影响
  - x_i 后面的元素有 $1/\mathrm{m}$ 的概率被哈希到 x_i 所在的链表
  - 插入在 x_i 前面元素个数的期望值： $\sum_{j=i+1}^{n} 1/m$

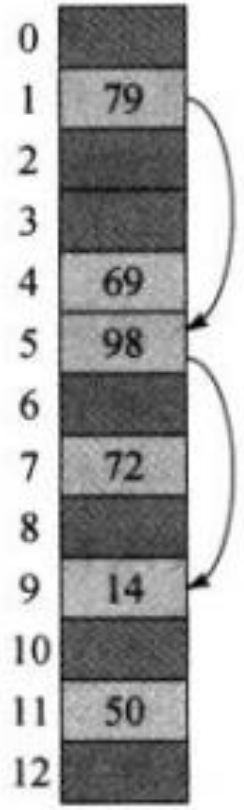
封闭寻址分析

- 对于封闭寻址冲突消解，一次成功查找的平均情况代价为 $\Theta(1+\alpha)$

$$
\begin{aligned}
E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right] &= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}E[X_{ij}]\right) \qquad \text{（由期望的线性性）} \\
&= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right) = 1+\frac{1}{nm}\sum_{i=1}^{n}(n-i) \\
&= 1+\frac{1}{nm}\left(\sum_{i=1}^{n}n-\sum_{i=1}^{n}i\right) = 1+\frac{1}{nm}\left(n^2-\frac{n(n+1)}{2}\right) \\
&= 1+\frac{n-1}{2m} = 1+\frac{\alpha}{2}-\frac{\alpha}{2n}
\end{aligned}
$$

开放寻址

- 所有元素存放在哈希表中
  - 没有使用链表
  - 负载因子不大于 1
- 冲突通过“rehashing”解决
  - 反复哈希，成功查找或者探测到一个空位
- 每次查找对应一个探测序列
  - 哈希表所有m个位置的一个排列



开放寻址分析

- 假设：每个键值的探测序列均匀等概率地对应到所有可能 m! 种排列中的某一个

- 一次不成功的平均探测代价：
  - 首先分析探测次数不少于 i 的概率，等价于前 i-1 次探测都探测到非空的位置
  - 根据均匀等概率假设，第一次探测到一个非空位置的概率为 $\frac{n}{m}$，第二次探测到一个非空位置的概率为 $\frac{n-1}{m-1}$
  - 探测次数不少于 $i(i \geq 2)$的概率是：

$$\frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdots \frac{n-i+2}{m-i+2} \leq \left(\frac{n}{m}\right)^{i-1} = \alpha^{i-1}$$

开放寻址分析

- 探测次数的期望值满足：

$$E[X]=\sum_{i=1}^{\infty}\Pr\{X\geqslant i\}\leqslant\sum_{i=1}^{\infty}\alpha^{i-1}=\sum_{i=0}^{\infty}\alpha^{i}=\frac{1}{1-\alpha}$$ ■

$1/(1-\alpha)=1+\alpha+\alpha^2+\alpha^3+\cdots$的这个界有一个直观的解释。无论如何，总要进行第一次探查。第一次探查发现的是一个已占用的槽时，必须要进行第二次探查，进行第二次探查的概率大约为α。前两次探查所发现的槽均是已占用时，需要进行第三次探查，进行第三次探查的概率大约为α^2，等等。

开放寻址分析

- 一次成功的查找：
  - 每个元素被查找的概率为 1/n
  - 成功查找到 x_{i+1}，相当于查找一个有 i 个元素的表，并且查找失败
  - 代价是 $\frac{1}{1-\frac{i}{m}} = \frac{m}{m-i}$
- 成功查找代价的期望值：

$$\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} = \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i} = \frac{1}{\alpha}\sum_{k=m-n+1}^{m}\frac{1}{k} \leqslant \frac{1}{\alpha}\int_{m-n}^{m}(1/x)\mathrm{d}x$$

$$= \frac{1}{\alpha}\ln\frac{m}{m-n} = \frac{1}{\alpha}\ln\frac{1}{1-\alpha}$$

Outline

- 平摊分析
  - MultiPop 栈
  - 二进制计数器
  - 数组扩充

平摊分析

- 对于哈希表的n次插入操作，单个操作的代价最高是O(n)，所以总代价的上界是 $O(n^2)$
- 更紧的上界，数组的扩充只会发生在数组大小为2的幂时，因而代价满足：

$$W(n) = \sum_{i=1}^{n} c_i$$

$$\leq \underbrace{n}_{\text{插入1个元素的代价}} + \underbrace{\sum_{j=0}^{\log n} 2^j}_{\text{数组扩张的代价}}$$

$$< n + 2n$$

$$= 3n$$

平摊分析

- 发现“廉价”操作和“昂贵”操作之间的内在联系，并通过这一联系将昂贵操作的代价“分摊”到廉价操作之上，进而得到一个更紧的上界。

- **实际代价C_{act}：**
  - 每个操作实际的执行代价。一般根据操作的实际代价将它们分为昂贵操作和廉价操作。
- **记账代价C_{acc}：**
  - 对于廉价操作，为其计算一个正的记账代价；对于昂贵操作，为其计算一个负的记账代价。
- **平摊代价C_{amo}：**
  - $C_{amo} = C_{act} + C_{acc}$

平摊分析

- 廉价操作的记账代价设计：设计一个正的记账代价，其原理是预先多计算一点代价。

- 昂贵操作的记账代价设计：设计一个负的记账代价，其原理是前面已经攒了一些钱（记账代价），此时使用攒好的钱。

- **保证分析的上界是正确的**：所以操作的记账代价的总和必须永远是非负的。

MultiPop 栈

- PUSH: 将一个元素压倒栈中。
- POP-ALL: 将栈中所以元素全部出栈。

操作	C_{amo}	C_{act}	C_{acc}
PUSH	2	1	1
POP-ALL	0	k	-k

- 任意n个栈操作的序列的代价总是不超过 2n= O(n)

二进制计数器

计数器值	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	总代价
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

二进制计数器

- 有进位的增1和无进位的增1

$$C_{acc}(\text{有进位增1}) = \underbrace{-k}_{k\text{个1变成0的记账代价}} + \underbrace{1}_{\text{最高位从0变成1的记账代价}}$$

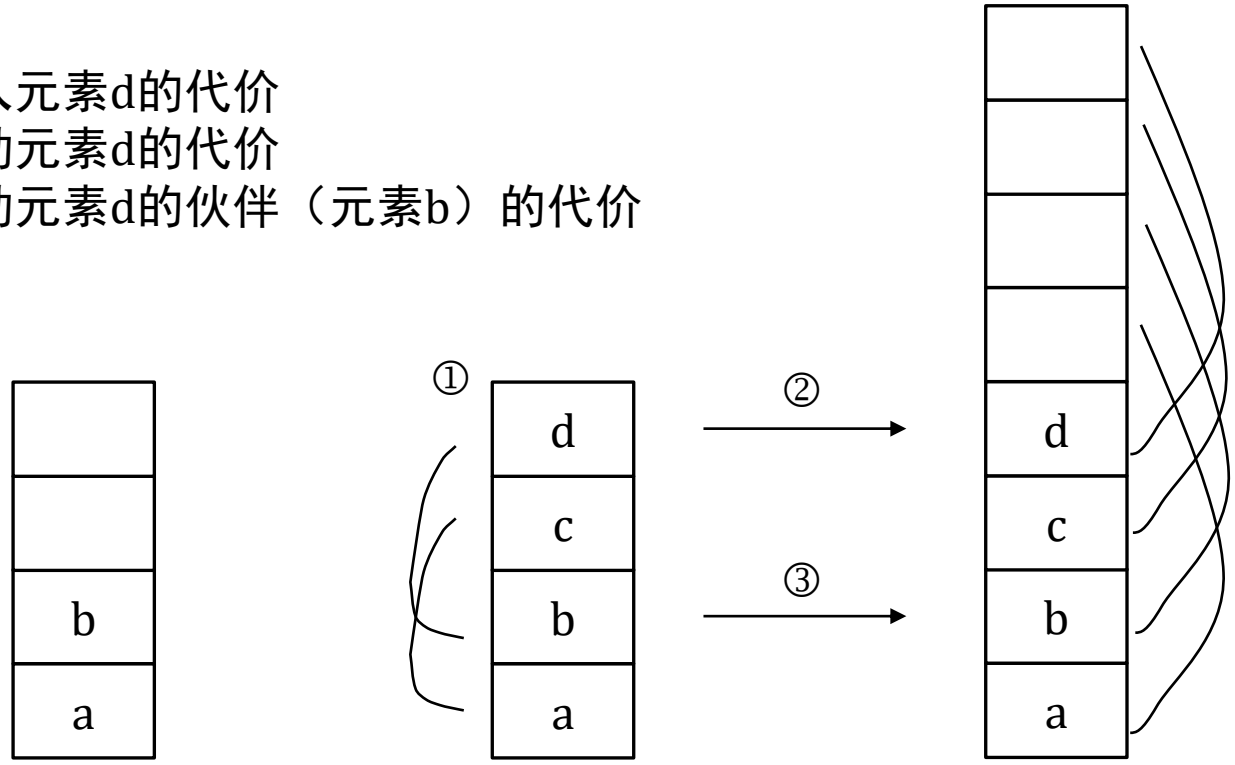
操作	C_{amo}	C_{act}	C_{acc}
无进位的增1	2	1	1
有进位的增1	2	k+1	-k+1

- 只针对比特位的操作：

操作	C_{amo}	C_{act}	C_{acc}
置1	2	1	1
置0	0	1	-1

数组扩充

① 插入元素d的代价
② 挪动元素d的代价
③ 挪动元素d的伙伴（元素b）的代价



刚挪入新空间　　新空间插满　　再挪入新空间

数组扩充

操作	C_{amo}	C_{act}	C_{acc}
普通插入	3	1	2
扩充插入	3	k+1	-k+2

数组扩充的平摊分析

Outline

- 动态等价关系
- 并查集
  - 普通并+普通查
  - 加权并+普通查
  - 加权并+路径压缩查

动态等价关系

- Kruskal 算法，贪心策略：
  - 新加入一条最小权重的边
  - 判断是否成环

- 是否成环等价于判断节点间(在局部最小生成树)的连通关系
- 节点间的连通关系是一个等价关系

- **在节点间等价关系动态变化的同时，高效地判断两个元素之间是否具有等价关系。**

动态等价关系

- 等价关系具有自反、对称、传递的二元关系的性质。
- 设 R 是集合 A 上的一个二元关系，若R满足：
  - 自反性：∀ a ∈A, => (a, a) ∈ R
  - 对称性：(a, b) ∈R∧ a ≠ b => (b, a)∈R
  - 传递性：(a, b)∈R,(b, c)∈R =>(a, c)∈R

动态等价关系

- 考虑 n 个元素 $a_1, a_2, \ldots, a_n$，在某种等价关系下被分为若干个等价类
- 初始时，每个元素单独成为一个等价类
- 两类操作：
  - FIND(a_i)：返回 a_i 所在的等价类
  - UNION(a_i, a_j)：将a_i, a_j 所在的等价类合并成一个等价类

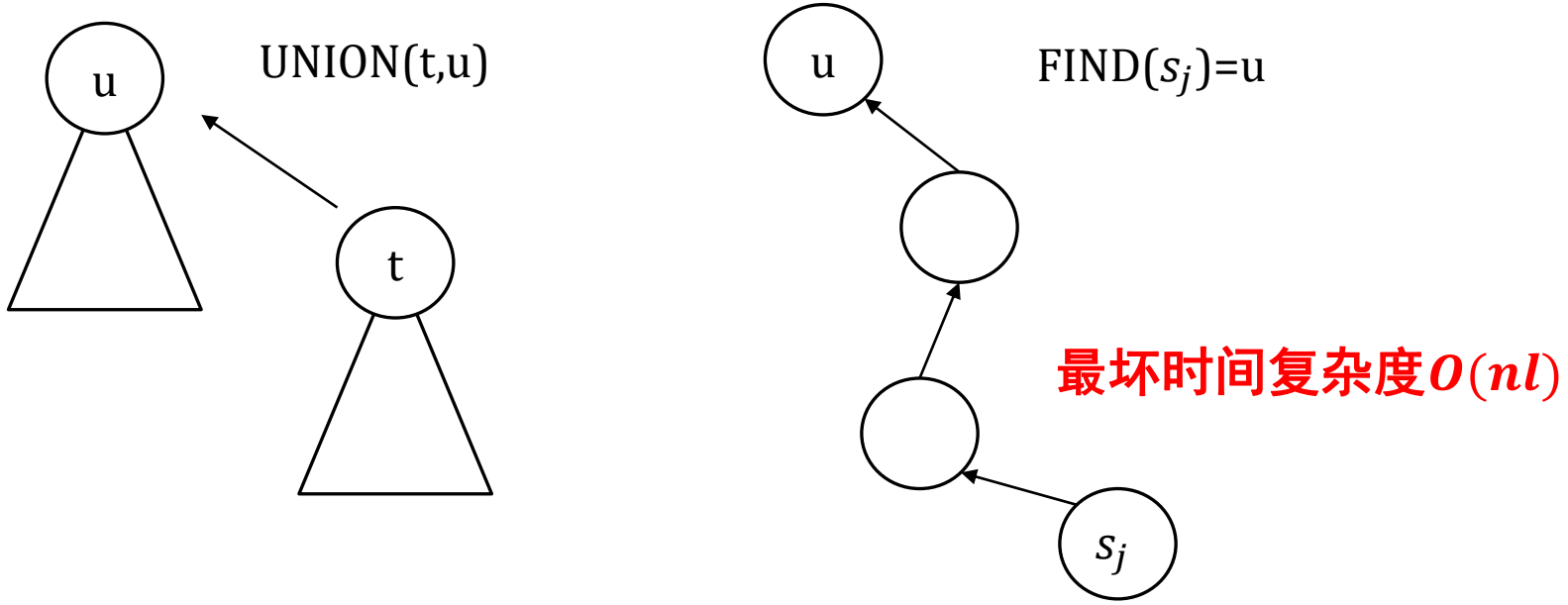
Kruskal—基于并查集的实现

- 判断加入边uv是否成环，只需判断点 u 和 v 是否在同一个等价类。
- 如果是，则加入 uv 必然成环
- 否则必然不成环，加入uv之后，需要将这两个点所在的等价类合并成一个。

- 两种朴素的实现方法
  - 基于矩阵：布尔矩阵表示二元关系，空间开销 $\Theta(n^2)$，最坏情况时间复杂度 $O(nl)$
  - 基于数组：每个位置存放元素所在等价类的代表元，空间开销 $\Theta(n)$，最坏情况时间复杂度 $O(nl)$

基于根树的实现

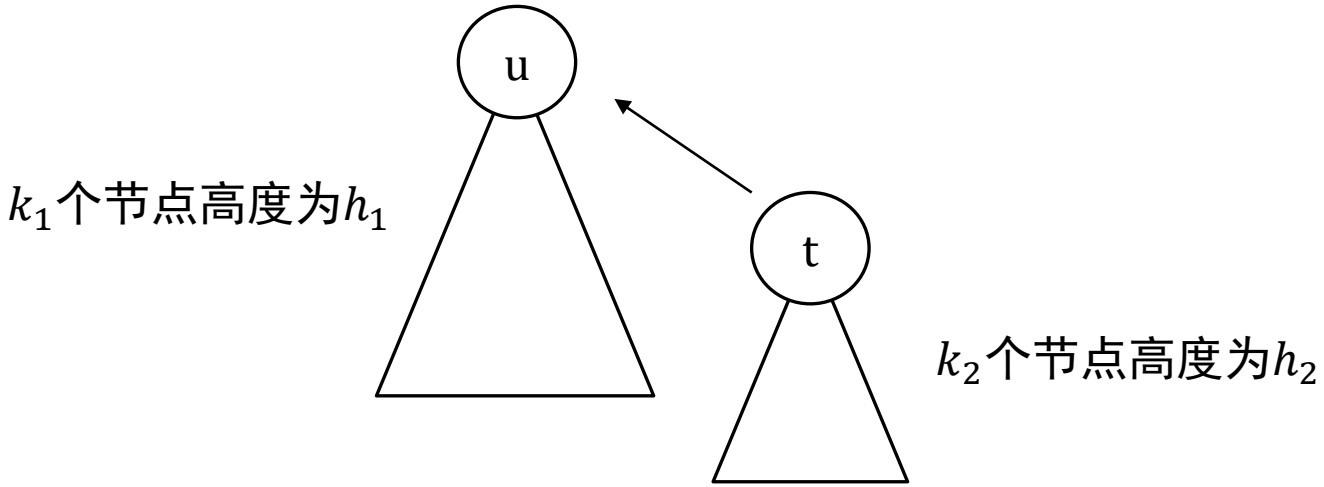
- 根节点为等价类的代表元
- 每棵子树的所有节点对应于一个等价类



图：并查集的基础实现

加权“并”+普通“查”

- 加权并：将节点数更少的树挂到节点数更多的树的根节点。



图：加权并的实现

加权“并”+普通“查”

- 引理：初始时，每个元素成为一个等价类。基于加权并来实现并查集的并时，包含 k 个节点的树，它的高度至多不超过 $\lfloor \log k \rfloor$

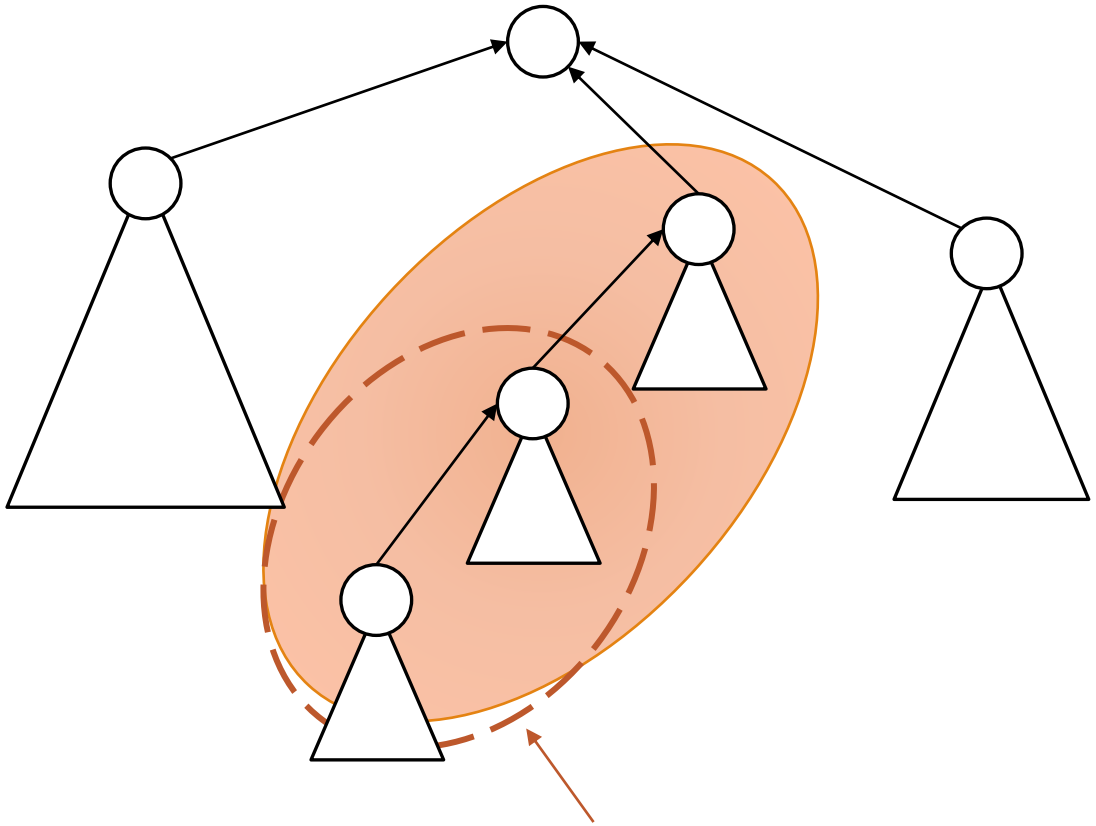
- 证明：数学归纳法
  - base case: k=1, the height is 0.
  - by inductive hypothesis:
    - $h_1 \le \lfloor \lg k_1 \rfloor$, $h_2 \le \lfloor \lg k_2 \rfloor$
  - h=max(h1, h2+1), k=k1+k2
    - if $h=h_1$, $h \le \lfloor \lg k_1 \rfloor \le \lfloor \lg k \rfloor$
    - if $h=h_2+1$, note: $k_2 \le k/2$
      so, $h_2+1 \le \lfloor \lg k_2 \rfloor + 1 \le \lfloor \lg k \rfloor$

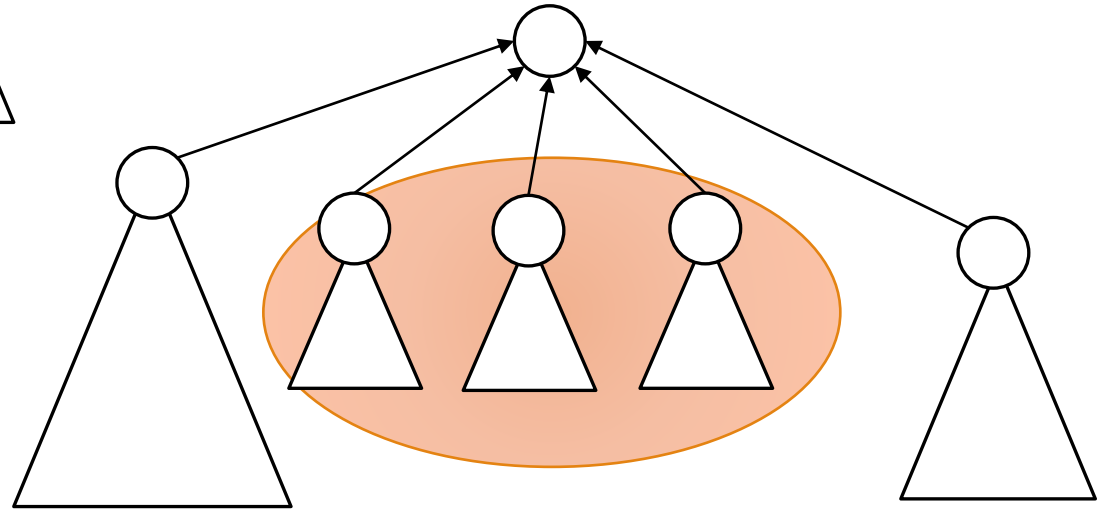
加权“并”+普通“查”

- 定理：采用加权“并”和普通“查”，对于 n 个元素的并查集与长度l的并查程序，最坏情况下代价为 $O(n + l \log n)$

- 初始代价： $O(n)$
- UNION 操作： $O(n)$
- FIND 代价： $O(l \log n)$

加权“并”+路径压缩“查”





加权“并”+路径压缩“查”

- 压缩过的节点后续被频繁查找，则路径压缩就能有效改进并查集的效率。
- 定理：采用加权“并”和路径压缩“查”，对于 n 个元素的并查集与长度l 的并查程序，最坏情况下代价为 $\boldsymbol{O((n+l)log^{*}n) \approx O(n+l)}$
- O(ln) => O(l log n) => O(n+l)

Thanks