

第1讲 基础数据结构

刘汝佳

目录

- 线性结构
- 二叉堆
- 并查集
- 哈希表
- 应用举例

一、线性结构

基础知识

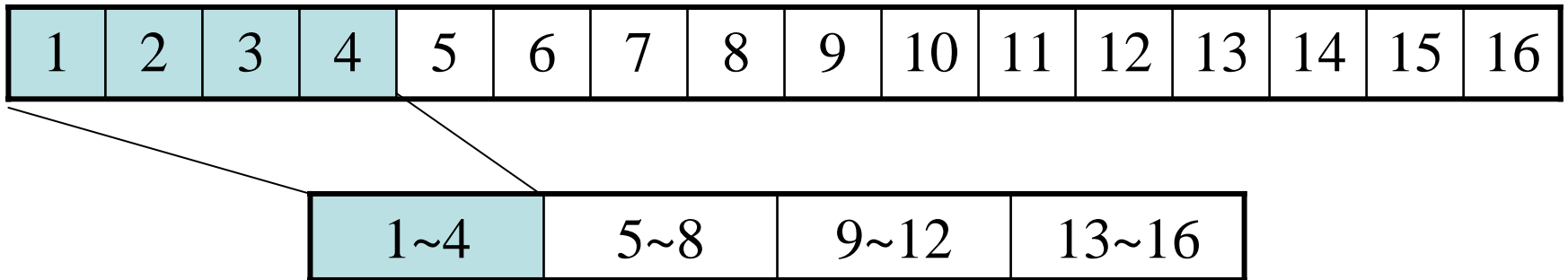
- 数组
- 带头结点的双链表
 - Head结点: 虚拟头结点
 - First结点: 第一个有实际内容的结点
- 队列: 循环队列与Open-Close表

例1. 最小值

- 实现一个 n 个元素的线性表 A 。每次可以修改其中一个元素，也可以询问闭区间 $[p, q]$ 中元素的最小值。
- $1 \leq n, m \leq 100000$

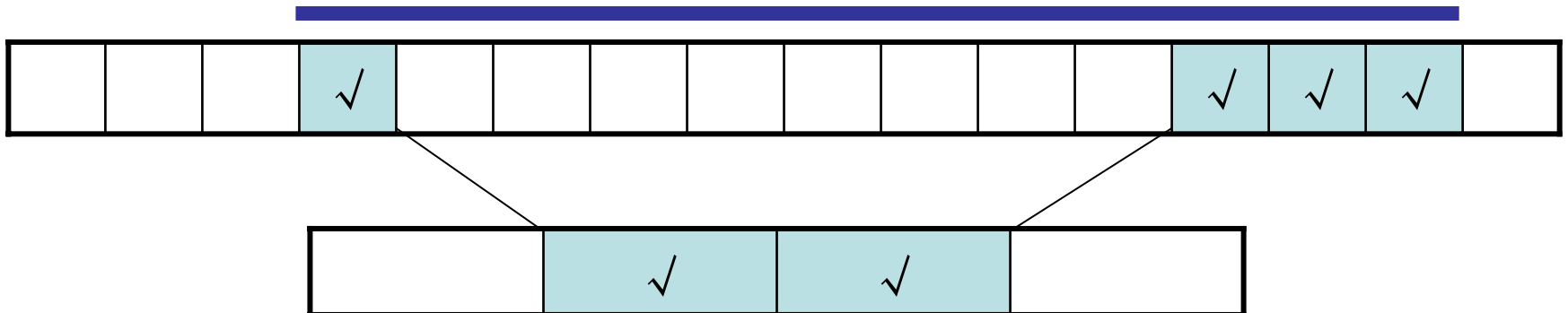
分析

- 利用二级检索的思想
 - 设块长为 L , 则一共有 n/L 个块
 - 维护数组 B , 保存每个块的最小值
- $\text{Modify}(x, y)$
 - $A[x] = y$ $O(1)$
 - 重算 x 所在块的最小值(更新 B) $O(L)$



Min操作

- $\text{Min}(a, b)$
 - 把区间 $[a, b]$ 分成若干部分
 - 完整块: 一共最多 n/L 个块 $O(n/L)$
 - 非完整块: 首尾各最多 $L-1$ 个元素 $O(L)$
- 每次操作时间复杂度: $O(n/L+L)$
 - 设 $L=O(n^{1/2})$ 则渐进时间复杂度为 $O(n^{1/2})$



例2. 最接近的值

- 给一个 n 个元素的线性表 A ，对于每个数 A_i ，找到它之前的数中，和它最接近的数。即对于每个 i ，计算

$$C_i = \min\{|A_i - A_j| \mid 1 \leq j < i\}$$

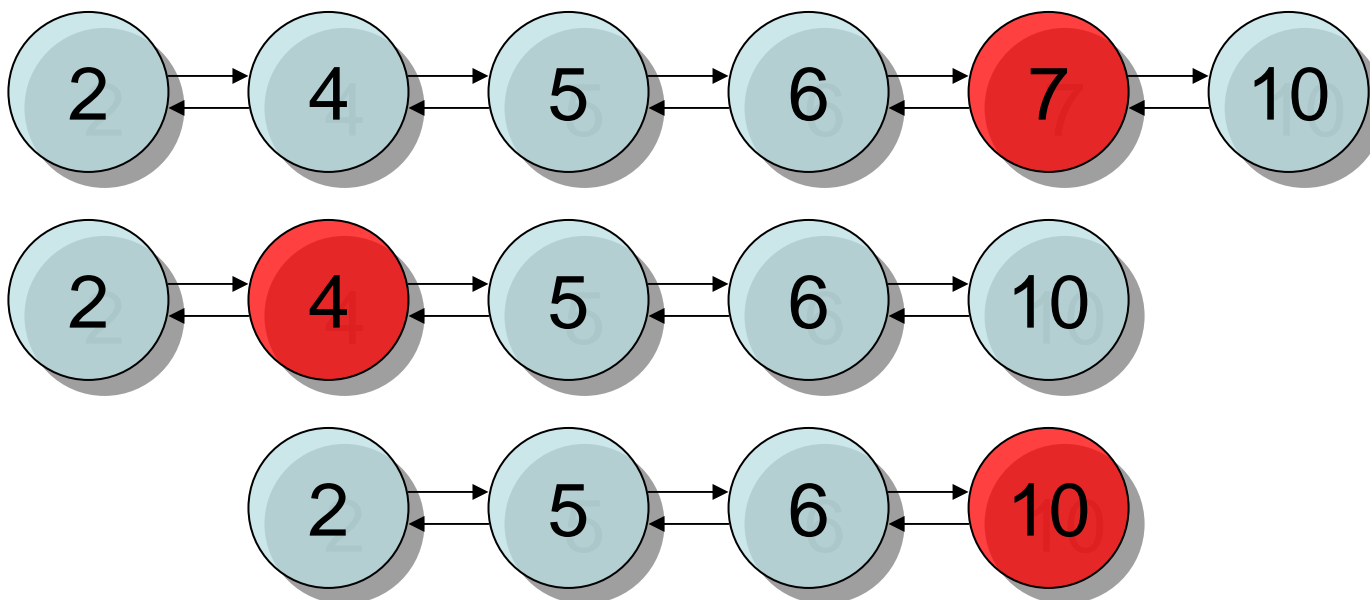
- 规定 $C_1 = 0$ 。

分析

- 问题的关键: 你只需要解决离线问题
 - 在线问题: 每输入一个 A_i , 立刻输出 C_i
 - 离线问题: 在给出任何输出之前得到所有 A_i
- 预处理: 用 $O(n \log n)$ 时间给所有 A_i 排序
 - 很快就会学到了☺
- 主算法
 - 根据从小到大的顺序构造双向链表
 - 依次计算 C_n, C_{n-1}, \dots, C_1 在线问题可能么?

主算法

- $A = \{2, 6, 5, 10, 4, 7\}$, 依次计算 C_6, C_5, C_4
 - 每次计算 C_i 时, 链表中恰好是计算 C_i 需要的元素
 - 计算 C_i 只需比较两个元素, 然后删除 A_i $O(1)$



例3. 移动窗口

- 有一个长度为 n 的数组，还有一个长度为 $k \leq n$ 的窗口。窗口一开始在最左边的位置，能看到元素 $1, 2, 3, \dots, k$ 。每次把窗口往右移动一个元素的位置，直到窗口的右边界到达数组的元素 n 。

窗口内元素位置	窗口最小值	窗口最大值
[2 7 1] 6 -3 2 5 0 4	1	7
2 [7 1 6] -3 2 5 0 4	1	7
2 7 [1 6 -3] 2 5 0 4	-3	6
2 7 1 [6 -3 2] 5 0 4	-3	6
2 7 1 6 [-3 2 5] 0 4	-3	5
2 7 1 6 -3 [2 5 0] 4	0	5
2 7 1 6 -3 2 [5 0 4]	0	5

分析

- 考虑最小值。假设窗口从左往右移动

...	2	3	5	4	...
-----	---	---	---	---	-----

- 保存5是不明智的, 因为从**现在**开始一直到5离开窗口, 5始终被4“压制”, 永远都不可能成为最小值。删除**5不会影响**结果
- 启发**: 算最小值的**有用**元素形成一个递增序列, 最小值就是队首元素
- 关键**: 窗口右移操作

分析

- **窗口右移**：队首出队，新元素入队，然后在队列中删除它前面比它大的元素

...	3	3	5	7	9	11	12	4	...
-----	---	---	---	---	---	----	----	---	-----

- 实现
 - 用链表储存窗口内**有用**元素
 - 则窗口右移的时间和删除元素的**总**次数成正比
- 元素被删除后不会再次被插入，因此每个元素最多被删除一次，总次数为 $O(n)$ ，即：
算法时间总复杂度为 **$O(n)$**

例4. 程序复杂度

- 给出一段程序，计算它的时间复杂度。这段程序只有三种语句：
 - **OP <x>**: 执行一条时间耗费为 x 的指令。这里的 x 为不超过100的正整数。
 - **LOOP <x>**: 与**END**配套，中间的语句循环 x 次，其中 x 可以为规模 n ，也可以是不超过100的正整数。
 - **END**: 与**LOOP**配套，循环终止。
- 注意：**OP**语句的参数不能为变量 n ，而**LOOP**语句的参数可以为变量 n ，但不允许有其他名字的变量。这样，整个程序的时间复杂度应为 n 的多项式。你的任务就是计算并显示出这个多项式。

例4. 程序复杂度

- 输出仅包含一行，即时间复杂度的多项式。这个多项式应该按通常的方式显示处理，即不要输出 $0n$ 这样的项， n 项也不要输出为 n^1 ，等等。

```
OP 1
LOOP n
  LOOP 10
    LOOP n OP 7 END
  OP 1
END
END
```

$$70n^2+10n+1$$

分析

- 考虑特殊情况：没有变量 n 只有常数的情况
- 两种思路
 - 递归求解：不直接，附加开销大☹
 - 基于栈的扫描算法：实现简单明了，效率高☺
- 扫描过程（基本想法）
 - **LOOP x** 和**OP x** , 把语句入栈
 - **END**, 不断从栈里弹出语句，直到弹出**LOOP**
 - 弹出过程中累加操作数 x ，然后乘以循环次数 y
 - 把**OP $x*y$** 压入栈中，相当于用一条**OP**等效一个**LOOP**

分析

- 栈扫描算法的直接推广
 - 栈里的每个操作数不是数，而是多项式
 - 多项式加法，多项式与整数的乘积
 - 所有通过至少一个数据的同学都采用此法
- 问题
 - 多项式如何表示
 - 多项式操作的时间复杂度是怎样的？

复杂性

- 大部分数据涉及到高精度整数运算
- 高精度运算的复杂性
 - 写起来相对麻烦
 - 时间复杂度: $O(n^2)$ ($n \log n$ is possible, but...)
 - 空间复杂度
- 实际情况: 所有使用了高精度运算的同学在时间“爆”掉之前空间先“爆”掉
 - 每个数10000位, n 次数可达10000 (或更多)
 - 栈里面可以有10000个多项式, **$10^{12}=1T$!!!!!**

空间...空间...空间...

- 是否可以找到一个空间需求比较小的算法？**哪怕时间效率略一点都可以**
- 输出文件已经不小了，需要尽量少的保存中间结果，因此最好不要栈！
 - 只要有栈，最坏情况中间结果的空间大小就是输出大小乘以栈的最大深度，而输出...

complexity8.ans

complexity8.in

complexity9.ans

complexity9.in

17,374 KB

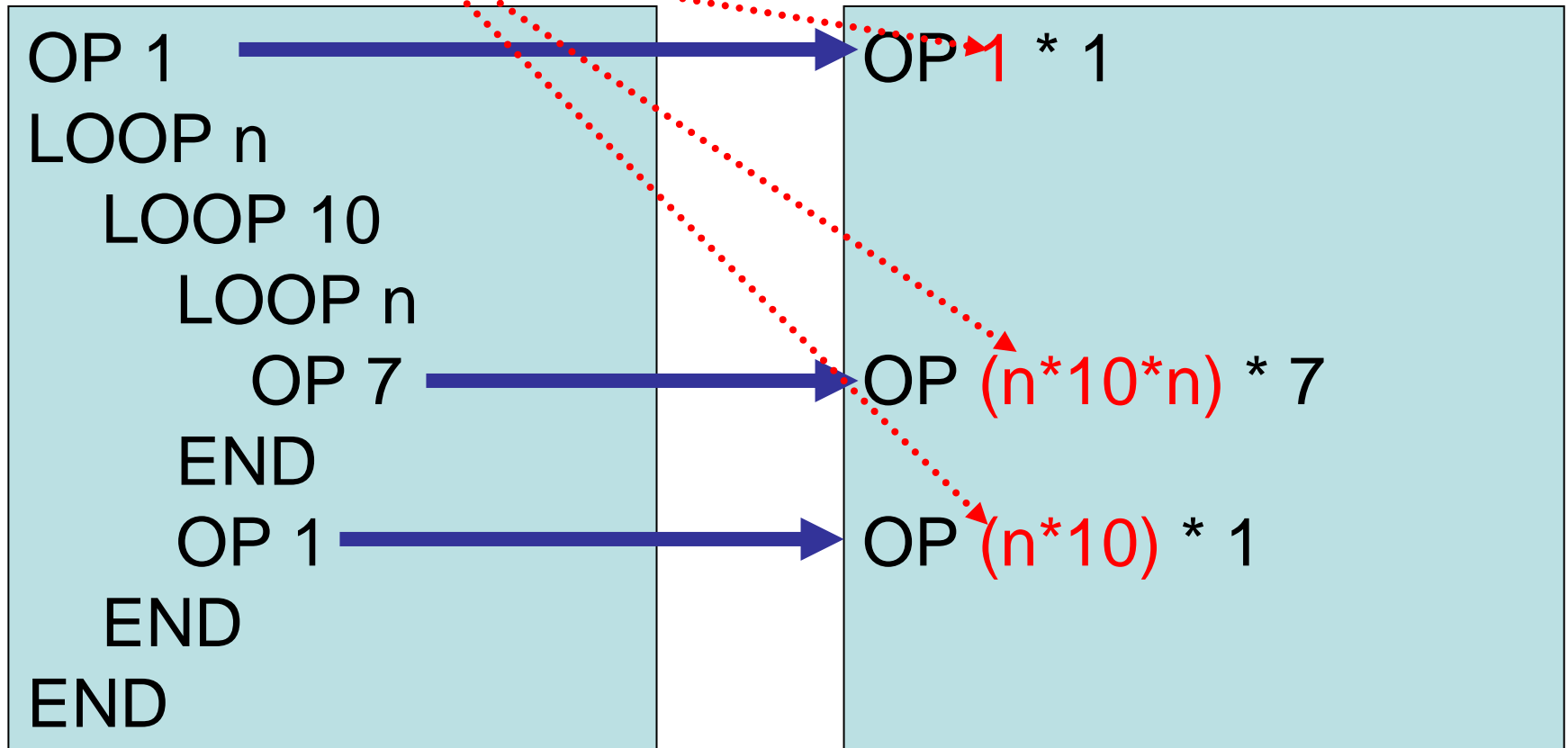
199 KB

19,402 KB

203 KB

“把括号展开”

- 先想办法去掉所有LOOP/END
- 借助当前乘数确定这次OP的最终效果



基本算法

- 设当前乘数为 m , 结果多项式为 ans
 - 初始化 $m = 1, ans = 0$
- 主循环：一条一条语句处理
 - 遇到LOOP x , $m = m * x$
 - 遇到END, $m = m / x$
 - 遇到OP x , $ans = ans + m * x$
- 数据结构：设 $m = a n^b$, 则用数对 (a, b) 表示 m , a 是高精度整数, b 是int类型
- 除了结果多项式 ans 和 a , 其他空间可忽略

二、二叉堆

堆

- 堆(heap)经常被用来实现优先队列(priority queue): 可以把元素加入到优先队列中, 也可以从队列中取出优先级最高的元素
 - **Insert(T, x):** 把x加入优先队列中
 - **DeleteMin(T, x):** 获取优先级最高的元素x, 并把它从优先队列中删除

堆的操作

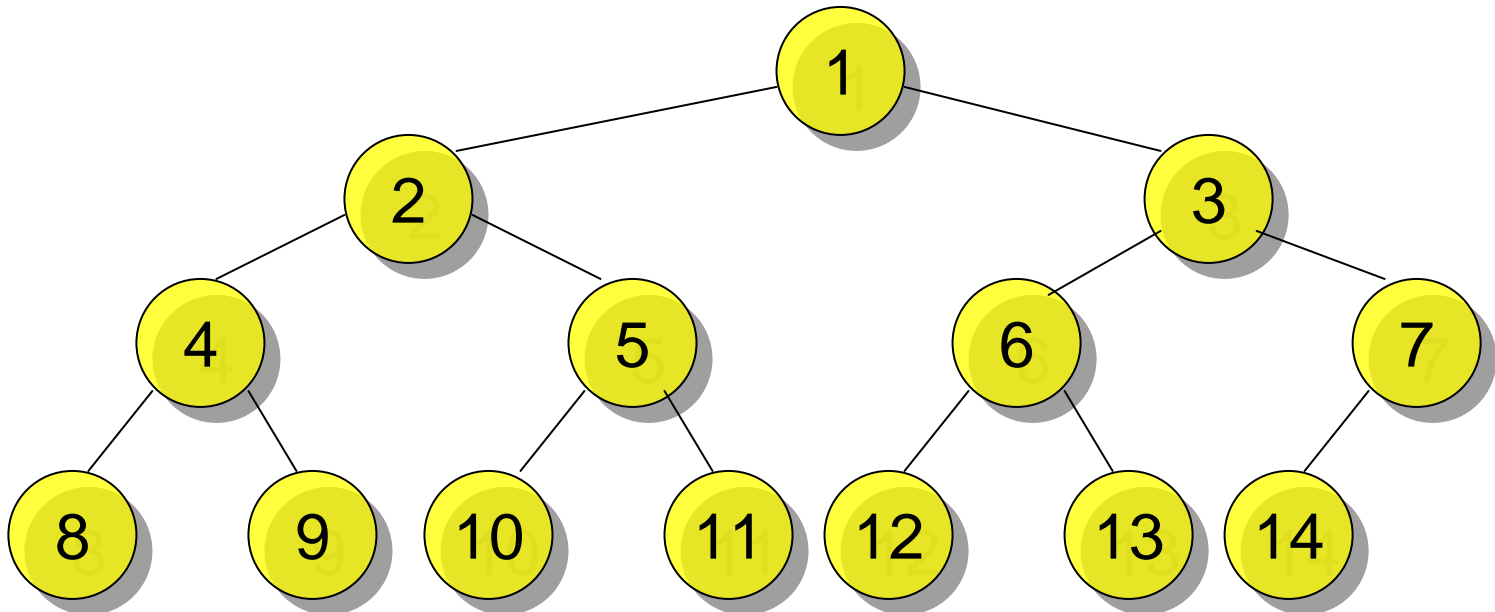
- 除了实现优先队列, 堆还有其他用途, 因此操作比优先队列多
 - **Getmin(T, x):** 获得最小值
 - **Delete(T, x):** 删除任意已知结点
 - **DecreaseKey(T, x, p):** 把x的优先级降为p
 - **Build(T, x):** 把数组x建立成最小堆
- 显然, 用堆很容易实现优先队列

堆的定义

- 堆是一个完全二叉树
 - 所有叶子在同一层或者两个连续层
 - 最后一层的结点占据尽量左的位置
- 堆性质
 - 为空, 或者最小元素在根上
 - 两棵子树也是堆

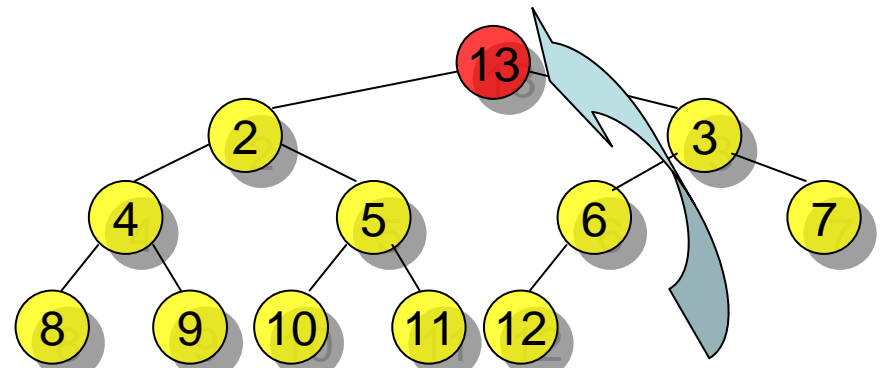
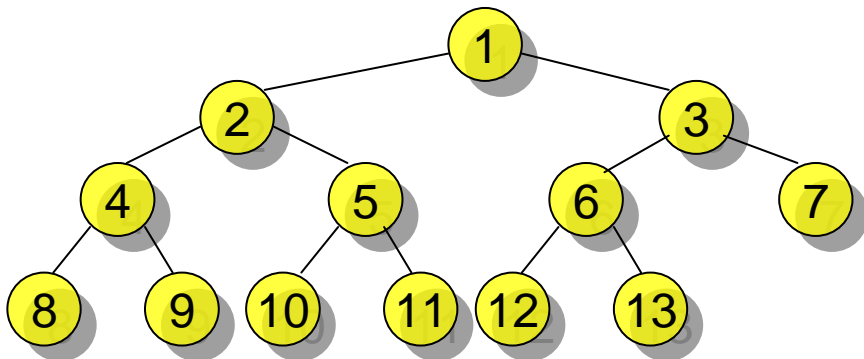
存储方式

- 最小堆的元素保存在`heap[1..hs]`内
 - 根在`heap[1]`
 - K 的左儿子是 $2k$, K 的右儿子是 $2k+1$,
 - K 的父亲是 $\lfloor k/2 \rfloor$



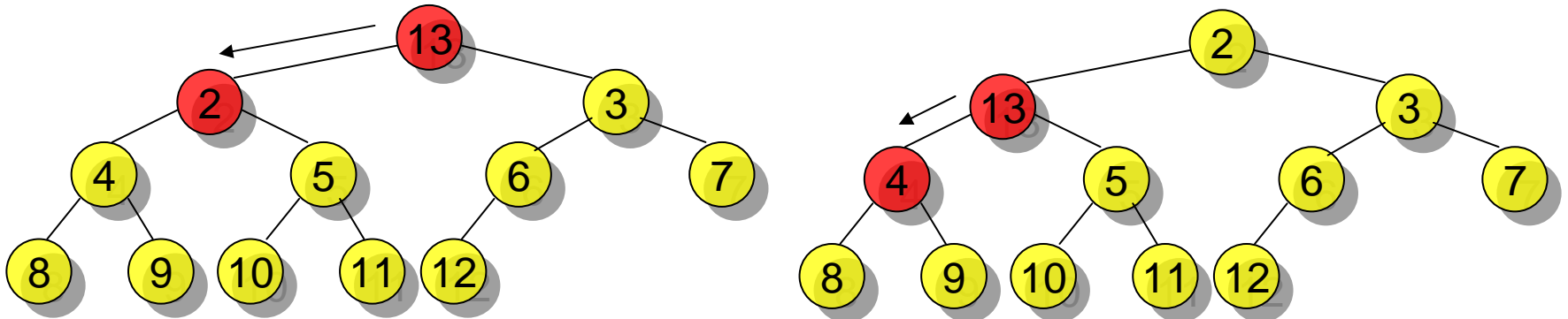
删除最小值元素

- 三步法
 - 直接删除根
 - 用最后一个元素代替根上元素
 - 向下调整



向下调整

- 首先选取当前结点p的较大儿子. 如果比p大, 调整停止, 否则交换p和儿子, 继续调整



```
void swim(int p){  
    int q = p>>1, a = heap[p];  
    while(q && a<heap[q]){ heap[p]=heap[q]; p=q; q=p>>1; }  
    heap[p] = a;  
}
```

插入元素和向上调整

- 插入元素是先添加到末尾, 再向上调整
- 向上调整: 比较当前结点 p 和父亲, 如果父亲比 p 小, 停止; 否则交换父亲和 p , 继续调整

```
void sink(int p){  
    int q=p<<1, a = heap[p];  
    while(q<=hs){  
        if(q<hs&&heap[q+1]<heap[q])q++;  
        if(heap[q]>=a) break;  
        heap[p]=heap[q]; p=q; q=p<<1;  
    }  
    heap[p] = a;  
}
```

堆的建立

- 从下往上逐层向下调整. 所有的叶子无需调整, 因此从 $hs/2$ 开始. 可用数学归纳法证明循环变量为 i 时, 第 $i+1, i+2, \dots, n$ 均为最小堆的根

```
void insert(int a)
{ heap[++hs]=a; swim(hs); }
int getmin()
{ int r=heap[1]; heap[1]=heap[hs--];
  sink(1); return r; }
int decreaseKey(int p, int a)
{ heap[p]=a; swim(p); }
void build()
{ for(int i=hs/2;i>0;i--) sink(i); }
```

时间复杂度分析

- 向上调整/向下调整
 - 每层是常数级别, 共 $\log n$ 层, 因此 $O(\log n)$
- 插入/删除
 - 只调用一次向上或向下调整, 因此都是 $O(\log n)$
- 建堆
 - 高度为 h 的结点有 $n/2^{h+1}$ 个, 总时间为

$$\sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \times O(h) = O \left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \right)$$

例1. k路归并问题

- 把k个有序表合并成一个有序表.
- 元素共有n个.

分析

- 每个表的元素都是从左到右移入新表
- 把每个表的当前元素放入二叉堆中, 每次删除最小值并放入新表中, 然后加入此序列的下一个元素
- 每次操作需要 $\log k$ 时间, 因此总共需要 $n \log k$ 的时间

例2. 序列和的前 n 小元素

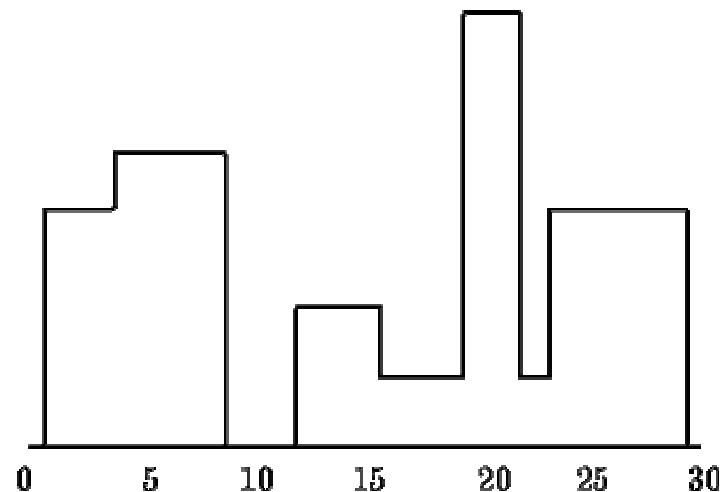
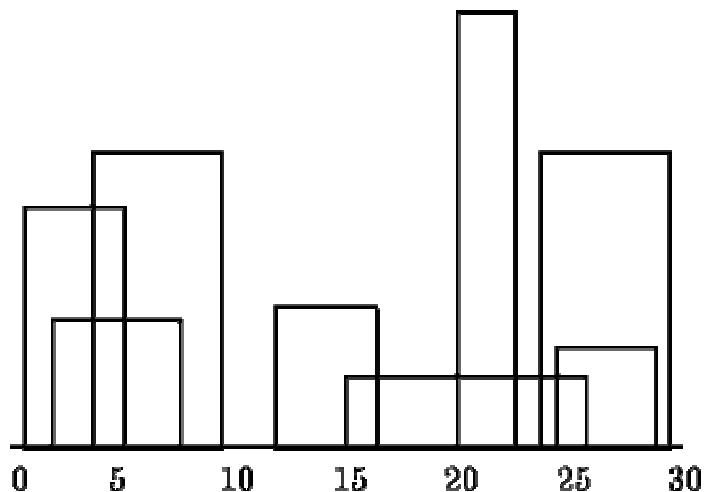
- 给出两个长度为 n 的有序表A和B, 在A和B中各任取一个, 可以得到 n^2 个和. 求这些和最小的 n 个

分析

- 可以把这些和看成 n 个有序表:
 - $A[1]+B[1] \leq A[1]+B[2] \leq A[1]+B[3] \leq \dots$
 - $A[2]+B[1] \leq A[2]+B[2] \leq A[2]+B[3] \leq \dots$
 - ...
 - $A[n]+B[1] \leq A[n]+B[2] \leq A[n]+B[3] \leq \dots$
- 类似刚才的算法, 每次 $O(\log n)$, 共取 n 次最小元素, 共 $O(n \log n)$

例3. 轮廓线

- 每一个建筑物用一个三元组表示(L, H, R), 表示左边界, 高度和右边界
- 轮廓线用X, Y, X, Y...这样的交替式表示
- 右图的轮廓线为: (1, 11, 3, 13, 9, 0, 12, 7, 16, 3, 19, 18, 22, 3, 23, 13, 29, 0)
- 给N个建筑, 求轮廓线



分析

- 算法一：用数组记录每一个元线段的高度
 - 离散化, 有 n 个元线段
 - 每次插入可能影响 n 个元线段, $O(n)$, 共 $O(n^2)$
 - 从左到右扫描元线段高度, 得轮廓线
- 算法二：每个建筑的左右边界为事件点
 - 把事件点排序, 从左到右扫描
 - 维护建筑物集合, 事件点为线段的插入删除
 - 需要求最高建筑物, 用堆, 共 $O(n\log n)$

例4. 丑数

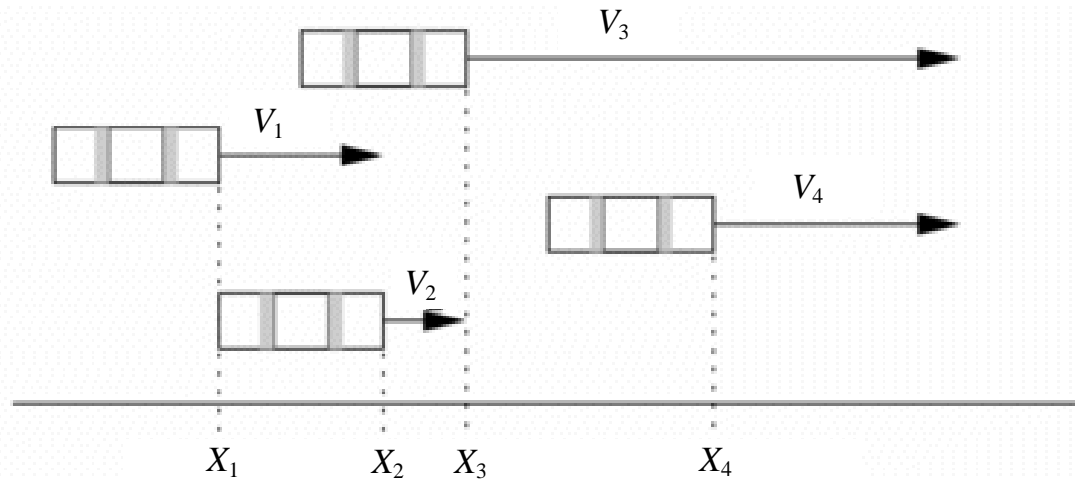
- 素因子都在集合 $\{2, 3, 5, 7\}$ 的数称为ugly number
- 求第n大的丑数

分析

- 初始：把1放入优先队列中
- 每次从优先队列中取出一个元素 k ，把 $2k$, $3k$, $5k$, $7k$ 放入优先队列中
- 从2开始算，取出的第 n 个元素就是第 n 大的丑数
- 每取出一个数，插入4个数，因此任何堆里的元素是 $O(n)$ 的，时间复杂度为 $O(n \log n)$

例5. 赛车

- 有 n 辆赛车从各不相同的地方以各种的速度(速度 $0 < v_i < 100$)开始往右行驶，不断有超车现象发生。



- 给出 n 辆赛车的描述（位置 x_i ，速度 v_i ），赛车已按照位置排序（ $x_1 < x_2 < \dots < x_n$ ）
- 输出超车总数以及按时间顺序的前 m 个超车事件

分析

- 事件个数 $O(n^2)$, 因此只能一个一个求
- 给定两辆车, 超越时刻预先可算出
- 第一次超车可能在哪些辆车之间?
 - 维护所有车的前方相邻车和追上时刻
 - 局部: 此时刻不一定是该车下个超车时刻!
 - 全局: 所有时刻的最小值就是下次真实超车时刻
- 维护: 超车以后有什么变化?
 - 相对顺序变化...改变三个车的前方相邻车
 - 重新算追上时刻, 调整三个权
 - 简单的处理方法: 删除三个再插入三个

例6. 可怜的奶牛

- 农夫John有 n ($n \leq 100\ 000$) 头奶牛，可是由于它们产的奶太少，农夫对它们很不满意，决定每天把产奶最少的一头做成牛肉干吃掉。但还是有一点舍不得，John打算如果不止有一头奶牛产奶最少，当天就大发慈悲，放过所有的牛。
- 由于John的奶牛产奶是周期性的，John在一开始就能可以了解所有牛的最终命运，不过他的数学很差，所以请你帮帮忙，算算最后有多少头奶牛可以幸免于难。每头奶牛的产奶周期 T_i 可能不同，但不会超过10。在每个周期中，奶牛每天产奶量不超过200。

分析

- 如果采用最笨的方法，每次先求出每头牛的产奶量，再求最小值，则每天的复杂度为 $O(n)$ ，总复杂度为 $O(Tn)$ ，其中 T 是模拟的总天数。由于周期不超过10，如果有的牛永远也不会被吃掉，那么我们需要多模拟2520天（1, 2, 3, ..., 10的最小公倍数）才能确定
- 周期同为 t 的奶牛在没有都被吃掉之前，每天的最小产奶量也是以 t 为周期的。因此如果把周期相同的奶牛合并起来，每天只需要比较10类奶牛中每类牛的最小产奶量就可以了，每天的复杂度为 $O(k)$ ，其中 k 为最长周期

分析

- 假设周期为6的牛有4头，每次只需要比较 k 组牛的“代表”就可以了，每天模拟的时间复杂度为 $O(k)$ 。

项 目	第 $6n+1$ 天	第 $6n+2$ 天	第 $6n+3$ 天	第 $6n+4$ 天	第 $6n+5$ 天	第 $6n+6$ 天
牛1	2	5	3	5	7	4
牛2	3	1	6	7	5	4
牛3	5	3	3	5	3	9
牛4	4	4	3	8	8	2
合并结果	2（牛1）	1（牛2）	3（多）	5（多）	3（牛3）	2（牛4）

分析

- 只要周期为6的牛都不被吃掉，这个表一直是有效的。但是在吃掉一头奶牛后，我们需要修改这个表，使它仍然记录着每天的最小产奶量
 - 方法一: 重新计算，时间 $O(h)$ ，其中 h 是该组的牛数
 - 方法二: 把一个周期中每天的最小产奶量组织成堆，每次删除操作的复杂度是 $O(k \log h)$
- 由于每头奶牛最多被吃掉一次，因此用在维护“最小产奶量结构”的总复杂度不超过 $O(nk \log n)$ 。每天复杂度为 $O(k)$ ，总复杂度为 $O(Tk + nk \log n)$

例7. 黑匣子

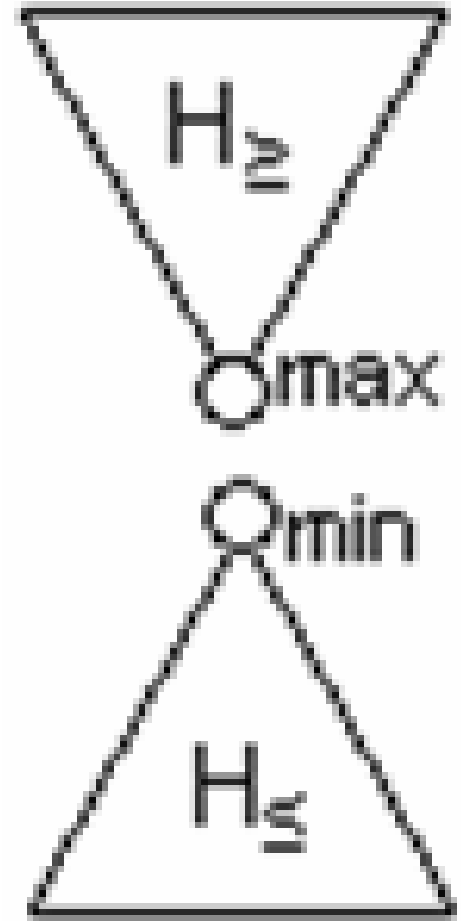
- 我们使用黑匣子的一个简单模型。它能存放一个整数序列和一个特别的变量 i 。在初始时刻，黑匣子为空且 i 等于0。这个黑匣子执行一序列的命令。有两类命令：
- **ADD(x)**: 把元素 x 放入黑匣子；
- **GET**: i 增1的同时，输出黑匣子内所有整数中第 i 小的数。牢记第 i 小的数是当黑匣子中的元素以非降序排序后位于第 i 位的元素

例7. 黑匣子

编号	命令	i	黑匣子内容	输出
1	ADD(3)	0	3	
2	GET	1	3	3
3	ADD(1)	1	1, 3	
4	GET	2	1, 3	3
5	ADD(-4)	2	-4, 1, 3	
6	ADD(2)	2	-4, 1, 2, 3	
7	ADD(8)	2	-4, 1, 2, 3, 8	
8	ADD(-1000)	2	-1000, -4, 1, 2, 3, 8	
9	GET	3	-1000, -4, 1 , 2, 3, 8	1
10	GET	4	-1000, -4, 1, 2 , 3, 8	2
11	ADD(2)	4	-1000, -4, 1, 2, 2, 3, 8	

分析

- 降序堆 H_{\geq} 和升序堆 H_{\leq} 如图放置
- H_{\geq} 根节点的值 $H_{\geq}[1]$ 在堆 H_{\geq} 中最大,
 H_{\leq} 根节点的值 $H_{\leq}[1]$ 在堆 H_{\leq} 中最小,
并满足
 - $H_{\geq}[1] \leq H_{\leq}[1]$
 - $\text{size}[H_{\geq}] = i$
- **ADD(x):** 比较 x 与 $H_{\geq}[1]$, 若 $x \geq H_{\geq}[1]$, 则将 x 插入 H_{\leq} , 否则从 H_{\geq} 中取出 $H_{\geq}[1]$ 插入 H_{\leq} , 再将 x 插入 H_{\geq}
- **GET:** $H_{\leq}[1]$ 就是待获取的对象。输出 $H_{\leq}[1]$, 同时从 H_{\leq} 中取出 $H_{\leq}[1]$ 插入 H_{\geq} , 以维护条件(2)



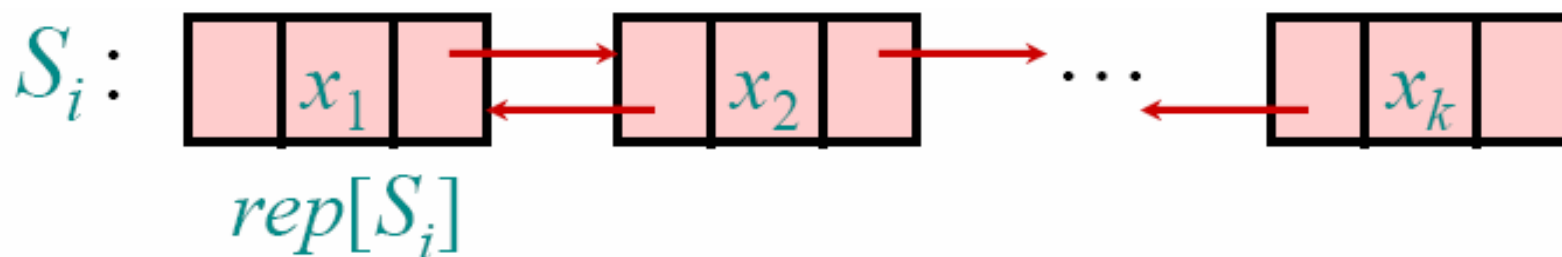
三、并查集

并查集

- 并查集维护一些不相交集合 $S = \{S_1, S_2, \dots, S_r\}$, 每个集合 S_i 都有一个特殊元素 $\text{rep}[S_i]$, 称为集合代表. 并查集支持三种操作
 - **Make-Set(x)**: 加入一个集合 $\{x\}$ 到 S , 且 $\text{rep}[\{x\}] = x$. 注意, x 不能被包含在任何一个 S_i 中, 因为 S 里任何两个集合应是不相交的
 - **Union(x, y)**: 把 x 和 y 所在的两个不同集合合并. 相当于从 S 中删除 S_x 和 S_y 并加入 $S_x \cup S_y$
 - **Find-Set(x)**: 返回 x 所在集合 S_x 的代表 $\text{rep}[S_x]$

链结构

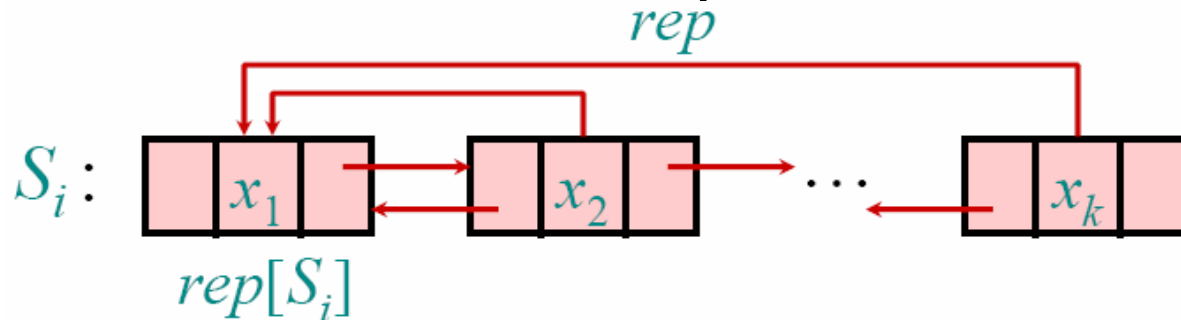
- 每个集合用双向链表表示, $\text{rep}[S_i]$ 在链表首部



- **Make-Set(x):** 显然是 $O(1)$ 的
- **Find-Set(x):** 需要不断往左移, 直到移动到首部. 最坏情况下是 $O(n)$ 的
- **Union(x, y):** 把 S_y 接在 S_x 的尾部, 代表仍是 $\text{rep}[S_x]$. 为了查找链表尾部, 需要 $O(n)$

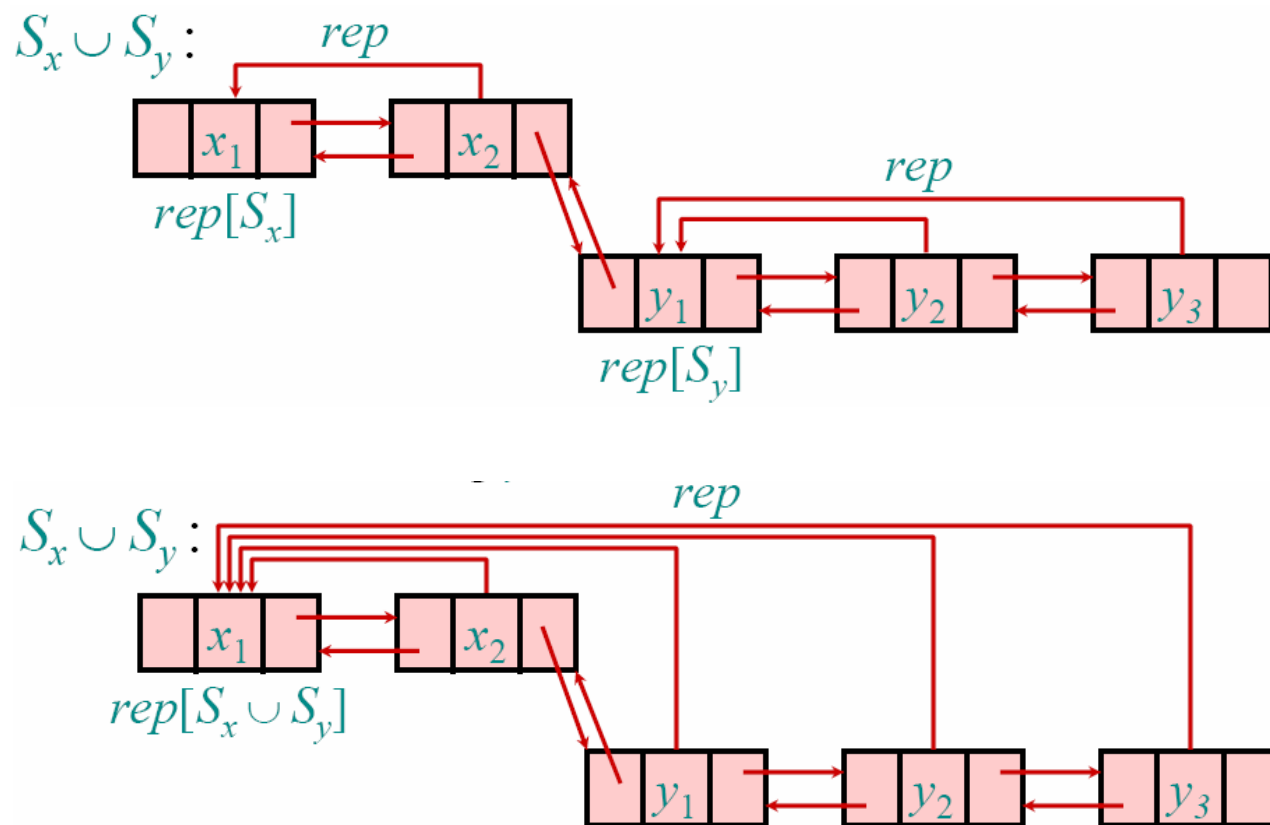
增强型链结构

- 给每个结点增加一个指回rep的指针



- Make-Set(x):** 仍为常数
- Find-Set(x):** 降为常数(直接读rep)
- Union(x, y):** 变得复杂: 需要把 S_y 里所有元素的rep指针设为 $rep[S_x]$!

增强型链结构的合并



- 可以把 x 合并到 y 中，也可以把 y 合并在 x 中

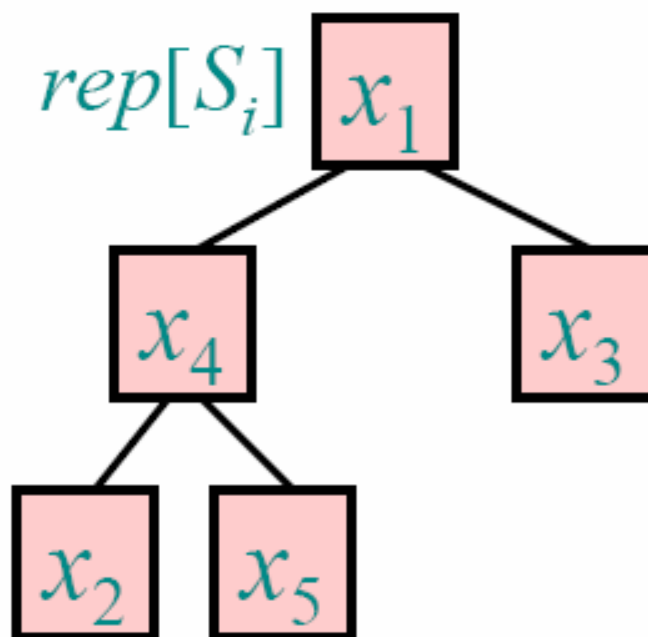
技巧1: 小的合并到大的中

- 显然, 把小的合并到大的中, 这一次Union操作会比较节省时间, 更精确的分析?
- 用 n , m , f 分别表示Make-Set的次数, 总操作次数和Find-Set的次数, 则有
- **定理:** 所有Union的总时间为 $O(n \log n)$
- **推论:** 所有时间为 $O(m + n \log n)$
- **证明:** 单独考虑每个元素 x , 设所在集合为 S_x , 则修改 $\text{rep}[x]$ 时, S_x 至少加倍. 由于 S_x 不超过 n , 因此修改次数不超过 $\log_2 n$, 总 $n \log n$

树结构

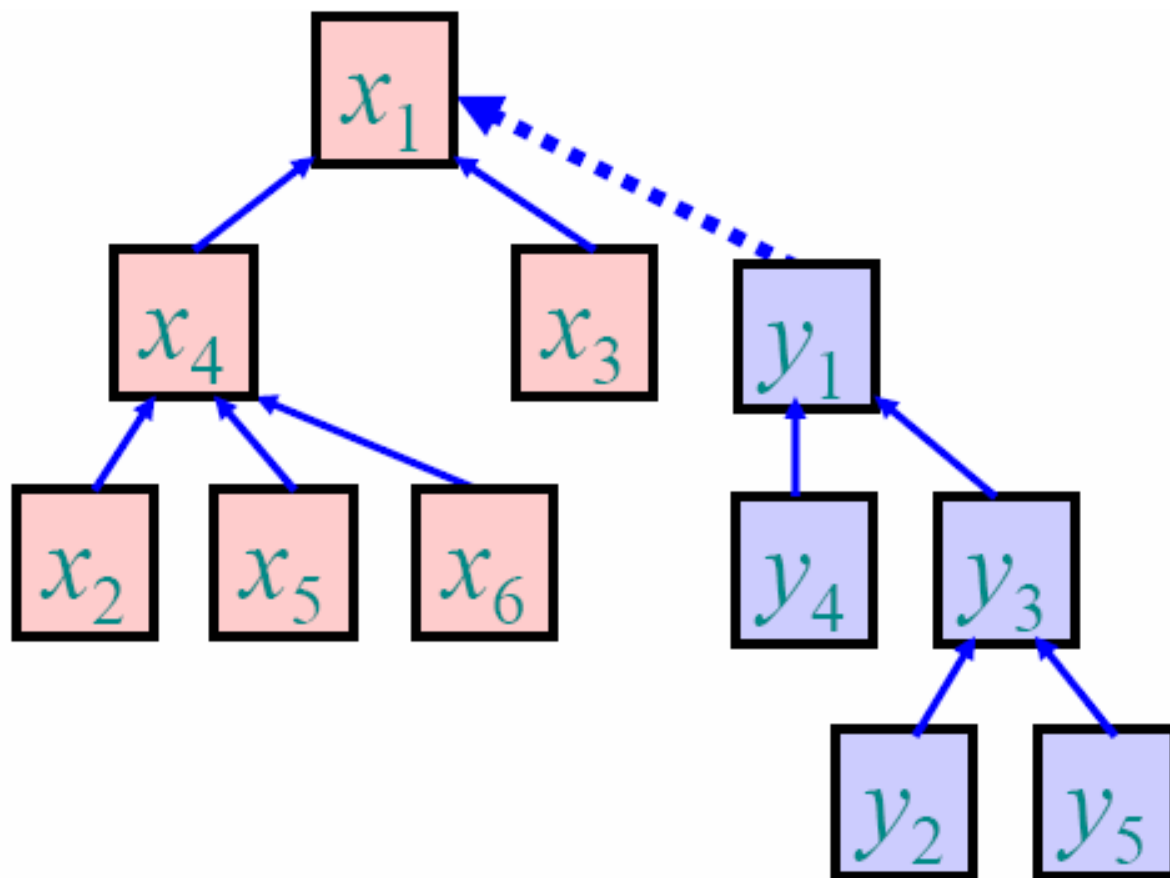
- 每个集合用一棵树表示, 根为集合代表

$$S_i = \{x_1, x_2, x_3, x_4, x_5\}$$



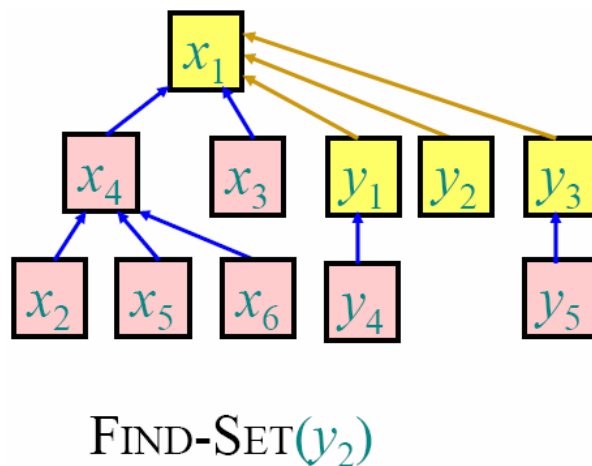
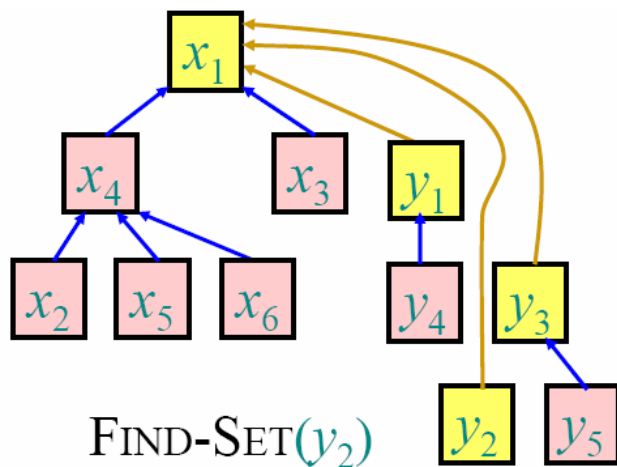
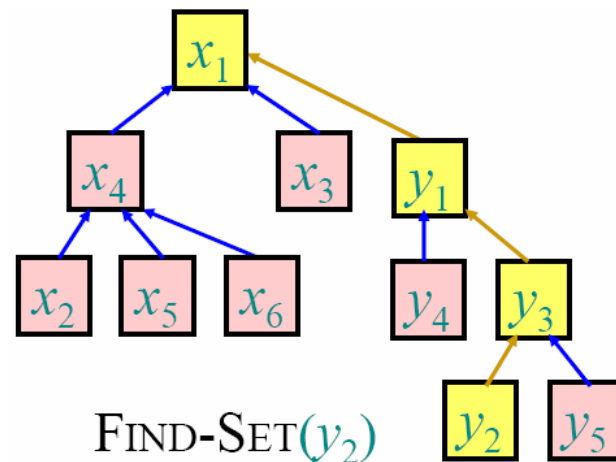
树结构的合并

- 和链结构类似, 小的合并到大的中



技巧2: 路径压缩

- 查找结束后顺便把父亲设置为根, 相当于有选择的设置rep指针而不像链结构中强制更新所有rep



路径压缩的分析

- 设 $w[x]$ 为 x 的子树的结点数, 定义势能函数
$$\phi(x_1, \dots, x_n) = \sum_i \lg \text{weight}[x_i]$$
- **Union(x_i, x_j)**增加势能. 最多会让 $w[\text{rep}[x_i]]$ 增加 $w[\text{rep}[x_j]] \leq n$, 因此势能增加不超过 $\lg n$
- **Find-Set(x)**减少势能. 把路径压缩看作是从根到结点 x 的向下走过程, 则除了第一次外的其他向下走的步骤 $p \rightarrow c$ 会让 c 的子树从 p 的子树中移出, 即 $w[p]$ 减少 $w[c]$, 而其他点的 w 值保持不变

路径压缩的分析

- **Find-Set**除了第一次外的其他向下走的步骤 $p \rightarrow c$ 会让 c 的子树从 p 的子树中移出
 - 情况一: $w[c] \geq w[p]/2$, 则势能将至少减少1
 - 情况二: $w[c] < w[p]/2$, 这种情况最多出现 $\log n$ 次, 因为 $w[p]$ 最多进行 $\log n$ 次除2操作就会得到1
- **Union**操作积累起来的 $m \log n$ 的势能将被 **Find-Set** 消耗, 情况一最多消耗 $m \log n$ 次, 情况二本身不超过 $m \log n$ 次, 因此
- **定理:** Find-Set 的总时间为 $O(m \log n)$

路径压缩的分析

- **定理:** 如果所有Union发生在Find-Set之前, 则所有操作的时间复杂度为 $O(m)$
- **证明:** 每次Find-Set将会让路径上除了根的所有结点为根的儿子. 所有结点只会有一次改变, 因此总时间复杂度为 $O(m)$
- 也就是说
 - 只使用技巧1(启发式合并): $O(m+n\log n)$
 - 只使用技巧2(路径压缩): $O(m\log n)$
- 同时使用呢?

Ackermann函数及其反函数

Define $A_k(j) = \begin{cases} j+1 & \text{if } k=0, \\ A_{k-1}^{(j+1)}(j) & \text{if } k \geq 1. \end{cases}$ – iterate $j+1$ times

$$A_0(j) = j + 1$$

$$A_0(1) = 2$$

$$A_1(j) \sim 2j$$

$$A_1(1) = 3$$

$$A_2(j) \sim 2j \cdot 2^j > 2^j$$

$$A_2(1) = 7$$

$$A_3(1) = 2047$$

$$A_3(j) > 2^{2^{2^{\dots^{2^j}}}} \quad \left. \vphantom{2^{2^{2^{\dots^{2^j}}}}} \right\} j$$

$A_4(j)$ is a lot bigger.

$$A_4(1) > 2^{2^{2^{\dots^{2^{2047}}}}} \quad \left. \vphantom{2^{2^{2^{\dots^{2^{2047}}}}} \right\} 2048$$

Define $\alpha(n) = \min \{k : A_k(1) \geq n\} \leq 4$ for practical n .

树结构的完整结论

- **定理:** m 个操作的总时间复杂度为 $O(m\alpha(n))$

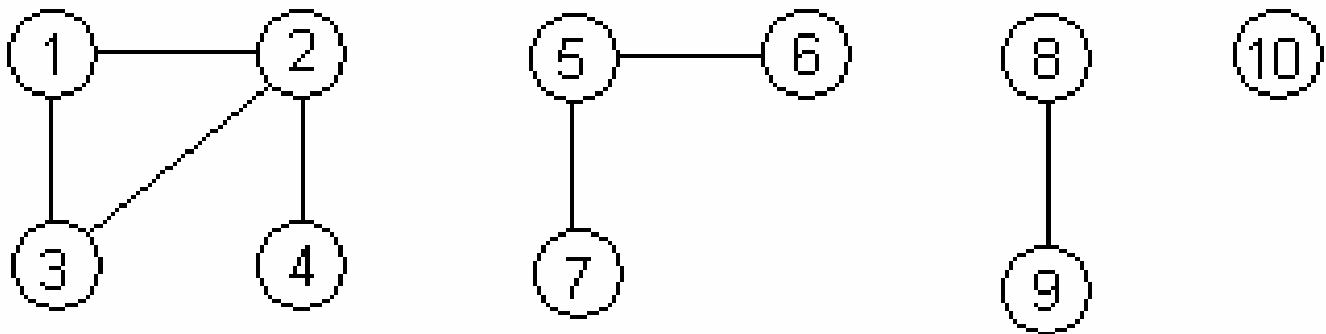
```
void makeset(int x){ rank[x] = 0; p[x]=x; }
int findset(int x){
    int i, px = x;
    while (px != p[px]) px = p[px];
    while (x != px) { i = p[x]; p[x] = px; x = i; }
    return px;
}
void unionset (int x , int y){
    x = findset(x); y = findset(y);
    if(rank[x] > rank[y]) p[y] = x;
    else { p[x] = y; if(rank[x] == rank[y]) rank[y]++; }
}
```

例1. 亲戚

- 或许你并不知道，你的某个朋友是你的亲戚。他可能是你的曾祖父的外公的女婿的外甥女的表姐的孙子。如果能得到完整的家谱，判断两个人是否亲戚应该是可行的，但如果两个人的最近公共祖先与他们像个好几代，使得家谱十分庞大，那么检验亲戚关系实非人力所能及。在这种情况下，最好的帮手就是计算机。
- 为了将问题简化，你将得到一些亲戚关系的信息，如同Marry和Tom是亲戚，Tom和Ben是亲戚，等等。从这些信息中，你可以推出Marry和Ben是亲戚。请写一个程序，对于我们的关于亲戚关系的提问，以最快的速度给出答案。

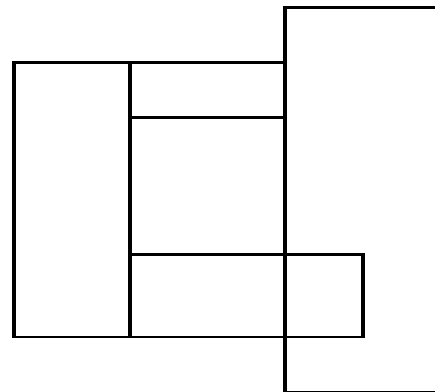
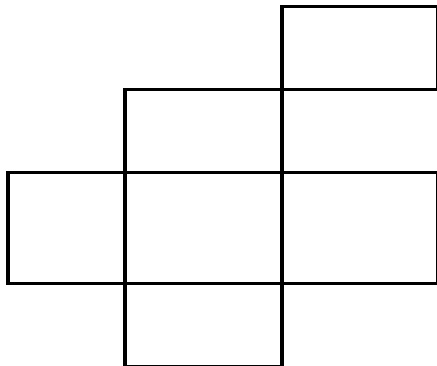
分析

- 本质: 是否在图的同一个连通块
- 问题: 图太庞大, 每次还需要遍历
- 解决: 用并查集



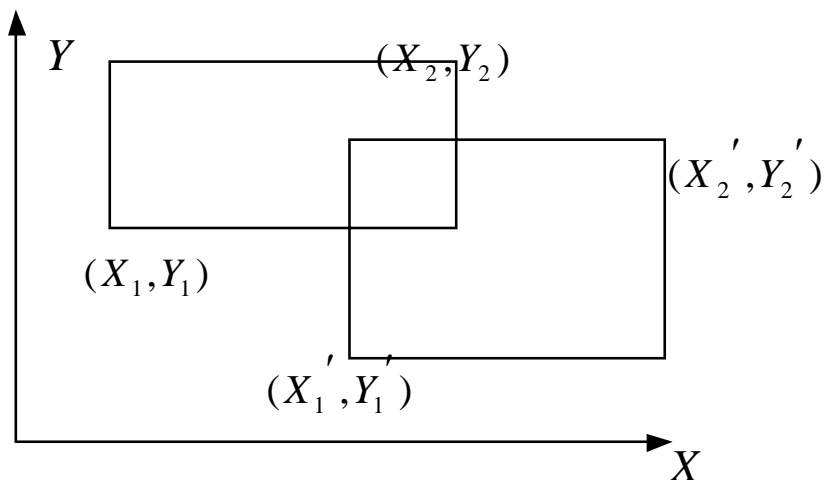
例2. 矩形

- 在平面上画了N个长方形，每个长方形的边平行于坐标轴并且顶点坐标为整数。我们用以下方式定义印版：
 - 每个长方形是一个印版；
 - 如果两个印版有公共的边或内部，那么它们组成新的印版，否则这些印版是分离的
- 数出印版的个数. 左图有两个，右图只有一个



分析

- 把矩形看作点，有公共边的矩形连边，问题转化为求连通分量的个数
- 判断方法：



例3. 代码等式

- 由元素0和1组成的非空的序列称为一个二进制代码。一个代码等式就是形如 $x_1x_2\ldots x_l=y_1y_2\ldots y_r$, 这里 x_i 和 y_j 是二进制的数字（0或1）或者是一个变量（如英语中的小写字母）
- 每一个变量都是一个有固定长度的二进制代码，它可以在代码等式中取代变量的位置。我们称这个长度为变量的长度
- 对于每一个给出的等式，计算一共有多少组解。
- 例: a,b,c,d,e的长度分别是4,2,4,4,2, 则 $1bad1 = acbe$ 有16组解

分析

- 长度为 k 的变量拆成 k 个长度为1的变量
- 每位得到一个等式
 - $1=1$ 或者 $0=0$: 冗余等式
 - $1=0$ 或者 $0=1$: 无解
 - $a=b$: a 和 b 相等 (a 为变量 b 可以为常数)
- 相等关系用并查集处理, 最后统计集合数为 n , 答案为 2^n 。

例4. 围墙

- 按顺序给出 M 个整点组成的线段，找到最小的 k ，使得前 k 条线段构成了封闭图形。
（任意两条线段只可能在端点相交）

分析

- 将所有出现过的坐标用整数表示，初始时候每个独立成树。读入连接**A**和**B**的线段后，将**A**、**B**所在的树和并。如果**A**、**B**在同一棵树，那么就出现了封闭图形（因为**x**个点**x**条边的图必定出现圈）
- 把坐标转换成编号的步骤，可以通过对坐标进行排序，再删除重复。
- 时间: $O(M\log M)$

例5. 可爱的猴子

- 树上挂着 n 只可爱的猴子（ $n \leq 2 \cdot 10^5$ ）。
- 猴子1的尾巴挂在树上
- 每只猴子有两只手，每只手可以抓住最多一只猴子的尾巴，也可以不抓。猴子想抓谁一定抓得到
- 所有猴子都是悬空的，因此如果一旦脱离了树，猴子会立刻掉到地上。
- 第0, 1, ..., m （ $1 \leq m \leq 400\,000$ ）秒中每一秒都有某个猴子把他的某只手松开，因此常有猴子掉在地上
- 请计算出每个猴子掉到地上的时间

分析

- 并查集？
- “时光倒流”
- 如何标记每只猴子的时间？
 - 枚举并查集的元素
 - 需要访问兄弟/儿子？
 - 链表即可
 - 不用指针

例6. 奇数偶数

- 你的朋友写下一个由0和1组成的字符串，并告诉你一些信息，即某个连续的子串中1的个数是奇数还是偶数。你的目标是找到尽量小的 i ，使得前 $i+1$ 条不可能同时满足
 - 例如，序列长度为10，信息条数为5
 - 5条信息分别为1 2 even, 3 4 odd, 5 6 even, 1 6 even, 7 10 odd
- 正确答案是3，因为存在序列(0,0,1,0,1,1)满足前3条信息，但是不存在满足前4条的序列

分析

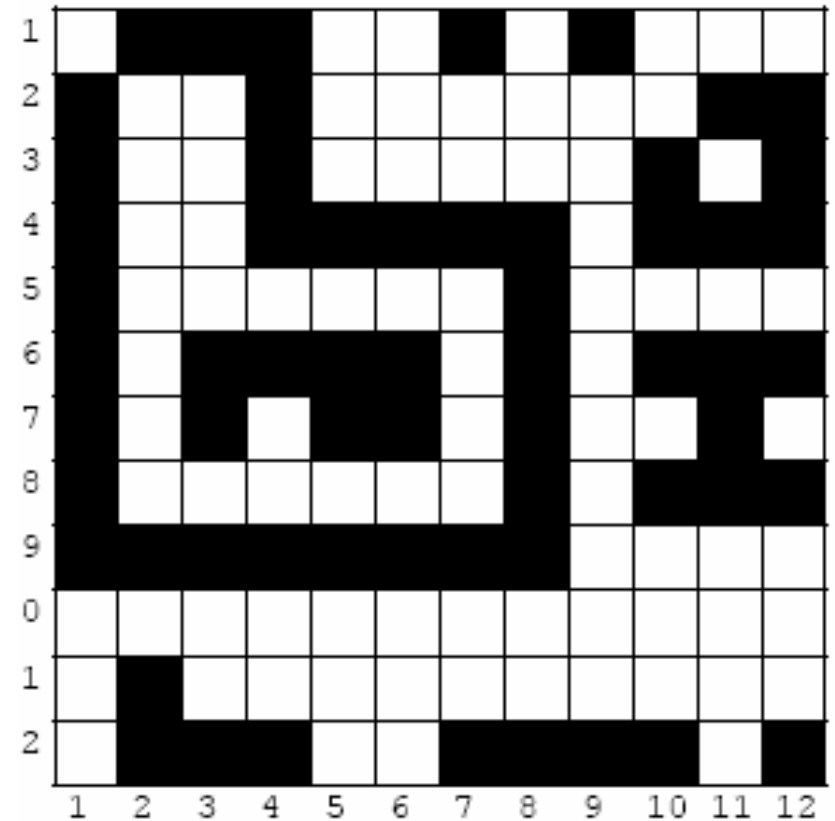
- 部分序列
 - 可以从s的奇偶性恢复整个序列
 - $a \ b \text{ even}$ 等价于 $s[b]$, $s[a-1]$ 同奇偶
 - $a \ b \text{ odd}$ 等价于 $s[b]$, $s[a-1]$ 不同奇偶
- 一开始每个 $s[i]$ 自成一集合
 - $a \ b \text{ even} \rightarrow$ 合并 $s[b]$, $s[a-1]$
 - $a \ b \text{ odd} \rightarrow ???$
- 矛盾的标志?

例7. 团伙

- 如果两个认识, 那么他们要么是朋友要么是敌人, 规则如下
 - 朋友的朋友是朋友
 - 敌人的敌人是敌人
- 给出一些人的关系(朋友、敌人), 判断一共最多可能有多少个团伙

例8. 船

- 给一个01矩阵, 黑色代表船. 右图有一个29吨的, 3个7吨的, 两个4吨的和三个一吨的.
- 输入行数 $N(<30000)$ 和每行的黑色格子(区间数和每个区间)
- 输出每种重量的个数
- 一共不超过1000个船, 每个的重量不超过1000



例9. 离线最大值

- 设计一个集合, 初始为空, 每次可以插入一个 $1 \sim n$ 的数($1 \sim n$ 各恰好被插入一次), 也可以删除最大值, 要求 m 次操作的总时间尽量小.

分析

- 在最后加入 $n-m$ 次虚拟的MAX操作, 并记第 i 个MAX操作为 M_i , 记 M_1 之前的插入序列为 S_1 , M_{i-1} ($1 < i \leq m$) 和 M_i 之间的插入序列为 S_i
- 如果 n 在 S_j 中被插入, 则 M_j 的输出一定是 n . 然后删除 M_j , 即把 S_j 合并到 S_{j+1} 中, 然后再查找 $n-1$ 所在的序列 S_k , 则 M_k 的输出为 $n-1$...如此下去, 从 n 到 1 依次查找每个数所在序列, 就可以得到它后面的MAX操作的结果, 并把它和紧随其后的序列合并

例10. 合并队列

- 初始时 n 个数 $1 \sim n$ 各在单独的一列中, 需要执行两个操作
 - **Move**(i, j): 把 i 所在列接到 j 所在列的尾部
 - **Check**(i, j): 询问 i 和 j 是否在同一列, 如果是, 输出二者之间的元素个数

分析

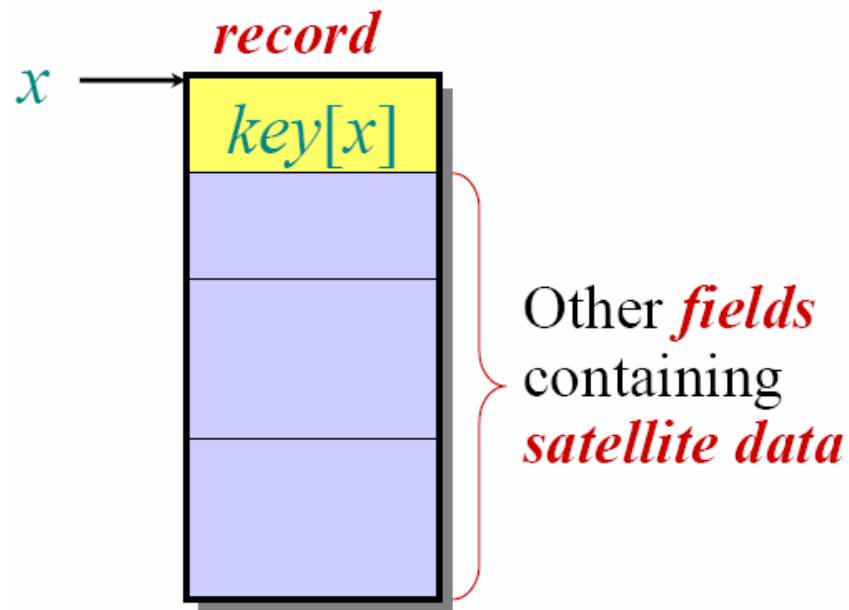
- 每个数 i 的 $p[i]$ 表示 i 和 $p[i]$ 在同一队列, 且 $p[i]$ 是 i 之前的第 $d[i]$ 个元素
- 对于队首 x , 有 $p[x]=x$, 附加变量 $tot[x]$ 表示以 x 为首的队列一共有多少个元素
 - Move需要进行两次查找和一次合并
 - Check需要两次查找
- FIND: 修改 $p[i]$ 时要修改 $d[i]$
- MERGE: 可以启发式合并么???

四、哈希表

哈希表

- 哈希表(Hash table)经常被用来做字典(dictionary), 或称符号表(symbol-table)

Symbol table T holding n *records*:



Operations on T :

- INSERT(T, x)
- DELETE(T, x)
- SEARCH(T, k)

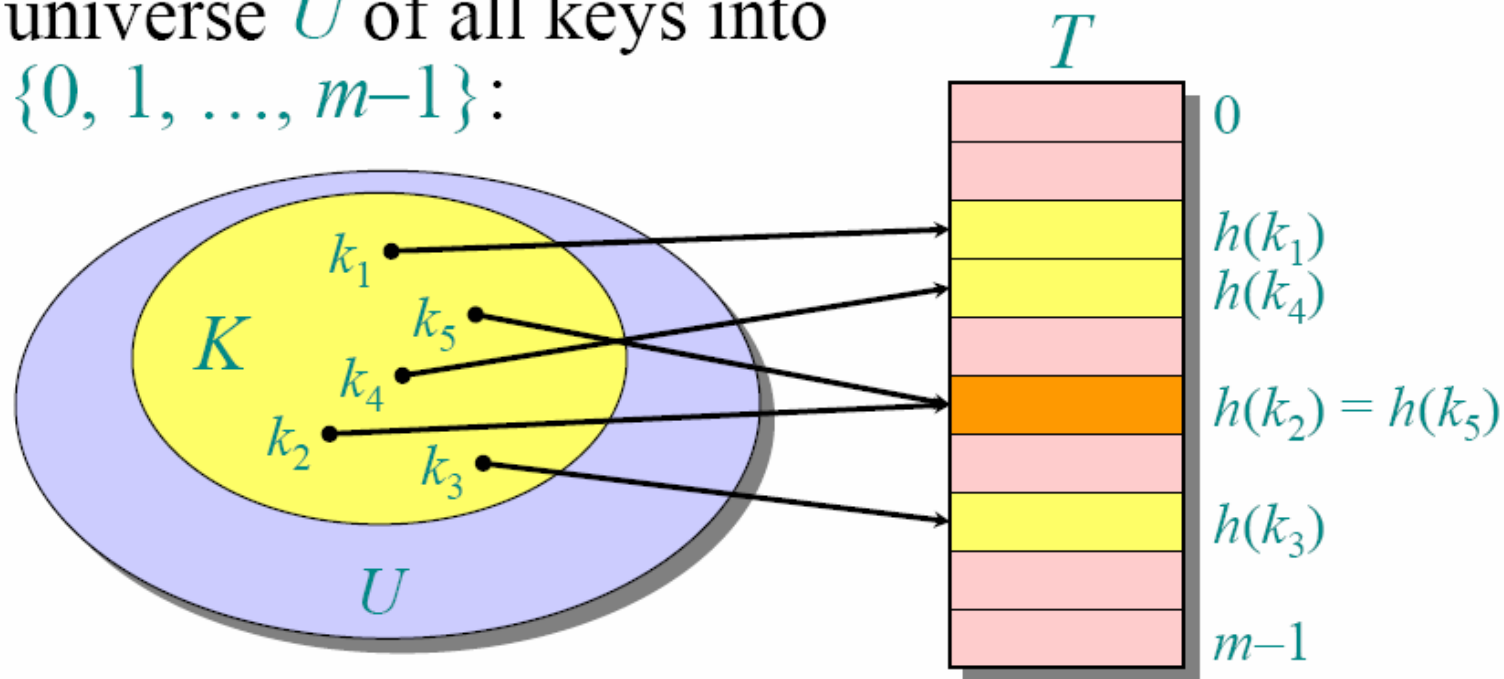
How should the data structure T be organized?

直接存取表

- 直接存取表(Direct-access table)的基本思想是: 如果key的范围为 $0 \sim m-1$ 而且所有key都不相同, 那么可以设计一个数组 $T[0..m-1]$, 让 $T[k]$ 存放key为k的元素, 否则为空(NIL)
- 显然, 所有操作都是 $O(1)$ 的
- **问题:** key的范围可能很大! 64位整数有18,446,744,073,709,551,616种可能, 而字符串的种类将会更多!

解决方案: 哈希函数

Solution: Use a *hash function* h to map the universe U of all keys into $\{0, 1, \dots, m-1\}$:



When a record to be inserted maps to an already occupied slot in T , a *collision* occurs.

哈希函数的选取

- 严格均匀分布的哈希函数很难寻找, 但一般说来有三种方法在实际使用中效果不错
 - 取余法: 取 $h(k) = k \bmod m$
 - 乘积法: 取 $h(k) = (A * k \bmod 2^w) \text{ rsh } (w-r)$
 - 点积法: 随机向量 a 和 k 的 m 进制向量做点积
- 实践中经常采用取余法

取余法

- 一般来说不要取 $m=2^r$, 因为它只取决于 k 的后 r 位

If $k = 1011000111\underbrace{011010}_2$ and $r = 6$, then
 $h(k) = 011010_2$.

- 一般取 m 为不太接近2或10的幂

乘积法

- 取 $m=2^r$, 计算机字长为 w , 则

$$H(k) = (A * k \bmod 2^w) \text{ rsh } (w-r)$$

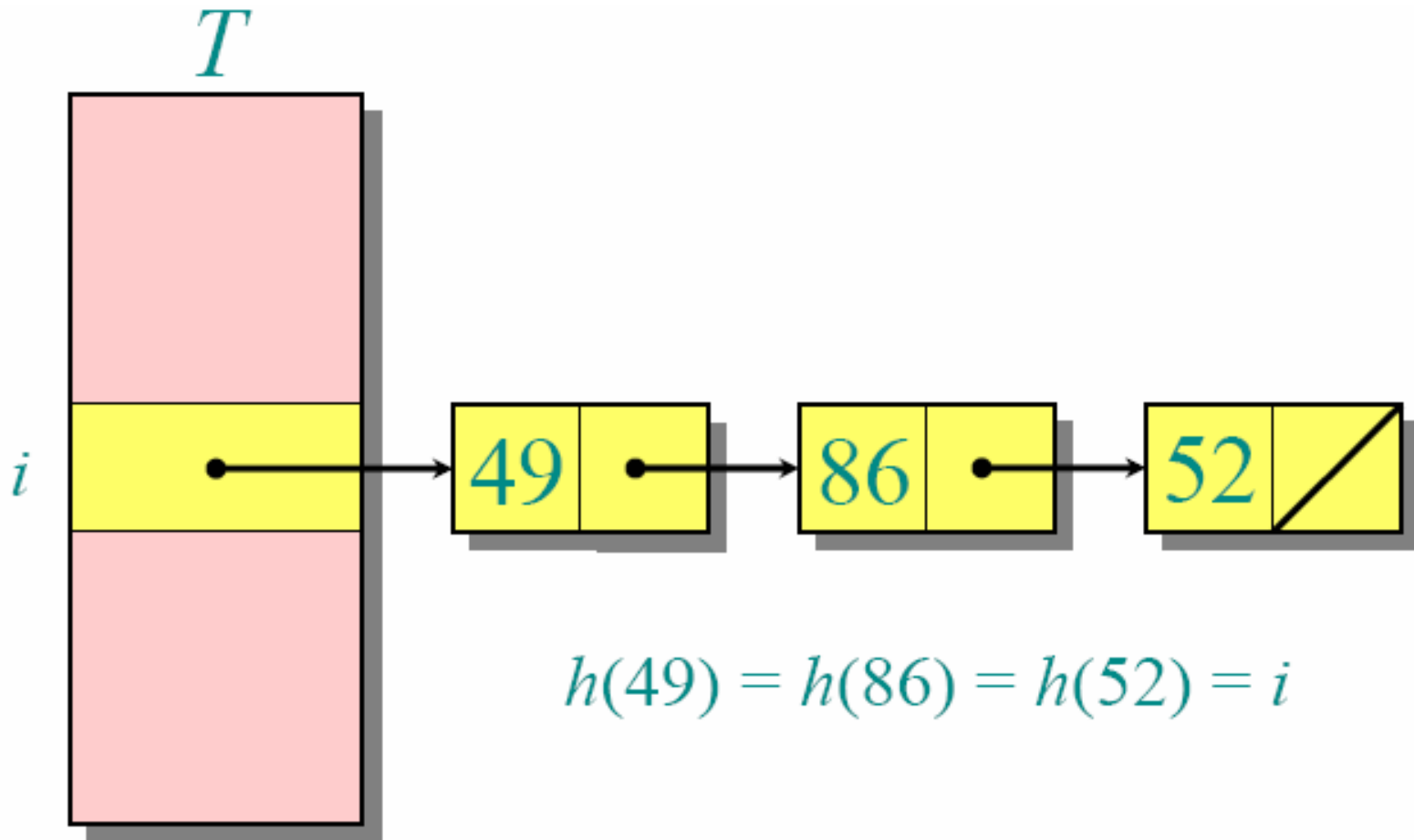
- 其中rsh表示右移位, A 是满足 $2^{w-1} < A < 2^w$ 的奇数. 注意: A 不应该太接近于 2^w . 由于乘法并对 2^w 取余是很快的(自然就会丢弃高位), 而算术右移也很快, 因此函数计算开销小

冲突解决

- 不同的key映射到同一个数, 称为冲突 (collision), 冲突的两个元素显然不可能放在哈希表的同一个位置
- 通常有两种冲突解决方案(resolving collisions)
 - 链方法(chaining): 把key相同的串成链表
 - 开放地址法(open addressing): 自己的位置被占了, 就去占别人的

链地址法

- Key相同的元素形成一个链表



链地址法的查找效率

- 链地址法的时间效率取决于hash函数的分布. 我们假设每个k将等可能的被映射到任意一个slot, 不管其他key被映射到什么地方
- 设n为key的数目, m为不同的slot数, 装载因子 $\alpha = n/m$, 即每个slot平均的key数, 则

Expected time to search for a record with a given key = $\Theta(1 + \alpha)$.

*apply hash
function and
access slot*

*search
the list*

- $n=O(m)$ 时期望搜索时间是 $O(1)$ 的

开放地址法

- 开放地址法只使用表内的空间. 如果冲突产生, 计算出第二个可能的位置, 如果那里也有其他元素, 再看第三个可能的位置...即按一个探测序列查找
- 位置应该是key和探测的次数(probe number)的函数, 第i次探测位置为 $\text{slot} = h(k, i)$
- 每个slot都应能被探测到, 因此每个k的探测序列
 $\langle h(k, 0), h(k, 1), h(k, 2), \dots, h(k, m-1) \rangle$
- 都应是 $\{0, 1, \dots, m-1\}$ 的排列
- 注意: 开放地址法不容易删除元素

开放地址法示例

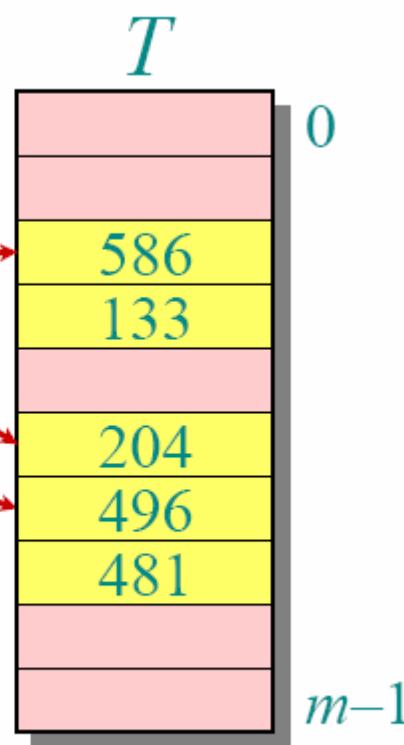
Search for key $k = 496$:

0. Probe $h(496,0)$

1. Probe $h(496,1)$

2. Probe $h(496,2)$

Search uses the same probe sequence, terminating successfully if it finds the key and unsuccessfully if it encounters an empty slot.



探测方法

- 线性探测:

$$h(k,i) = (h'(k) + i) \bmod m.$$

- 虽然很简单, 但是容易形成元素堆积
- 二次哈希: 组合两个哈希函数

$$h(k,i) = (h_1(k) + i \cdot h_2(k)) \bmod m.$$

- 一般效果会好很多, 但应保证 $h_2(k)$ 和 m 互素, 比如取 m 为2的幂而让 $h_2(k)$ 只产生奇数

开放地址法的查找效率

- 假设每个key等概率的把 $m!$ 个排列中的任何一个作为它的探测序列, 则有
- **定理:** 装载因子 $\alpha < 1$ 时, 不成功查找的期望探测次数为 $1/(1-\alpha)$
- 证明: 第 i 次探测到非空slot的概率为
$$(n-i)/(m-i) < n/m = \alpha$$
- 下面各种情况按概率加权计算期望

开放地址法的查找效率

- 各种情况按概率加权, 得期望的探测次数为

$$1 + \frac{n}{m} \left(1 + \frac{n-1}{m-1} \left(1 + \frac{n-2}{m-2} \left(\dots \left(1 + \frac{1}{m-n+1} \right) \dots \right) \right) \right)$$

$$\leq 1 + \alpha (1 + \alpha (1 + \alpha (\dots (1 + \alpha) \dots)))$$

$$\leq 1 + \alpha + \alpha^2 + \alpha^3 + \dots$$

$$= \sum_{i=0}^{\infty} \alpha^i$$

$$= \frac{1}{1-\alpha} . \quad \square$$

The textbook has a more rigorous proof.

查找效率比较

- 把开放地址法说通俗一点，就是
 - 装载因子为常数时，期望查找次数为常数
 - 一半装满时，期望查找次数为 $1/(1-0.5)=2$
 - 装满90%时，期望查找次数为 $1/(1-0.9)=10$
- 装得比较满(尤其是 $n>m$)时，使用链地址法
 - 在哈希表里保存每个key的第一个元素
 $\text{first}[0..m-1]$,
 - 在一个数组 $\text{data}[1..n]$ 里装着所有元素和下一个元素 $\text{next}[1..n]$

链地址法参考代码

- 在实际应用中，推荐使用链地址法
 - first[i]表示哈希函数值为i的第一个数据下标
 - key[i]和next[i]表示第i个数据的key和下一个

```
int find(int k){
    int h = hash(k);
    int p = first[h];
    while (p){ if(key[p] == k) return p; p = next [p]; }
    return 0;
}

void insert(int x){
    int h = hash(key[x]); next[x] = first[h]; first[h] = x;
}
```